NIAKWA PROGRAMMING LANGUAGE

TECHNICAL REFERENCE GUIDE

PROGRAMMER'S GUIDE

1st Edition - July 1993
COPYRIGHT © 1993 Niakwa, Inc.

Niakwa, Inc.
23600 N. Milwaukee Avenue
Mundelein, IL 60060

PHONE: (708) 634-8700 FAX (708) 634-8718 TELEX 3719965 NIAK UB

DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITIES AND PROPRIETARY RIGHTS

The staff of Niakwa, Inc. (Niakwa), has taken due care in preparing this guide. Nothing contained herein shall be construed to modify or alter in any way the standard terms and conditions of the Niakwa Programming Language (NPL) Support and Distribution License Agreement, the End-User Support Only License Agreement, the Niakwa Software License Agreement and Warranty and any other Niakwa License Agreement (collectively, the "License Agreements") by which this software package was acquired.

This manual is to serve as a guide for use of the Niakwa software only and not as a source of representations or additional undertakings by Niakwa. The licensee must refer to the License Agreements for Niakwa product and service representations.

No ownership of Niakwa software is transferred by any of the License Agreements. Any use of Niakwa software beyond the terms and conditions of the License Agreements, without the written authorization of Niakwa, is prohibited.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without prior written permission from Niakwa, Inc.

Niakwa is a registered trademark of Niakwa Management Services 1975 Ltd., and is licensed to Bluebird Systems.

Niakwa Programming Language (NPL), Bluebird and SuperDOS are registered trademarks of Bluebird Systems.

All other trademarks are property of their respective holders.

NIAKWA SOFTWARE LICENSE AGREEMENT AND WARRANTY

The Niakwa Software is licensed on the condition that the Licensee agrees to the following terms and conditions. If you do not agree with them, return the unopened package to Niakwa or your distributor, and your money will be promptly refunded. OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS.

Niakwa agrees to grant, and the Licensee agrees to accept, a non-exclusive, non-transferable license to use the Niakwa Software herein contained under the following terms and conditions:

1. Grant of License:
A. Upon payment by Licensee of the Software License fee, Licensee may use Niakwa Software. Licensee's right to use the Niakwa Software is restricted to use on a single C.P.U. for the appropriate number of users as indicated on the RunTime Package, or in the case of multi-user software on a Niakwa-approved network, to use on a single network for the appropriate number of users as indicated on the RunTime Package. The RunTime Software may be used temporarily in conjunction with a backup C.P.U. or network only in the event the primary computer or network becomes involuntarily inoperative. Licensee shall not use the RunTime Software for internal development use unless Licensee has paid the Development Package fee.

B. In the case of certain Niakwa products, including but not limited to the Niakwa Data Manager and certain Niakwa libraries, the licensee is granted an unlimited use license of the Niakwa Software, provided licensee's software is also installed on the same CPU or network as the Niakwa Software. Only those Niakwa products indicated "Unlimited Use License" on the diskette label may be used on an unlimited use basis.

C. At each of its wholly owned branches that has properly licensed the Niakwa RunTime Package, a Licensee may install Development Software FOR SUPPORT PURPOSES ONLY. The installation of Development Software at any branch is for purposes of on-site application software support (bug corrections) only. The Development Software shall not be used at any branch for software development of any kind. Licensee shall be responsible for informing all branch personnel of the limited use for which the Development Software is installed at the branch facility and shall ensure that such limited use is observed by all branch personnel.

2. Additional Terms On Use:
Licensee may merge the Niakwa Software into other software; however, Licensee has no right to modify any item of Niakwa Software which is supplied to it unless specifically authorized in writing by Niakwa. In the case of the Development Tools, Licensee shall not merge and/or install the Niakwa Software individually or collectively unless a Niakwa Programming Language RunTime Package has first been properly licensed and installed. Licensee's rights are limited to use on equipment and operating system software combinations indicated on the RunTime Package of Development Software. For a current list of approved compatibles, contact Niakwa or your authorized Niakwa distributor. Licensee shall not use the Niakwa Software in derogation of any third party's proprietary rights in any software program or in violation of any agreement between Licensee and any third party in respect of any software program. The use by Licensee of the Niakwa Software resulting in the misuse of any third party's software shall constitute a breach of the agreement. Niakwa may, from time to time, publish information relating to the use of Niakwa Software in conjunction with other software. Niakwa assumes such published information will be used by Licensee only if Licensee first determines it may do so without violating any third party's software rights. Licensee is solely responsible for all claims of third parties arising out of or related to Licensee's activities under this agreement or relating to the Niakwa Software (whether those claims are based on contract, tort, or otherwise) except as to Niakwa negligence. Licensee agrees to indemnify and hold Niakwa harmless from any and all such claims, and to pay all expenses including attorneys' fees, that Niakwa may incur as a result of such claims, including the expense of defending against them.

3. Proprietary Rights of Niakwa:
Full ownership rights to each item of the Niakwa Software rests with Niakwa, and Licensee shall have no rights to the Niakwa Software, or any changes made therein by Niakwa. Licensee shall not copy the physical diskettes unless otherwise authorized by Niakwa in writing. Licensee warrants that it will not inspect, disassemble, reverse-engineer or in any way attempt to determine the internal methods of the Niakwa Software. Licensee and its employees shall not disclose or transfer any portion of the Niakwa Software except as specifically authorized by Niakwa, whether in physical, magnetic, or any other form, to any person or organization. In the event Licensee attempts to use, copy, disclose, or transfer any portion of the Niakwa Software or any modification thereof in a manner contrary to the terms of this agreement or in derogation of Niakwa's rights, whether those rights are explicitly stated, determined by law, or otherwise, the Licensee shall be in breach of this agreement and Niakwa shall have the right, in addition to any other remedies available to it, to injunctive relief, either restrictively enjoining such acts and/or mandatorily requiring certain acts, it being acknowledged that other remedies are inadequate.

4. Term and Termination:
This License is effective from the date of receipt by Licensee and shall remain in force unless terminated by Niakwa upon a breach by Licensee. In the event of termination of this License, Licensee shall return to Niakwa within 20 days all existing copies of the Niakwa Software in Licensee's possession unencumbered and certify to Niakwa that all copies or partial copies of the Niakwa Software have been returned. No refund shall be made in respect to any returned Niakwa Software.

5. Limited Warranty and Limitation of Liability:

What Is Covered:
Niakwa warrants that the magnetic media containing the Niakwa Software and the Documentation are free from defects in materials and workmanship under normal use. Niakwa warrants that the Niakwa Software itself will perform substantially in accordance with the specifications set forth in the Reference Guide, the Development Package and the Installation Guide.

For How Long:
The above warranties are made for a period of (90) days from the date of original delivery to you.

What We Will Do:
Niakwa will replace any magnetic diskette, or Documentation which proves defective in materials or workmanship without charge, provided Licensee is not in breach of this agreement. Niakwa may either replace or repair any Niakwa Software that does not perform substantially in accordance with the specifications set forth in the Reference Guide, Development Package, or User's Guide with a corrected copy of the Niakwa Software or corrective code. Niakwa may also provide addenda or substitute pages in the case of an error in the Reference Guide, Development Package, or User's Guide.

If Niakwa is unable to replace a defective diskette or if Niakwa elects not to provide corrected Niakwa Software, Niakwa will refund the fees paid for the Niakwa Software.

What We Will Not Do:
Niakwa does not warrant that the functions contained in the Niakwa Software will meet your requirements or that the operation of the Niakwa Software will be uninterrupted or error-free. The warranty does not cover any Niakwa Software which has been subjected to damage or abuse, or which has been altered or changed in any way by anyone other than Niakwa. Niakwa is not responsible for problems caused by computer hardware or non-Niakwa software.

ANY IMPLIED WARRANTIES COVERING THE NIAKWA SOFTWARE INCLUDING ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY TO YOU. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you.

NIAKWA SHALL NOT IN ANY CASE BE LIABLE FOR SPECIAL, INCIDENTAL, CONSEQUENTIAL, INDIRECT OR OTHER SIMILAR DAMAGES ARISING FROM BREACH OF CONTRACT, NEGLIGENCE OR ANY OTHER LEGAL THEORY EVEN IF NIAKWA OR OUR AGENT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. This means we are not responsible for any costs incurred as a result of lost profits or revenue, loss of use of the software, loss of data, costs of re-creating lost data, the cost of any substitute program, claims by any party other than you, or for the above limitation or exclusion may not apply to you.

What You Must Do:
(1) To qualify for warranty coverage, Licensee must complete and return to Niakwa the USER REGISTRATION CARD contained herein.

(2) You must return the defective item to Niakwa or its authorized distributor postpaid within ninety (90) days of your original delivery, and it must be received within one hundred-five (105) days of delivery. You assume the risk of loss or damage in transit. Any claim under the above warranty must include a dated proof of purchase such as a copy of your receipt or invoice.

Other Conditions:
This warranty allocates risks of product failure between YOU and Niakwa. Niakwa's software pricing reflects this allocation of risk and the limitations of liability contained in this warranty. The warranty set forth above is in lieu of all other express warranties, whether oral or written. The agents, employees, , and dealers of Niakwa are not authorized to make modifications to this warranty, or additional warranties binding on Niakwa. Accordingly, additional statements such as dealer advertising or presentations, whether oral or written, do not constitute warranties by Niakwa and should not be relied upon.

State Law Rights:
This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

6. Diskette Replacement Policy:
After expiration of the ninety (90) day warranty period, and for two (2) years from the date of delivery, the original magnetic media may be returned to Niakwa or its authorized distributor for replacement if it becomes damaged. The damaged magnetic media must be returned with a service replacement fee.

In the case of RunTime Software, upon return of the magnetic media that has failed to operate and service replacement fee, if applicable, a substitute RunTime Package will be delivered.

Niakwa shall have no obligation to provide replacement Niakwa Software for releases other than the most current or to any Licensee who has failed to return the enclosed USER REGISTRATION CARD to Niakwa.

7. Attorneys' Fees:
Licensee shall be responsible to pay all expenses, including attorneys' fees, that Niakwa may incur as a result of enforcing its rights against Licensee under this agreement.

8. General:
This agreement may be altered, amended, or modified only by a written agreement executed by an authorized officer of Niakwa. This agreement shall be governed by and interpreted pursuant to the laws of the State of Illinois, United States of America. Licensee hereby consents to the jurisdiction of the courts of the State of Illinois and the United States District Court for the Northern District of Illinois.

PREFACE

P.1 The NPL Documentation -- An Overview

The Niakwa Programming Language (NPL) provides a portable development environment, allowing developers to distribute their applications on an array of hardware and operating environments. Because the underlying NPL source code is compatible with all NPL supported operating environments, developers can use a single source code approach in their application development, while maintaining maximum portability of an application.

Due to this source code portability, Niakwa has structured the NPL documentation into generic and operating environment dependent manuals to allow developers easy reference to detailed platform specific information. Each of these manuals is described below.

P.1.1 The NPL Technical Reference Guide

The NPL Technical Reference Guide consists of two manuals, the NPL Programmer's Guide (this manual) and the NPL Statements Guide. The Technical Reference Guide (TRG) is intended as a generic guide for programmers in the correct use of the Niakwa Programming Language, and its program development and debugging facilities on all supported environments.

This Programmer's Guide provides a detailed discussion on the operation of all areas of NPL. This guide also provides a detailed discussion of the Niakwa Compiler (B2C), the Niakwa Utilities, and a series of structured NPL examples.

The Statements Guide provides syntactical descriptions, discussion, examples, compatibility issues and references for all NPL statements. The Statements Guide also provides developers with a series of available NPL library functions. Additionally, the Statements Guide provides a Language Compatibility chart documenting all NPL statements.

P.1.2 Operating System-Specific Supplements

The NPL Operating System Specific Supplements are used in conjunction with the NPL TRG, to clearly document operating system dependent features of NPL, from one operating system to the next. Whenever an operating system dependent feature is discussed within the context of the TRG, the developer is advised to refer to the appropriate Operating System Specific Supplement for details.

Each NPL Supplement contains a general manual that discusses the operating system features of a specific operating system, along with a series of operating environment-specific addenda (as required). These addenda document NPL operations and options that are either unique to a given environment, or function differently from platform to platform.

Additionally, each supplement contains a set of Release Notes, providing information pertaining to an operating system specific-supplement that was not available at press time. Release Notes are dated and denote the NPL Revision to which these apply. As interim revisions of NPL are released to the field, new Release Notes are published to document changes and updates.

P.1.3 Installation Guides

An NPL Installation Guide is distributed with each NPL RunTime Package. The Installation Guide is operating system specific, and is intended to assist end users in the correct installation and operation of the NPL RunTime.

P.1.4 Upgrade Guides

Upgrade Guides are intended to guide an end user through the correct process of applying an NPL Upgrade RunTime to a currently installed NPL RunTime (where available).

P.1.5 Supplementary Documentation/Services

As with all programming languages, the potential for bugs always exist. To minimize these, the staff at Niakwa has taken due care to perform stringent quality assurance testing of the Niakwa Programming Language.

All bugs that are discovered will be published first on the Niakwa Bulletin Board System (NBBS). New bugs are posted on a monthly basis to the currently open bug report, which is closed and distributed to the field on a quarterly basis.

Although most bugs are minor and are easily worked around, occasionally, a more severe bug is discovered and requires an immediate fix. In this event, Niakwa will distribute notification and make available documentation and software to all NPL developers running on the current NPL revision and platform for which the patch is intended. Distribution of these patches is typically performed using Niakwa's BBS (or by request).

As a continuing service to our developers, Niakwa will periodically publish and distribute Tech Notes to all NPL developers. Niakwa Tech Notes cover a broad spectrum of technical issues that may have an impact on Niakwa developers. Like patches, all Tech Notes are maintained and available on the NBBS.

P.2 How to Use the Programmer's Guide

The following sections provide an overview of the NPL Programmer's Guide and recommended guidelines on how to proceed.

P.2.1 Documentation Conventions

This guide uses the following notational conventions:

NOTE: Notes provide information of particular importance.

WARNING--Warnings are special conditions that require extra care by the user.

HINT: Hints provide helpful comments pertaining to the use of particular features.

The followingg conventions are used in the illustration and defination of NPL:

- Each statement appears on a separate page, with the statement as a page header.
- The general form of each statement is enclosed within a box.
- Uppercase letters ("A" through "Z"), digits ("0" to "9"), and special characters (such as "$", "#", ":") must always appear exactly as presented in the general format.
- All lower-case words indicate information that the user must supply. These words appear in *italic* type.

For example:

```
LEN (alpha-variable)
```

The user must supply the alpha-variable.

- When braces, "{ }", enclose a vertically stacked list, or a horizontal list with each item separated by a comma (","), the user must choose one of the options within braces. Information within braces is shown in *italic* type.

  For example:

  ```
  ALL ({literal-string, alpha-variable, two-hexdigits})
  ```

  or

  ```
  ALL   ({literal-string  })
         {alpha-variable  }
         {two-hexdigits   }
  ```

Here, the ALL instruction must be followed by one and only one of the items in the list.

- Brackets, "[ ]", indicate that the enclosed items are optional. When brackets enclose a vertical list or a horizontal list, the user may specify one or none of the items. Information within brackets is shown in *italic* type.

  For example:

  ```
  INPUT [literal-string,] variable [,variable]...
  ```

  Here, the INPUT instruction may optionally contain a literal-string followed by an optional comma preceding the required "variable". Additional variables may optionally be specified.

  or:

  ```
  CLEAR [V                               ]
        [N                               ]
        [P [line-number1][,[line-number2]]]
  ```

  Here, either the V, N, or P parameter may be specified, but no parameter is required.

NOTE: Here, line-number parameters may be optionally specified only if the "P" parameter is specified.

- The presence of an ellipsis (...) within any format indicates that the unit immediately preceding the ellipsis can occur one or more times in succession.

  For example:

  ```
  DEFFN'integer[(variable[,variable]...)]
  ```

  Here, any number of "variable" may be specified, but the format ",variable" must be used for the second and subsequent "variables".

- All other punctuation such as commas or parentheses must be included where shown.

P.2.2 Programmer's Guide Overview

This section provides a brief description of each chapter and appendix contained within this guide.

Chapter 1 provides an overview of the NPL Development Environment and its structure.

Chapter 2 takes an in-depth look at working within the NPL environment using the interpretive (RTI) and non-interpretive (RTP) RunTime programs.

Chapter 3 discusses RunTime memory considerations.

Chapter 4 takes a detailed look at structured programming and describes the recommended NPL program constructs to use in developing structured applications.

Chapter 5 discusses working with the interpretive (RTI) RunTime to create, save and edit NPL programs.

Chapter 6 discusses the debugging features of NPL.

Chapter 7 provides a detailed discussion of addressing hardware devices under NPL.

Chapter 8 identifies and discusses NPL error handling techniques.

Chapter 9 describes a series of NPL system variables initialized by the RunTime at startup.

Chapter 10 discusses usage of the NPL $SHELL command.

Chapter 11 discusses operating and developing context sensitive help with the NPL HELP processor.

Chapter 12 provides details on developing self running demonstration programs using the NPL $DEMO statement.

Chapter 13 discusses the Niakwa Utilities programs provides details on the use of each utility.

Chapter 14 discusses the how to use the Niakwa compiler program (B2C) and its options.

Chapter 15 discusses various methods of porting NPL programs and data from one NPL supported environment to another.

Chapter 16 describes the process of integrating external routines written in other supported languages with NPL.

Chapter 17 provides a series of example NPL programs.

Appendix A discusses the immediate benefits of upgrading code from previous releases of NPL.

Appendix B lists all NPL error messages.

Appendix C lists all NPL compiler errors.

Appendix D discusses the terminal characteristics of all NPL supported terminals.

Appendix E describes NPL Keyboard Keywords.

Appendix F discusses NPL de-compiler compatibility.

Appendix G provides an NPL "RAW" device compatibility chart.

P.2.3 Getting Started with NPL

This section is intended as a guide to assist programmers in deciding where to begin within the Programmer's Guide, based upon the programmers current knowledge of NPL.

New to NPL

If you are new to the Niakwa Programming Language and have had experience with other languages, the following chapters will get you acquainted with NPL and get you up to speed in minimal time.

- Chapters 1 and 2 provide a high level overview of the NPL Development Environment and a complete introduction to the NPL RunTime programs (RTI and RTP).

- Chapter 4 introduces many of the structured constructs implemented in NPL Release IV. "C" programmers in particular, should feel quite comfortable with these constructs. Heavy emphasis is placed on structured programming techniques and how they are applied using NPL. A review of the example programs documented in Chapter 17 of this manual and included with the NPL Development Package software will compliment this chapter.

- Chapters 5 and 6 introduce the NPL Editor and NPL Debug facilities, respectively.

- Chapter 7 expands on the virtual machine environment concept discussed in Chapter 1 and provides detailed information on how NPL handles each component of this virtual machine environment.

- Chapter 9 identifies a series of internal NPL system variables initialized at startup. These variables provide useful information to the programmer and, in most cases, may be modified.

- Chapter 13 documents the Niakwa Utilities. These utilities can serve programmers as useful tools in developing and maintaining applications.

Experienced BASIC-2 User - New to NPL

If you are a BASIC-2 user and are new to the Niakwa Programming Language , the following chapters will get you introduced to NPL and how it emulates the Wang 2200 environment.

- Chapters 1 and 2 provide a high level overview of the NPL Development Environment and a complete introduction to the NPL RunTime programs (RTI and RTP).
- Chapter 7 expands on the virtual machine environment concept discussed in Chapter 1 and provides detailed information on how NPL handles each component of this virtual machine environment. This chapter will clarify how the NPL emulates the Wang 2200 environment.
- Chapters 5 and 6 introduce the NPL Editor and NPL Debug facilities, respectively. Many enhancements have been implemented in the NPL Editor that do not exist in BASIC-2.
- Chapters 14 and 15 discuss the Niakwa Compiler and porting procedures for converting and moving your applications from a Wang 2200 to a Niakwa-supported platform.
- Chapter 9 identifies a series of internal NPL system variables initialized at startup. These variables provide useful information to the programmer and, in most cases, may be modified.
- Chapter 4 introduces many of the structured constructs implemented in NPL Release IV. Heavy emphasis is placed on structured programming techniques and how they are applied using NPL. A review of the example programs documented in Chapter 17 of this manual and included with the NPL Development Package software will compliment this chapter.
- Chapter 13 documents the Niakwa Utilities. These utilities can serve programmers as useful tools in developing and maintaining applications.

Experienced NPL Users

If you are an experienced NPL user, welcome to Release IV. The following chapters should help you get acquainted quickly to the new documentation layout and highlight the new areas.

- Review the Preface and Table of Contents to note the NPL documentation layout changes and any new sections that may be of interest.
- Appendix A provides a discussion on upgrading code that has been written using previous releases of NPL.
- Chapter 4 introduces many of the structured constructs implemented in NPL Release IV. Many enhancements and changes have been introduced in Release IV, and this chapter represents the core of these changes. Programmers are encouraged to become comfortable with the constructs introduced in this chapter. Heavy emphasis is placed on structured programming techniques and how they are applied using NPL. A review of the example programs documented in Chapter 17 of this manual and included with the NPL Development Package software will compliment this chapter.
- Chapters 5 and 6 discuss the NPL Editor and NPL Debug facilities, respectively. Many enhancements have been implemented in the NPL Editor with Release IV.

P.3 Getting Help

This section is intended to assist developers' in getting the best and most expedient support from Niakwa if a problem is encountered.

P.3.1 Warranty Information

Refer to the Niakwa Software Licence Agreement and Warranty at the start of this manual.

P.3.2 Patch - Update - Upgrade Policies

Niakwa is continually working on the development of NPL. Public releases of new versions of our products typically fall into one of three categories. These are:

Patches. Patches typically consist of a series of bug corrections for an existing product and are produced as needed. Although the version number has been updated with the patch release (i.e., 4.00.08 to 4.00.15), only minor changes to the product will have been made. Distribution of patches is done over Niakwa's BBS (or by request) and are free of charge. Notification of availability provided by Niakwa's Tech Notes.

Updates. Updates generally provide several significant enhancements to the existing Niakwa product and are provided within the life of a current product release (i.e., Release III). With updates, the revision number of the product is updated with the new update release (i.e., 3.00 to 3.20). Distribution of updates is provided only by ordering and paying the appropriate fee. Notification of availability is provided by Niakwa's Marketing Bulletins.

Upgrades. Upgrades provide major changes to an existing Niakwa product and only occur every couple of years. With upgrades, the release number of the product is updated with the new upgrade release (i.e., 3. to 4.). Distribution is provided only by ordering and paying the appropriate fee. Notification of availability is provided by Niakwa's Marketing Bulletins.

NOTE: Developers must be currently licensed with Niakwa to receive any of the above.

P.3.3 Product Support

At Niakwa, we are dedicated to keeping our customers satisfied--not only with quality products, but with outstanding support as well.

To better serve you when problems arise, Niakwa has made a substantial investment in technical support staff and their training. The level of support and speed of response we can provide is directly dependent on the information provided to us.

Customers can obtain the best product support by following the steps shown below and using the enclosed form **prior** to calling Niakwa's technical support.

1. Refer to the NPL manuals, Release Notes, Bug Reports, and Tech Notes to see if the problem is covered there.

2. Run the program in question without other programs in memory to see if the problem continues.

3. Use the NPL Problems/Comments/Suggestions Form provided below to write down the required information on the problem system.

4. Have available all configuration details: CONFIG.SYS and AUTOEXEC.BAT for MS-DOS; profile, login, PATH and BASIC2C_TERM for XENIX/UNIX; CONFIG.P for SuperDOS; etc.

5. Be ready to describe the problem clearly and in detail (make notes if necessary).

6. Call from a phone near the computer; the Support Analyst may have questions about the system.

7. Make sure the software in question is running on the computer, and have related manuals nearby.

8. Please have patience. Other customers may have questions, and Niakwa wants to help everyone.

Our support Analysts will answer questions about NPL and its functions, operation, and installation, and can guide developers through difficult procedures. We can also advise customers about the appropriate solution to any limitation known to exist in earlier versions of our software. We will also attempt to answer your questions on the setup of NPL on the specific hardware you are working with.

NOTE: Upgrading or ordering questions should be directed to the Niakwa NPL sales staff. Sales should also be contacted with suggestions for future upgrades of NPL and additional system ports to be considered.

It is your responsibility to have a technically proficient person available for each NPL platform you are supporting. We do not provide help for loading operating systems, operating system level problems, or hardware problems. Please contact the vendor's technical support departments directly for problems of this nature. For instance, if a printer does not work from the operating system level, it will not work with NPL.

Technical support for NPL is available at (708) 634-8700, from 8:00 a.m. to 5:00 p.m. (Central Standard Time). You may also FAX your problems to us at (708) 634-8718.

To further open the lines of communication, the Niakwa Bulletin Board System (NBBS) is online 24 hours. The NBBS is an open forum for all NPL developers to exchange ideas and information. Niakwa also maintains updated bug reports and current patch information on this system. Login as "GUEST" or, to obtain a personal LOGIN to the NBBS, contact Niakwa Technical Support.

NOTE: **We are happy to extend our services to all /resellers so they can better help their end-users. Please do not tell an end-user to call us. We are not equipped to handle all of our distributors'/resellers' end-users directly.**

Support Analysts are always ready to explain a situation, provide a solution, or offer suggestions on NPL related topics. Niakwa aims to provide NPL customers with the most comprehensive technical support available. If we don't know the answer to an NPL related issue, we will find out!

P.3.4 Reporting Problems

If you are encountering problems or simply have a comment or suggestion, simply fill out the form on the next page and FAX it to Niakwa at (708) 634-8718. Remember, your thoughts count!!!

NPL Problems/Comments/Suggestions Form

Date:_______________________________ Phone Number:______________________

Company:___________________________ FAX Number:______________________

Reported By:_________________________

Type: __Problem __Comment __Suggestion __Other:_______________________

Product____________ Documentation____________ Software___________

Operating System: __MS-DOS __Novell __SuperDOS __XENIX __UNIX __VMS __68000 UNIX
__Other:__________________________

Operating System Release/Revision: __________________________

RunTime Revision (Please provide all 7 digits): _________________________________
(use $REV in 3.00 or greater or start the RunTime and look at the security screen)

Hardware: ________________________________(Please provide any important details about your hardware)

Detailed Description:____

____

____

____

____

____
(If needed, please attach more pages) Attached pages__________

Sample program attached ? _____
(If it applies, please provide a small sample program that replicates easily the reported problem)

If applicable, please answer the following questions:
Any recent changes in the operating system ? ______

Did the program/security work before ? __________

Logged on as "root" (XENIX/UNIX systems) or "SUPERVISOR" (on NOVELL) or "user"? __________

Problem occurs only on console or only on terminals or both ? _______

Anything else specific to your system? _______________________________________

Niakwa Support Representative: ____

TABLE OF CONTENTS

P-1

PREFACE P-1

The NPL Documentation -- An Overview P-1
- The NPL Technical Reference Guide P-2
- Operating System-Specific Supplements P-2
- Installation Guides P-3
- Upgrade Guides P-3
- Supplementary Documentation/Services P-3

How to Use the Programmer's Guide P-3
- Documentation Conventions P-4
- Programmer's Guide Overview P-6
- Getting Started with NPL P-8
- **New to NPL** P-8
- **Experienced BASIC-2 User - New to NPL** P-9
- **Experienced NPL Users** P-10

Getting Help P-10
- Warranty Information P-10
- Patch - Update - Upgrade Policies P-10
- Product Support P-11
- Reporting Problems P-13

INTRODUCTION 1-1

Overview 1-1

The Niakwa Development Environment 1-2
- The Niakwa Programming Language (NPL) 1-2
- **Creating NPL Programs** 1-4
- NPL Add-Ons 1-6
- The Niakwa Development Tools 1-6
- **Niakwa Data Manager** 1-6
- **Intelligent Query** 1-7
- Niakwa's Objectives 1-7

NPL Technical Features 1-8
Performance 1-8
Interactive Environment 1-8
Debugging Facilities 1-9
Portability 1-11
Multi-User 1-11
Program Constructs 1-11
Program Modules (Libraries) 1-11
FUNCTION Interface 1-12
Mixed Language Programming 1-12
Data Conversion Statements 1-12
Advanced Math Functions 1-12
String Handling 1-13
Advanced Matrix Manipulation 1-13
Device Independent 1-13
Native Operating System Calls 1-13
Data Verification and Error Handling 1-13
HELP Subsystem 1-13
$DEMO 1-14
Program Encryption 1-14
Wang 2200/Basic-2 Upward Compatibility 1-14
NPL Limits 1-14
The NPL Virtual Machine Environment 1-17
Input/Output 1-18
Storage Devices 1-18
Native Operating System Files 1-19
ISAM Files 1-19
Display Devices 1-20
NPL Character Sets 1-20
NPL Virtual Keyboard 1-23

RUNTIME OPERATIONS 2-1

Overview 2-1
RTP Versus RTI 2-2
Enabling RTI 2-2
Starting the NPL RunTime 2-3
Starting from the Command Processor 2-3
Starting from Batch/Script Files 2-4
Starting from a Menu 2-4

Command Line (Start-up) Options 2-5
/B (Background Partition) 2-5
/D (DET Entries) 2-6
/G (Graphics Mode) 2-7
/H (Handle Table Size) 2-8
/K (Mouse Support) 2-10
/L (Leave Overhead Memory) 2-11
/M (XMS Memory) 2-11
/P (Pre-boot) 2-11
/R (Remote Control) 2-13
/S (Separate Program Segments) 2-13
/T (Terminal/Port Number) 2-14
/U (UMB Memory) 2-14
/X (External Library) 2-15
Boot Program 2-16
Immediate Mode Operation 2-18
Two Types of Immediate Mode Entries 2-18
Immediate Mode Program Lines (Deferred Execution) 2-18
Immediate Mode Commands (Immediate Execution) 2-19
Invoking Immediate Mode 2-20
Invoking Immediate Mode from the Keyboard 2-20
Invoking Immediate Mode from Program Control 2-21
Conditionally Invoking Immediate Mode 2-22
Invoking Immediate Mode on ERROR Conditions 2-23
Summary Table 2-24
Immediate Mode Operation 2-24
Statements Which Are Illegal As Commands 2-25
Statements Which Operate Differently As Commands 2-25
1. **Transfer of Control Statements** 2-26
2. **Program LOAD Statements** 2-26
3. **PRINT Statements** 2-27
4. **$SHELL Statement** 2-27
Preserving a Program's "Execution Status" 2-27
Exiting Immediate Mode 2-28
CONTINUE Command 2-28
EXECUTE Key 2-29
RUN Command 2-29
$END Command 2-29
Exiting NPL 2-30
Exiting under Program Control 2-30

Exiting Using the HELP Key 2-31
Exiting Using the Interrupt Key 2-31
The HELP Processor........ 2-31
Selecting HELP Options 2-32
HELP Options........ 2-33
Disabling Immediate Mode Options 2-35
Print Control 2-36
Accessing Printer Control 2-36
The Print Control Screen 2-36
Printer Control Options 2-37
Print Control Values 2-39
Redirect Feature for Remote Support........ 2-40
Establishing the Communications Link 2-40
Turning Control Over to the Remote Terminal 2-41
Example of Redirect Feature........ 2-44
Troubleshooting from Remote Sites 2-45
Relinquishing Control Back to the End-User Site 2-46
RunTime Errors 2-47
Error Handling Under RTP 2-47
Error Handling Under RTI 2-50
Device Not Ready Conditions........ 2-50
Error Descriptions 2-52

RUNTIME MEMORY USE 3-1

Overview........ 3-1
Dynamic Partition Size........ 3-2
Space Functions 3-3
P-Code Size Relative to Wang 2200........ 3-4

NPL PROGRAMMING CONSTRUCTS 4-1

Overview........ 4-1
NPL Logical Program Organization........ 4-2
Structured Programming 4-3
Code Modularization........ 4-4
Program Lines........ 4-5
Statement separators........ 4-6
Order of Execution 4-6
Statement Labels 4-7

Transfer of Control 4-7
Data Types 4-9
String 4-9
Numeric 4-9
Literals and Constants 4-9
Literals 4-10
Numeric Constants 4-10
Constant Variables 4-10
Variables 4-11
Naming Conventions 4-11
Significance of Spaces in Syntax 4-11
Identifier Abbreviations 4-12
Case Conventions 4-12
Short Identifier Names 4-13
Long Identifier Names (LINs) 4-13
Variable Types 4-14
String Variables 4-14
Numeric Variables 4-15
Array 4-16
Pointer 4-17
Constant Variables 4-17
Field Identifiers and Logical Records 4-17
Allocation Classes of Variables 4-18
Static Allocation 4-18
Recursive Allocation 4-18
Scope of Variables 4-19
Function Private Variables 4-20
Module Private Variables 4-20
Public Variables 4-20
Variable Declaration 4-21
Undeclared Variables 4-22
Variables Larger than 64KB 4-22
Restrictions on Use 4-23
Expanding Array Size 4-24
Numeric and String Expressions 4-25
Numeric Operators 4-25
Numeric Expressions 4-26
Matrix Support 4-27
String Operators 4-27
String Expressions 4-28

Data Conversion 4-29
FUNCTIONs and PROCEDUREs 4-29
Parameter Types of Functions...4-30
Parameter Passing Conventions 4-30
Parameter Specifying Modifiable Arguments 4-31
Passing Large Arguments by Reference 4-31
Scope of Functions 4-31
Module Private Functions 4-32
Public Functions 4-32
Scope of Variables Declared within Functions..4-33
Coding Restrictions 4-33
Location of the Body of a Function 4-33
Branching Restrictions in a Function...4-33
Nesting of Function Bodies...4-34
Function Body Skip-over...4-34
DEFFN' Declarations..4-34
Calling a Defined Function 4-34
Argument Type Checking 4-34
Use of Constant Parameters 4-34
Use of /POINTER Parameters..4-34
Restrictions on Arguments for POINTER Parameters..........................4-35
Resolution of Function Identifier 4-35
Indirect Specification of Function Name 4-35
Modules 4-36
Module Concepts...4-36
Module Categories 4-37
Root Module 4-37
Run Module...4-37
Executing Module 4-37
List Module 4-38
Choosing Module Names 4-38
Reviewing Loaded Modules...4-39
Modifying an Active Module 4-39
Nested Included Modules 4-40
Module Initialization and Cleanup..4-41
Discarding Modules 4-41
Effect on Immediate Mode Commands 4-42
Execution Time LOAD and SAVE...4-43
Guidelines for Writing Modules 4-43
Logical Constructs 4-44

Structured IF...ELSE...ENDIF 4-44
WHILE...WEND 4-45
REPEAT...UNTIL 4-46
FOR/BEGIN...NEXT 4-46
SWITCH...CASE 4-47
RECORDs/FIELDs 4-48
Field Identifiers 4-49
Field Expressions 4-50
Indirect Field References 4-52
Example of RECORDs/FIELDs 4-53

CREATING NPL PROGRAMS AND THE NPL EDITOR 5-1

Overview 5-1
Creating NPL Programs 5-2
NPL P-Code 5-2
The NPL Interpreter (RTI) 5-2
The NPL Compiler (B2C) 5-3
$SOURCE/$OBJECT 5-4
Loading Programs 5-4
Program Text Merging 5-5
Loading from Diskimages 5-5
Loading from Native Operating System Files (BOOT and PREBOOT) 5-6
Loading Program Text at RunTime 5-7
Loading Modules 5-8
The Line Editor 5-8
Multi-Command Buffering 5-11
EDIT Functions 5-11
Standard Keys 5-12
Special Function Keys 5-12
Special Function Keys - DEFFN Mode 5-13
Special Function Keys - EDIT Mode 5-14
Alternative to EDIT Mode SF Keys 5-16
Multi-Character Insertion 5-18
Overstrike Mode 5-18
Insert Mode 5-19
Multi-Statement Program Lines 5-21
Suppression of Statement Separators 5-21
Statement Insertion and Deletion 5-22

Adding Statements....5-23
Inserting a Statement....5-23
Deleting a Statement....5-24
Forcing Return-Graphics....5-24
Multi-Screen Text Lines....5-25
Use of TAB/SHIFT-TAB....5-26
Editing Programs....5-27
Selecting a Module to Edit....5-27
Entering a New Program Line....5-28
Replacing a Program Line....5-28
Changing a Program Line with RECALL....5-28
Deleting Program Lines....5-29
Concatenating Program Lines....5-30
Moving Program Lines....5-30
Recalling Long Identifier Names....5-31
Syntax Errors....5-32
Other Useful Editing Tools....5-33
Saving Programs....5-34
Saving New Programs to Diskimages....5-35
Saving Existing Programs to Diskimages....5-36
Insufficient Space in Old Program....5-36
Use of DEFFN'....5-36
Saving Portions of Programs to Diskimages....5-37
Saving Programs to Native Operating System Files....5-38
Scramble Protecting Programs....5-39

DEBUGGING CODE 6-1

Overview....6-1
Prerequisites....6-2
Debugging Modules....6-2
Inspection and Modification of Program Logic....6-4
Logic Inspection Using Stepped Execution....6-4
Step Mode....6-5
Step Range....6-5
Activating Stepped Execution....6-5
Operating Stepped Execution....6-7
Temporary Exit from Stepped Execution....6-9
CONTINUE RETURN....6-9
CONTINUE NEXT....6-9

CONTINUE LOAD 6-10
DEFFN' 6-10
Deactivating Stepped Execution 6-10
Logic Inspection Using TRACE 6-11
Activating TRACE Mode........ 6-11
TRACE Output........ 6-12
TRACE Output on Transfer of Control 6-12
Trace Output on Variable Modification........ 6-13
Selective TRACE 6-15
Redirecting TRACE Output........ 6-17
Logic Inspection Using LIST 6-17
LIST STACK........ 6-17
LIST DT 6-18
Logic Modification........ 6-18
GOTO 6-19
RETURN 6-19
RETURN CLEAR........ 6-19
RETURN (value)........ 6-20
RETURN ERROR (code) 6-20
NEXT 6-20
RUN # 6-21
RUN 6-21
Inspection/Modification of Program Variables........ 6-21
Variable Inspection........ 6-21
Automatic Variable Inspection with TRACE........ 6-23
TRACE V 6-24
Variable Modification 6-25
Inspection of Program Text 6-26
Inspection Using the LIST Commands........ 6-26
Redirecting LIST Output........ 6-27
Inspection of the Program Environment 6-28
Commands........ 6-28
System Variables 6-29
Public Functions, Procedures and Variables........ 6-30

DEVICE SUPPORT 7-1

Overview........ 7-1
Device Emulation 7-2
Device Addressing........ 7-3

The Device Equivalency Table (DET)....7-3
The $DEVICE Statement 7-4
Default Device Equivalence Table....7-6
Maximum Entries Allowed in the Device Equivalence Table 7-7
Inspecting the Device Equivalence Table....7-7
Modifying the DET....7-7
Clearing Entries in the Device Equivalence Table 7-8
The Internal Device Table (DT)....7-8
Default Devices....7-9
File Number Slots 7-10
Modifying the Internal Device Table....7-10
Use of the SELECT Statement....7-10
Other Functions and Statements 7-11
Examining the DT....7-11
Module Status 7-12
Wang 2200 Compatibility Note....7-12
Exclusive Access 7-12
The $OPEN Statement....7-13
Implied Exclusive Access 7-13
Storage Devices 7-14
NPL Diskimages....7-15
General Features of Diskimages 7-15
Sector Size 7-15
Maximum Size 7-16
Maximum Number of Diskimages....7-16
File Storage....7-16
Compatibility 7-16
Compatibility with Native Operating System File Structures....7-17
Diskimage Addressing 7-17
Native Operating System Files as Diskimage Files....7-18
Creating Diskimage Files....7-18
Changing the Size of a Diskimage File 7-19
Deleting Diskimage Files....7-20
Wang 2200 Compatibility Notes 7-20
Native Operating System "RAW" Devices as Diskimages 7-21
Purpose of Using "Raw" Devices....7-21
Characteristics of "Raw" Devices 7-21
Internal Structure of Diskimage Files 7-22
Catalog Index and Catalog Area 7-22
Listing the Index....7-25

Non-Cataloged Area 7-25
Cataloged Files 7-26
Wang 2200 Compatibility Notes 7-27
Creating Files 7-28
Maintaining Files 7-28
Accessing Files 7-29
Internal Format of Program Files 7-29
Internal Format of Data Files 7-29
Catalog Access Methods 7-30
Summary of Statements Used for Catalog Access to Data Files 7-30
The Logical Record 7-32
Diskimage Utilization for Logical Records 7-32
Logical End-of-File 7-33
Use of the Internal Device Table 7-34
Direct Access 7-35
Summary of Statements Used for Direct Access to Data Files 7-35
Extended Diskimages 7-37
Implementation 7-37
Maximum Size 7-38
Compatibility 7-38
Implications for Application Software 7-38
Three-Byte Addresses 7-39
Direct Access to the Index 7-39
Use of Bytes 7 & 8 for Other Purposes 7-39
Internal Structure of Diskimages 7-40
NPL Statements Affected 7-41
For Non-Extended Diskimages: 7-43
Native Operating System Files 7-44
ASCII File Access 7-44
Print Class Devices 7-44
Disk Class Devices 7-45
The Niakwa Data Manager (NDM) 7-47
Screen Handling 7-47
General Features 7-47
Addressing the Screen 7-48
The NPL Screen Character Set 7-49
Screen Output Character Translation 7-51
Internal Access 7-51
Screen Control Codes and Attributes 7-52
Video Attributes 7-53

Graphics 7-55
Color Support 7-55
Dynamic Selection of Colors Using HEX Sequences 7-57
Underline Color Replacement 7-58
Selecting "BRIGHT" Attribute 7-59
Sample Color Program 7-59
Box Graphics 7-60
True Boxes 7-60
Character Boxes 7-60
Pixel Graphics 7-61
Plotting Capabilities 7-61
Statements Used to Print to the Screen 7-61
132-Column Support 7-62
Uses and Limitations of INPUT SCREEN/PRINT SCREEN 7-63
Keyboard Support 7-64
The Keyboard Device Address 7-64
Virtual Key Names and Codes 7-65
Use of Special Function Keys by NPL 7-66
The HALT Key 7-67
The Underline Key 7-67
Keyboard Translation 7-68
Statements Used for Keyboard Input 7-69
Extended Keyboard Entry 7-69
Mouse Support 7-71
Terminal/Monitor Support 7-72
Determination of Type 7-73
NPL Directly Supported Terminals/Monitors 7-73
Local Printer Support 7-75
Support of Native Operating System and Utilities 7-75
Printers 7-76
Valid NPL Print Addresses 7-76
Equivalent Native Operating System Devices or Files 7-76
Specifying the Output Printer Device 7-77
Available Print Statements 7-77
Control Sequences 7-78
Use of Native Operating System Files 7-78
Printer Translation 7-79
Local Printer 7-81
Special End-of-Line Handling 7-81
Serial Devices 7-82

Directing Output to Other Devices 7-82
Accepting Input 7-83
Input from the Serial Port 7-83
Telecommunications Support....................7-85

ERROR HANDLING 8-1

Overview....................8-1
Error Classes 8-2
RunTime Program Errors and Warnings 8-2
Recoverable Versus Non-Recoverable Errors....................8-3
Resolution Versus Execution Errors 8-5
Handling Under Program Control 8-7
ERROR DO 8-7
ERR 8-8
ERR$ 8-10
$OSERR 8-11
Error Help Files 8-12

SYSTEM VARIABLES 9-1

Overview....................9-1
$MACHINE....................9-2
$OPTIONS....................9-2
Other System Variables 9-3
$BOXTABLE....................9-3
$KEEPREMS 9-4
$KEYBOARD....................9-4
$NETID 9-4
$PRINTER....................9-5
$PROGRAM 9-5
$REV 9-5
$SCREEN....................9-5
$SER 9-6

$SHELL 10-1

Overview....................10-1
Uses of $SHELL....................10-2

General Versus Specific Program Use 10-3
General Form 10-3
Executing Specific Commands 10-5
Native Operating System File System 10-6
Return Code 10-6
Command Usage 10-6
From Immediate Mode 10-6
From The HELP Processor 10-7
From Program Control 10-7
Memory Requirements 10-8

THE HELP PROCESSOR 11-1

Overview 11-1
User-Definable Help Screens 11-2
The HELP ENTRY 11-2
Stand-alone HELP Files 11-3
Indexed HELP Files 11-3
The $HELP Statement 11-4
Special Sequences 11-5
Nested Help Entries 11-6
Indexed Help Files 11-7
Advantages and Disadvantages of Indexed HELP Files 11-7
Creating an Indexed HELP File 11-8
Defining HELP Entries Within an Indexed File 11-8
Creating the Index for the File 11-9
To Use an Indexed HELP File 11-9
Alternative Search 11-10
External References within Indexed HELP Files 11-11
Internal References within an Indexed File 11-11
General Notes on Help Files 11-12
Additional HELP Processor Features 11-13
Mouse Support 11-13
Keyboard Functionality 11-13

$DEMO 12-1

Overview 12-1
$DEMO Statement 12-2
Syntax 12-2

Effect of Execution 12-3
Determining $DEMO Status 12-4
DEMO Script File 12-4
Keystrokes 12-4
Keystroke Processing 12-5
Special Keys 12-6
Repeat Keys 12-7
Timing 12-7
Display of Informational Text 12-8
The BOX Statement 12-8
Special Sequences 12-9
End-of-Text Delimiter 12-10
Modifying Box Parameters 12-11
BOX Graphics Display 12-11
Redefining the Prompt 12-12
Adding Comments 12-12
Restrictions 12-13
The HELP Key 12-13
Polling KEYIN 12-13
$IF ON 12-13
Keyboard Logging 12-13

NPL UTILITIES 13-1

Overview 13-1
Starting the NPL Utilities 13-3
The NPL Utilities Menu 13-4
Menu Utility 13-4
Format Floppy Disks 13-4
General Backup Utility 13-4
General Recovery Utility 13-5
General File Copy Utility 13-5
Diskimage Listing 13-5
New Diskimage Creation 13-5
Change Diskimage Size 13-5
Scratch Single File 13-5
Change Utilities Device Equivalences Table 13-5
Keyboard Translation Tables Editor 13-5
Screen Translation Tables Editor 13-5
Font Table Editor 13-5

Printer Control Table Editor.......... 13-5
Indexed Help Table Editor.......... 13-5
Diagnostic Dump File Processor.......... 13-6
Printer Translation Table.......... 13-6
Edit Options.......... 13-6
Other Program.......... 13-6
Operation.......... 13-6
Menu Options Setup.......... 13-7
Accessing the Menu Options Program.......... 13-7
Error Handling.......... 13-8
Format Floppy Disks.......... 13-8
Features.......... 13-9
Operation.......... 13-9
General Backup Utility.......... 13-10
Features.......... 13-11
Operation.......... 13-12
Selecting Files.......... 13-13
The Copying Process.......... 13-13
Limitations.......... 13-14
General Recovery Utility.......... 13-15
Features.......... 13-15
Operation.......... 13-16
Error Handling.......... 13-17
Limitations.......... 13-17
General File Copy Utility.......... 13-18
Features.......... 13-18
Operation.......... 13-19
Selecting Files.......... 13-20
The Copying Process.......... 13-21
Exceptions13-21
Overwrite Exceptions.......... 13-22
Diskimage Listing.......... 13-22
Features.......... 13-23
Operation.......... 13-24
New Diskimage Creation.......... 13-24
Features.......... 13-25
Operation.......... 13-25
Change Diskimage Size.......... 13-26
Features.......... 13-26
Operation.......... 13-27

Scratch Single Files 13-28
Features 13-28
Operation 13-28
Change Utilities Device Equivalences Table 13-29
Features 13-29
Operation 13-30
Keyboard Translation Tables Editor 13-31
Features 13-31
Operation 13-32
Screen Translation Tables Editor 13-34
Features 13-35
Operation 13-35
Font Table Editor 13-37
Features 13-37
Operation 13-37
Printer Control Table Editor 13-40
Features 13-40
Operation 13-40
Indexed HELP File Processor 13-41
Features 13-41
Operation 13-41
Error Handling 13-42
Diagnostic Dump Variables List 13-43
Printer Translation Table 13-43
Features 13-44
Operation 13-44
Edit Options 13-45
Features 13-45
Operation 13-46
OTHER Program 13-47
Features 13-47
Operation 13-47
Generating 2200 Source 13-48

COMPILER OPERATION 14-1

Overview 14-1
The Compiler Command Line 14-3
Format 14-4
Discussion 14-6

Selecting Input Programs for Compilation 14-7
Literal Program Selection 14-7
Wildcard Program Selection 14-8
Compiler Options 14-9
Default Options 14-10
Changing Options During a Compiler Run 14-10
Specifying Path Names and Device Names 14-10
The SRCLOC Option 14-10
Format 14-11
Discussion 14-11
Diskimage Files 14-11
The OBJLOC Option 14-13
Format 14-13
Discussion 14-13
Diskimage Files 14-14
Native Disk Environment 14-15
No Object Programs Produced 14-15
Scrambled Program Considerations 14-15
The OBJFORMAT Option 14-16
Format 14-16
Discussion 14-16
Scrambled Program Considerations 14-16
The OBJEXTRA Option 14-17
Format 14-17
Discussion 14-17
The LSTLOC Option 14-17
Format 14-18
Discussion 14-18
Diskimage Files 14-18
Native Directory 14-19
Native Files 14-19
"RAW" Diskettes 14-19
Print Device 14-20
The LSTFORMAT Option 14-20
Format 14-20
Discussion 14-20
.LST Format 14-20
.SRC Format 14-21
2200 Format 14-21
2200 "Compressed" Format 14-21

The ERRLOC Option ... 14-21
Format ... 14-22
Discussion ... 14-22
Native Files ... 14-22
Print Device ... 14-22
The WARNINGS Option ... 14-23
Format ... 14-23
Discussion ... 14-23
The NUMBERS Option ... 14-23
Format ... 14-24
Discussion ... 14-24
The REM$ Option ... 14-24
Format ... 14-25
Discussion ... 14-25
The KEEPREMS Option ... 14-25
Format ... 14-26
Discussion ... 14-26
The KEEPREMS Compiler Option ... 14-26
The DISPLAY Option ... 14-27
Format ... 14-27
Discussion ... 14-28
Explicitly Invoking the Compiler Display ... 14-28
Implicitly Invoking the Compiler Display ... 14-28
The TRANSLATE Option ... 14-28
Format ... 14-29
Discussion ... 14-29
Translation Is Performed Only When Required ... 14-31
The LCASE Option ... 14-31
Format ... 14-31
Discussion ... 14-31
The User-Friendly Compiler Display ... 14-32
Invoking the Display ... 14-32
The Display Screen ... 14-33
Limitations of the Compiler Display ... 14-34
Compiler Display Example ... 14-35
Batch Files ... 14-35
Discussion ... 14-36
Example Batch Files ... 14-37
2200 Compatibility Issues ... 14-37

Compatibility with Earlier Revisions 14-38
New Statements 14-39
Program Stamp 14-39
Using the Compiler 14-39
Commonly Used Compiler Options 14-39

PORTING PROGRAMS AND DATA 15-1

Overview 15-1
Porting Options 15-2
"RAW" Diskette Transfer 15-2
Serial Communications 15-3
Native Operating System Utility Options 15-4
NPL General Backup/Recovery Utilities 15-5
Steps Required for Transfer 15-5
NPL General File Copy Utility Use 15-8
User Programs 15-9
Transferring Over Your Own Utility Programs 15-10
Executing Your Own Utility Programs 15-11
Transfer to and from the Wang 2200 15-11
Formatting a PC Interchange Disk on the 2200 15-12
File Types for Porting 15-14
Methods of Transferring 15-15

MIXED LANGUAGE PROGRAMMING 16-1

Overview 16-1
Interface to External Subroutines 16-2
GOSUB/DEFFN Versus FUNCTION/PROCEDURE Interface 16-3
Using the GOSUB'/DEFFN' Interface 16-3
Using the FUNCTION/PROCEDURE Interface 16-5
Callbacks to /PUBLIC FUNCTIONs from the External Library 16-7
External Subroutine Interface Specifications 16-8
Calling Conventions 16-8
Near Versus Far Pointers 16-8
C Versus Pascal Calling Convention 16-9
Parameter Format 16-10
Numeric Parameters 16-10
String Parameters 16-12
Execution of the Subroutine 16-13

- Return Value of the Subroutine 16-13
- Example of External Subroutine 16-14

External Function Interface Specifications 16-15

- Calling Conventions 16-15
- Near Versus Far Pointers 16-15
- C Versus Pascal Calling Convention 16-16
- Parameter Format 16-16
- Base Data Types 16-16
- Numeric Data Type 16-16
- String Data Type 16-18
- Arrays of Base Data Types 16-18
- Parameters Passed by Reference 16-18
- Declaring the Parameter Block 16-18
- Declaring the Function 16-20
- Accessing the Parameter Block 16-20
- Accessing Parameters Passed by Value 16-21
- Accessing Parameters Passed by Reference (/POINTER) 16-21
- Execution of the Function 16-22
- Return Value of the Subroutine 16-23
- Example of an External Function 16-23

NPL Function Callback Interface Specifications 16-25

- Locating the Function 16-25
- Validating the Function 16-26
- Calling the Function 16-26
- Passing Parameters by Value 16-26
- Passing Parameters by Reference (/POINTER) 16-26
- Passing the Address of the Function to be Called 16-27
- Making the Callback 16-28
- Checking for Errors and Getting the Return Value 16-28
- Restrictions on Callbacks 16-29
- Error Handling in Callbacks 16-29
- Example of External Callback to NPL Function 16-29

Writing a Mainline for External Subroutines 16-31

- Why a Mainline is Needed 16-31
- Calling NPL from the Mainline - The RunTime Subroutine 16-32
- **The RTP() Subroutine Calling Conventions** 16-32
- **The RTP() Subroutine Parameters** 16-32
- **The RTP() Subroutine Return Value** 16-32
- **Example of Mainline** 16-33

Writing the RTPEXT Subroutine 16-33

What Is the RTPEXT Subroutine? ... 16-33
The RTPEXT Calling Conventions ... 16-35
The RTPEXT Parameters ... 16-35
Information Requests about External DEFFN's ... 16-37
Named Aliases of External DEFFN' Routines ... 16-40
RTPEXT Requests for Information about External FUNCTIONs ... 16-41
Validating a FUNCTION/PROCEDURE ... 16-43
The RTPEXT Return Code ... 16-45
Example RTPEXT Subroutine ... 16-45
Another Example RTPEXT Subroutine ... 16-48
Writing a Test Program ... 16-50
Example Test Program ... 16-51
Example Linkage for Test Program ... 16-53
Running the Test Program ... 16-53
Running the Example Test Program ... 16-53
Making the External Library ... 16-54
Example Linkage for External Library ... 16-54
Using an External Library ... 16-55
Using the Example External Library ... 16-56
Cautions and Restrictions ... 16-57
Compatibility with Other NPL Applications ... 16-58
Memory Requirements ... 16-58
Isolation from NPL Environment ... 16-58
Resource Conflicts ... 16-58
Increased Expertise Requirement ... 16-59
Portability ... 16-59
17-1

EXTENDED EXAMPLES 17-1

Overview ... 17-1
Programming Style Used in Examples ... 17-3
Example 1: WHILE/WEND ... 17-4
Program Listing ... 17-4
Discussion ... 17-4
General Comments ... 17-5
Example 2: REPEAT/UNTIL ... 17-5
Program Listing ... 17-5
Discussion ... 17-5
General Comments ... 17-6

Example 3: Simple SWITCH/CASE 17-6
Program Listing 17-6
Discussion 17-6
General Comments 17-7
Example 4: Example Module (Library) 17-7
Program Listing 17-7
Discussion 17-7
General Comments 17-9
Example 5: Module using PUBLIC Sections 17-9
Program Listing 17-9
Detailed Discussion 17-11
General Comments 17-14
Example 6: Module using PROCEDURE 17-15
Program Listing 17-15
Detailed Discussion 17-16
General Comments 17-16
Example 7: A Recursive Function 17-17
Program Listing 17-18
Detailed Discussion 17-18
General Comments 17-19
Example 8: QSORT Routine 17-19
Program Listing 17-20
Discussion 17-22
General Comments 17-22
Example 9: Logical SWITCH/CASE 17-22
Program Listing 17-22
Discussion 17-22
General Comments 17-23
Example 10: Complex SWITCH/CASE 17-23
Program Listing 17-23
Results 17-24
Discussion 17-24
General Comments 17-24
Example 11: RECORDS/FIELDS 17-24
Program Listing 17-25
Results 17-25
Discussion 17-26
Example 12: Complex Use of RECORDs 17-27
Program Listing 17-27
Results 17-28

Discussion.. 17-28

UPGRADING OLDER NPL CODE A-1

Compatibility with Earlier Revisions A-1
Immediate Benefits of Modularization.. A-2
Immediate Benefits of Long Variable Names.. A-3
Upgrading Code Containing GOTO #... A-3
Useful NPL Enhancements.. A-5
Language Features A-5
Editor Enhancements.. A-6
Miscellaneous Enhancements.. A-6
Additional Considerations A-7

NPL ERRORS B-1

Overview.. B-1
Error Code Classes B-2
RunTime Program Errors and Warnings... B-2
Recoverable Versus Non-Recoverable Errors B-3
RunTime Program Errors B-4
Miscellaneous Errors B-4
ERR A00 - Not Implemented.. B-4
ERR A01 - Memory Overflow (Text < --> Variable Table). B-4
ERR A02 - Value Stack Overflow.. B-4
ERR A03 - Insufficient Memory for Buffer. B-4
ERR A04 - Operator Stack Overflow. B-4
ERR A05 - Program Line Too Long... B-5
ERR A06 - Program Protected.. B-5
ERR A07 - Illegal Immediate Mode Statement... B-5
ERR A08 - Statement Not Legal Here.. B-5
ERR A09 - Program Not Resolved. B-5
Syntax Errors B-5
Program Errors B-6
ERR P32 - Start greater than End.. B-6
ERR P33 - Line-Number Conflict.. B-7
ERR P34 - Illegal Value... B-7
ERR P35 - No Program in Memory... B-7
ERR P36 - Undefined Line-Number or CONTINUE Illegal. B-7
ERR P37 - Undefined Marked Subroutine. B-7

ERR P38 - Undefined FN Function. B-7
ERR P39 - FN's Nested Too Deep. B-7
ERR P40 - No Corresponding FOR for NEXT Statement. B-7
ERR P41 - RETURN Without GOSUB. B-8
ERR P42 - Illegal Image. B-8
ERR P43 - Illegal Matrix Operand. B-8
ERR P44 - Matrix Not Square. B-8
ERR P45 - Operand Dimensions Not Compatible. B-8
ERR P46 - Illegal Microcommand. B-8
ERR P47 - Missing Buffer Variable. B-8
ERR P48 - Illegal Device Specification. B-8
ERR P49 - Interrupt Table Full. B-9
ERR P50 - Illegal Array Dimensions or Variable Length. B-9
ERR P51 - Variable or Value Too Short. B-9
ERR P52 - Variable or Value Too Long. B-9
ERR P53 - Noncommon Variables Already Defined. B-9
ERR P54 - Common Variable Required. B-9
ERR P55 - Undefined Variable. B-9
ERR P56 - Illegal Subscripts. B-9
ERR P57 - Illegal STR Arguments. B-10
ERR P58 - Illegal Field/Delimiter Specification. B-10
ERR P59 - Illegal Redimension. B-10
Computational Errors B-10
ERR C60 - Underflow. B-10
ERR C61 - Overflow. B-10
ERR C62 - Division by Zero. B-10
ERR C63 - Zero Divided by Zero or Zero ^ Zero. B-10
ERR C64 - Zero Raised to Negative Power. B-10
ERR C65 - Negative Number Raised to Non-integer Power. B-10
ERR C66 - Square Root of Negative Value. B-10
ERR C67 - LOG of Zero. B-10
ERR C68 - LOG of Negative Value. B-11
ERR C69 - Argument Too Large. B-11
Execution Errors B-11
ERR X70 - Insufficient Data. B-11
ERR X71 - Value Exceeds Format. B-11
ERR X72 - Singular Matrix. B-11
ERR X73 - Illegal INPUT Data. B-11
ERR X74 - Wrong Variable Type. B-11
ERR X75 - Illegal Number. B-12

ERR X76 - Buffer Exceeded. B-12
ERR X77 - Invalid Partition Reference. B-12
ERR X79 - Invalid Password. B-12
Disk Errors B-12
ERR D80 - File Not Open. B-12
ERR D81 - File Full. B-12
ERR D82 - File Not in Catalog. B-12
ERR D83 - File Already Cataloged. B-12
ERR D84 - File Not Scratched. B-13
ERR D85 - Index Full. B-13
ERR D86 - Catalog End Error. B-13
ERR D87 - No End-of-File. B-13
ERR D88 - Wrong Record Type. B-13
ERR D89 - Sector Address Beyond End-of-File. B-13
I/O Errors B-13
ERR I90 - Disk Hardware Error. B-14
ERR I91 - Disk Hardware Error. B-14
ERR I92 - Timeout Error. B-14
ERR I93 - Format Error. B-14
ERR I94 - Format Key Engaged. B-14
ERR I95 - Device Error (Write Protect). B-14
ERR I96 - Data Error. B-14
ERR I97 - Longitudinal Redundancy Check Error. B-14
ERR I98 - Illegal Sector Address or Platter Not Mounted. B-14
ERR I99 - Read-After-Write Error. B-15
NPL Extended Errors B-15
ERR 200 - Handle table full / cannot expand. B-15
ERR 201 - Variable already referenced in program. B-15
ERR 202 - Variable not referenced in program. B-15
ERR 203 - Branch into body of a function or procedure. B-15
ERR 204 - Branch out of body of a function or procedure B-15
ERR 205 - This error code is reserved for future use by NPL. B-15
ERR 206 - FUNCTION/PROCEDURE not matched with END FUNCTION/PROCEDURE. B-15
ERR 207 - Nested functions/procedures not permitted. B-16
ERR 208 - Function/ procedure already defined. B-16
ERR 209 - Function/ procedure differs from /FORWARD declaration. B-16
ERR 210 - RETURN(value) not within FUNCTION body. B-16
ERR 211 - RETURN(value) incorrect type for FUNCTION. B-16
ERR 212 - End Structure name does not agree. B-16

ERR 213 - Arguments of function/procedure incorrect. B-16
ERR 214 - No such function or procedure has been declared or included... B-16
ERR 215 - Argument for non-constant /POINTER parameter is not modifiable. B-17
ERR 216 - Not legal after an :ERROR clause. B-17
ERR 217 - Only simple IF permitted here. B-17
ERR 218 - This error code is reserved for future use by NPL..................... B-17
ERR 219 - This error code is reserved for future use by NPL..................... B-17
ERR 220 - No enclosing WHILE, REPEAT or FOR/BEGIN loop structure. B-17
ERR 221 - This error code is reserved for future use by NPL..................... B-17
ERR 222 - Structured IF not matched with END IF. B-17
ERR 223 - Structured ELSE not matched with END IF. B-17
ERR 224 - REPEAT not matched with UNTIL. B-18
ERR 225 - WHILE not matched with WEND. B-18
ERR 226 - FOR/BEGIN not matched with NEXT. B-18
ERR 227 - Recursive/parameter value not legal in declaration................... B-18
ERR 228 - Variable already declared with different access. B-18
ERR 229 - Local variable already declared. B-18
ERR 230 - Local function already declared. B-18
ERR 231 - /BEGINS declaration not preceded by /FORWARD declaration. B-18
ERR 232 - Structured statements not allowed as THEN or ELSE clause.... B-19
ERR 233 - Recursive arrays cannot be redimensioned................................ B-19
ERR 234 - CASE located after default CASE for this SWITCH. B-19
ERR 235 - SWITCH not matched with CASE/END SWITCH. B-19
ERR 236 - CASE expression not same type as SWITCH. B-19
ERR 237 - FUNCTION/FNx() term not legal in declarations. B-19
ERR 238 - Conflicting options in function declaration. B-19
ERR 239 - /FORWARD declaration not followed by /BEGINS declaration. B-19
ERR 240 - Invalid module name. B-20
ERR 241 - Circular INCLUDE/USE references. B-20
ERR 242 - Internal Error. B-20
ERR 243 - No EXTERNAL function is defined with this name and these parameters. B-20
ERR 244 - Already defined as Non-public Variable. B-20
ERR 245 - Already defined as PUBLIC function. B-20
ERR 246 - Already defined as PUBLIC section. B-20
ERR 247 - PUBLIC not matched with END PUBLIC. B-20
ERR 248 - No such PUBLIC section. B-21
ERR 249 - Another procedure is already marked as /MAIN, /EXIT. B-21
ERR 250 - This error code is reserved for future use by NPL..................... B-21

ERR 251 - Forced error exit to /EXTERNAL from callback. B-21
ERR 252 - Callbacks not available. B-21
ERR 253 - Nested RECORD declarations not permitted. B-21
ERR 254 - RECORD not matched with END RECORD. B-21
ERR 255 - RECORD exceeds maximum allowed length. B-21
ERR 256 - FIELD statement not in RECORD declaration. B-21
ERR 257 - PUBLIC section is not in an INCLUDEd module...................... B-22
ERR 258 - This error code is reserved for future use by NPL..................... B-22
ERR 259 - Statement label already defined. B-22
ERR 260 - No such statement label in mainline. B-22
ERR 261 - No such statement label in same FUNCTION/PROCEDURE body. B-22
ERR 262 - FUNCTION used by declaration did not complete. B-22
ERR 263 - DIM/RECURSIVE must be in a FUNCTION/PROCEDURE body. B-22
ERR 300 - Incorrect number or types for parameters of GOSUB’. B-23
ERR 301 - Insufficient memory or bad COMSPEC for $SHELL............... B-23
ERR 302 - No such PUBLIC FIELD name. B-23
ERR 303 - No such PUBLIC PROCEDURE/FUNCTION name. B-23
ERR 304 - Cannot expand array variable. B-23
RunTime Program Warnings B-23
WARN 01 - Warning: Programs merged. B-23
WARN 02 - Warning: Program moved to end of platter. B-24
WARN 03 - Warning: RUN statement is in progress. B-24
Error Message Files B-24
Operating System Level Errors B-24

COMPILER ERRORS C-1

Compiler Options Errors C-1
Cannot open LSTLOC (or ERRLOC) file C-1
Cannot open SRCLOC (or OBJLOC) as a diskimage file............................. C-2
Extensions not permitted C-2
Fields may not contain embedded spaces C-2
Filename too long C-2
LSTFORMAT option must be one of; .LST, .SRC, 2200, 2200S C-2
Missing Option after (literal) C-3
No source programs specified C-3
Non-Hex digits in option C-3
OBJFORMAT must be one of; SCRAMBLED, UNSCRAMBLED C-3
Option is not a number C-3
Option must be one of; ON, OFF C-4

Option too long C-4
SRCLOC (or OBJLOC) is not a diskimage file C-4
Too many parameters on command line C-4
Unknown option on command line (literal)=(parameter) C-4
Diskimage index is full C-5
Filename in catalog is not a program C-5
No names found in (location) for wildcard (wildcard) C-5
Object program was moved to end of catalog in object diskimage C-5
Out of space in Object diskimage C-6
Program Syntax Errors C-6
Program Syntax Warnings C-6
ALL(x) & X$ is illogical syntax, likely an error C-7
Cannot do RESTORE LINE / DATA checks due to table overflow C-7
ELSE clause not preceded by IF statement C-7
ERROR clause not preceded by statement C-7
Programs which use SPACE(K) may need adjustment ... under "THIS O/S" C-7
RESTORE LINE does not point to DATA statement in this module C-7
SAVE DA/DC is not supported at run time C-7
Statement requires RTP Rev x.xx.xx or later to execute C-7

NPL DIRECTLY SUPPORTED TERMINAL CHARACTERISTICS D-1

Overview D-1
General Considerations D-2
Determination of Terminal Type D-2
Supported Terminals D-2
Support of Native Operating Systems and Utilities D-3
Altos III D-3
Screen Character Set D-3
Downloadable Fonts D-3
Alternate Character Set (Pixel Graphics) D-3
Non-English Character Sets D-3
Character Translation D-4
Box Graphics D-4
Attributes D-4
Cursor Handling D-5
Support for 132-Column Mode D-5
Keyboard Characteristics D-5
Local Printer Support D-8

Configuration Requirements D-9
Use as a Remote Terminal D-9
Use with Native Operating System Functions and Utilities D-9
Altos V D-10
Screen Character Set D-10
Downloadable Fonts D-10
Alternate Character Set (Pixel Graphics) D-10
Non-English Character Set D-10
Character Translation D-11
Box Graphics D-11
Attributes D-11
Cursor Handling D-12
Support for 132-Column Mode D-12
Keyboard Characteristics D-12
Local Printer Support D-15
Configuration Requirements D-16
Use as a Remote Terminal D-16
Use with Native Operating System Functions and Utilities D-17
Bull HDS 1 ..... D-17
Screen Character Set D-17
Downloadable Fonts D-17
Alternate Character Set (Pixel Graphics) D-17
Non-English Character Sets D-17
Character Translation D-17
Box Graphics D-17
Attributes D-18
Cursor Handling D-18
Support For 132-Column Mode D-18
Keyboard Characteristics D-18
Configuration Requirements D-22
Use As Remote Terminals D-22
Use With Native Operating System Functions And Utilities D-22
Bull HDS 3 ..... D-23
Screen Character Set D-23
Downloadable Fonts D-23
Alternate Character Set (Pixel Graphics) D-23
Non-English Character Set D-23
Character Translation D-24
Box Graphics D-24
Attributes D-24

Cursor Handling D-25
Support for 132-Column Mode D-25
Keyboard Characteristics D-25
Local Printer Support D-28
Configuration Requirements D-29
Use as a Remote Terminal D-29
Use with Native Operating System Functions and Utilities D-29
DEC VT100 .... D-30
Screen Character Set D-30
Downloadable Fonts D-30
Alternate Character Set (Pixel Graphics) D-30
Non-English Character Set D-30
Character Translation D-30
Box Graphics D-30
Attributes D-30
Cursor Handling D-31
Support for 132-Column Mode D-31
Keyboard Characteristics D-31
Local Printer Support D-34
Configuration Requirements D-34
Use as a Remote Terminal D-34
Use with Native Operating System Functions and Utilities D-35
DEC VT200 Series D-35
Screen Character Set D-35
Downloadable Fonts D-35
Alternate Character Set (Pixel Graphics) D-35
Non-English Character Set D-35
Character Translation D-36
Box Graphics D-36
Attributes D-37
Cursor Handling D-37
Support for 132-Column Mode D-37
Keyboard Characteristics D-37
Local Printer Support D-40
Configuration Requirements D-41
Use as a Remote Terminal D-41
Use with Native Operating System Functions and Utilities D-41
IBM 3151 .... D-42
Screen Character Set D-42
Downloadable Fonts D-42

Alternate Character Set (Pixel Graphics).. D-42
Non-English Character Sets.. D-42
Character Translation D-42
Box Graphics.. D-42
Attributes D-42
Cursor Handling D-43
Support for 132-Column Mode.. D-43
Keyboard Characteristics D-43
Local Printer Support D-46
Configuration Requirements D-46
Use as Remote Terminals... D-46
Use with Native Operating System Functions and Utilities D-46
NCR 4970 .... D-47
Screen Character Set D-47
Downloadable Fonts.. D-47
Alternate Character Set (Pixel Graphics).. D-47
Non-English Character Set... D-47
Character Translation D-48
Box Graphics.. D-48
Attributes.. D-48
Cursor Handling.. D-49
Support For 132-Column Mode... D-49
Keyboard Characteristics... D-49
Local Printer Support.. D-52
Configuration Requirements.. D-52
Use As Remote Terminals... D-53
Use With Native Operating System Functions And Utilities................. D-53
Spectrix SPX 701 .... D-54
Screen Character Set.. D-54
Downloadable Fonts.. D-54
Alternate Character Set (Pixel Graphics).. D-54
Non-English Character Set... D-54
Character Translation D-54
Box Graphics.. D-54
Attributes.. D-55
Cursor Handling.. D-55
Support For 132-Column Mode... D-55
Keyboard Characteristics... D-55
Local Printer Support.. D-58
Configuration Requirements.. D-58

Use As Remote Terminals D-59
Use With Native Operating System Functions And Utilities D-60
Wang 2110A D-60
Screen Character Set D-60
Downloadable Fonts D-60
Alternate Character Set (Pixel Graphics)........ D-60
Non-English Character Set D-60
Character Translation D-60
Box Graphics D-60
Attributes D-60
Cursor Handling D-61
Support for 132-Column Mode D-61
Keyboard Characteristics D-61
Local Printer Support D-64
Configuration Requirements D-65
Use as a Remote Terminal D-65
Use with Native Operating System Functions and Utilities D-65
Wang 2x36 DE/DW D-66
Screen Character Set D-66
Downloadable Fonts D-66
Alternate Character Set (Pixel Graphics) D-66
Non-English Character Set D-66
Character Translation D-66
Box Graphics D-66
Attributes D-66
Cursor Handling D-67
Support for 132-Column Mode D-67
Keyboard Characteristics D-67
Local Printer Support D-71
Configuration Requirements D-71
Use as a Remote Terminal D-71
Use with Native Operating System Functions and Utilities D-71
Wyse 50 .... D-72
Screen Character Set D-72
Downloadable Fonts D-72
Alternate Character Set (Pixel Graphics) D-72
Non-English Character Sets D-72
Character Translation D-72
Box Graphics D-72
Attributes D-72

Cursor Handling D-73
Support for 132-Column Mode D-73
Keyboard Characteristics D-73
Local Printer Support D-76
Configuration Requirements D-77
Use as a Remote Terminal D-77
Use with Native Operating System Functions and Utilities D-77
Wyse 60, 150, and 160 .... D-78
Screen Character Set D-78
Downloadable Fonts D-78
Alternate Character Set (Pixel Graphics) D-78
Non-English Character Set D-78
Character Translation D-78
Box Graphics D-78
Attributes D-79
Cursor Handling D-79
Support For 132 Column Mode D-79
Keyboard Characteristics D-79
Local Printer Support D-85
Configuration Requirements D-86
Use As Remote Terminal D-86
Use with Native Operating System Functions and Utilities D-86
Wyse 370 .... D-87
Screen Character Set D-87
Downloadable Fonts D-87
Alternate Character Set (Pixel Graphics) D-87
Non-English Character Set D-87
Character Translation D-88
Box Graphics D-88
Attributes D-89
Cursor Handling D-89
Support For 132-Column Mode D-89
Keyboard Characteristics D-89
Local Printer Support D-93
Configuration Requirements D-93
Color Support D-93
Use As Remote Terminals D-93
Use With Native Operating System Functions And Utilities D-94

KEYBOARD KEYWORDS E-1

DECOMPILER COMPATIBILITY F-1

Overview....F-1
Differences In Source Code Decompilation....F-2
Differences Between Rev. 1.03 and Later Revs. F-9
Summary....F-10

"RAW" DEVICE COMPATIBILITY CHART G-1

"Raw" Diskette Chart G-1

GLOSSARY 1-1

CHAPTER 1

INTRODUCTION

1.1 Overview

This chapter provides a general introduction to the Niakwa Programming Language (NPL) and its role in the Niakwa Development Environment. In addition, the technical features and limitations of NPL are presented along with a high-level overview of the NPL virtual machine environment. Developers new to NPL are encouraged to begin their NPL training with this chapter.

Section 1.2 discusses the NPL Development Environment.

Section 1.3 discusses NPL's features.

Section 1.4 discusses NPL's limitations.

Section 1.5 provides a high-level overview of the NPL virtual machine environment.

1.2 The Niakwa Development Environment

The Niakwa Development Environment has been designed to provide the vertical market developer the best application development environment in which to develop truly portable applications for small to medium-sized companies or departments. Niakwa has brought the best third-generation language features (speed, flexibility, etc.) and combined them with the powerful functions available in traditional, fourth-generation languages to produce a development environment that offers the strengths of both generations, but without the weaknesses.

The Niakwa Development Environment allows developers unparalleled portability as well as the tools necessary to develop powerful applications quickly and efficiently.

1.2.1 The Niakwa Programming Language (NPL)

The Niakwa Programming Language (NPL) is a portable, high-level programming language that operates on hundreds of the computer industry's most popular computers, under eight major operating environments--AIX, MS-Windows, MS-DOS, Novell NetWare, SuperDOS, UNIX, VMS, and Xenix. NPL provides its users full object and source code compatibility across all Niakwa-supported platforms, thereby protecting its users from reliance on a single technology.

NPL is based upon the original Dartmouth BASIC, however, many extensions beyond the Dartmouth specification have been incorporated into the language which render it an excellent tool for the development of software applications in both the commercial and scientific fields. This is evidenced by Niakwa's worldwide distribution network of over 500 professional software developers/resellers who represent more than 80,000 users.

An unusual characteristic of NPL not found in other languages is that NPL incorporates many operating system-like specifications in its language definition. As one example of this, a detailed and flexible protocol for controlling external devices such as workstations, disk devices, printers, and communication devices is all formalized as an integral part of NPL itself.

Specifications such as these, which are internal to NPL, tend to form a consistent "shell" around NPL-written applications. This shell or "virtual machine environment" contributes greatly to their portability to new operating systems and hardware.

NPL is implemented in the form of three physical software products: the NPL Interpreter, the NPL Compiler and the NPL RunTime Program. The common link between all three products is that they all create and/or execute NPL pseudo-code (p-code). NPL p-code is a highly portable and efficient representation of NPL source programs and is an intermediate form between pure object and pure source. From the simplest view:

- The Interpreter (RTI) both creates and executes p-code.
- The RunTime Program (RTP) executes p-code.
- The Compiler (B2C) creates p-code.

The intended purpose of each product and considerations for selecting each for certain tasks is explained below.

The NPL Interpreter is used by software authors as a tool for the efficient development and maintenance of programs in NPL. Using a powerful "Immediate Mode", the Interpreter supports real-time: source code editing, syntax checking, variable inspection, variable modification, program halting, single instruction stepping and trapping, instruction logic tracing and many other program development and debugging aids.

This is all possible due to Niakwa's use of incremental compiler technology that allows for interactive program development and debugging. In addition, this also makes the Interpreter appropriate for use at end-user sites for the execution of NPL applications in a "production" environment. The relatively small code size and high performance characteristics of the Interpreter allow its development capabilities to be used for end-user support without penalty to application performance.

The NPL RunTime program executes p-code generated by either the Interpreter or Compiler. The RunTime program is used for the execution of NPL applications at end user sites where the smallest possible memory overhead is desired. The RunTime program executes p-code only, it does not allow for program inspection, modification, variable inspection, etc., as provided in the Interpreter.

The NPL Compiler is used by software authors primarily as a tool to assist the Interpreter in NPL program manipulation, which is most conveniently performed on a batch basis.

As one example, the compiler can be used to process an entire library of NPL programs developed under the Interpreter (which contain source REMarks, documentary spacing, indentation, etc.) and produce a "production" version of the same library with REMarks, etc., removed all as one operation.

Another typical use of the compiler is in porting to or from the Wang System 2200 processor. The NPL Compiler can both accept and generate Wang 2200 Basic-2 "atomized" programs to/from NPL p-code.

P-code generated by the compiler is fully executable by the NPL Interpreter or the NPL RunTime Program without modification.

Creating NPL Programs

An NPL program is defined as any file of codes (called pseudo-code or p-code) which is directly executable by the NPL Interpreter or the NPL RunTime Package. This special p-code storage format of NPL programs is employed due to its highly efficient memory usage, execution time performance, and portability.

NPL programs can be created using the NPL Interpreter, the NPL Compiler, and can be dynamically created by using the $SOURCE/$OBJECT program functions.

Interpreter--With the Interpreter, NPL programs are created by entering new source program lines in Immediate Mode with the line editor. As the source program lines are entered, they are immediately compiled by the Interpreter into executable p-code and stored in program text memory. The original source line is then effectively discarded. Whenever the original source line must be subsequently viewed (for listing, editing, debugging, etc.) it is regenerated from the stored p-code by a built-in de-compiler.

Once a program has been created in this way, the completed program p-code in memory can be saved to disk using a number of Immediate Mode SAVE commands.

The NPL Interpreter is the primary means used for creating, editing, maintaining, and debugging NPL programs.

With the Interpreter, NPL program editing is predominantly performed by the combined use of two different features of the NPL Interpreter. These are Immediate Mode commands and the Line Editor.

Immediate Mode commands are integral to program editing in performing such tasks as loading and saving programs, listing programs, cross-referencing variables and line numbers, searching for text strings, renumbering and merging of program text and many other functions.

The NPL Line Editor is used for the initial creation of program text and for editing existing program text which resides in memory on a line-by-line basis. This line editor is closely coupled with the Interpreter and,as a result,allows for immediate syntax checking of commands and program lines as they are entered. Line editing is further enhanced by the use of Special Function keys which aid in the rapid correction of program text in a minimum of keystrokes.

The Interpreter is also fundamental to the NPL program debugging capabilities due to the high level of integration with program editing. This is done by design because an essential part of program debugging is the ability to acquire control of an executing program. Once control is established, the program may then be inspected by the programmer in its various stages of execution. NPL allows control over an executing program by invoking Immediate Mode.

Compiler--NPL programs are created with the compiler by reading source program lines from a file, compiling them, and then saving the resultant p-code into another file. With the compiler, the source program storage format can be in one of several different forms. ASCII format, Wang 2200 atomized format, and p-code format itself are all valid as input source program format to the compiler.

$SOURCE/$OBJECT--$SOURCE, $OBJECT, and their associated library functions are programmable instructions that can be used by an NPL program to create other NPL programs dynamically at execution time. The $OBJECT function receives a specified line of program source (in ASCII) and returns the executable p-code for the given source to a specified variable. $SOURCE performs the opposite function. By saving p-code generated in this way to disk, the resulting p-code program may be then loaded and executed like any other NPL program.

NOTE: NPL programs cannot be directly edited by native operating system editors since they are stored in p-code format and native editors generally operate on ASCII format files. However, the NPL Compiler can be used to easily convert p-code into ASCII (on a batch basis), thereby enabling the use of native editors or any other native utilities to manipulate NPL source ASCII files. Once the manipulation is complete, the compiler is used to convert from ASCII back to p-code for execution.

NPL Add-Ons

In addition to the above, Niakwa also provides access to the following add-ons with NPL.

Utilities--The NPL Development Package also contains a set of convenient utilities designed specifically to aid NPL developers (i.e., file backup, recovery, copy, diskimage creation, listing, scratching, device settings, options, etc.).

Libraries--As of Release IV, NPL allows NPL developers the ability to develop and use NPL libraries. By using libraries, developers can easily integrate added functionality without spending time writing the routines. Niakwa is acting as a clearinghouse for these libraries by offering a BBS forum and periodic catalog listing of available NPL libraries. Please contact your Niakwa Sales Representative for more information on the Niakwa Library Program.

1.2.2 The Niakwa Development Tools

NPL is complemented in the Niakwa Development Environment by the Niakwa Development Tools product line. This is a group of products that bring features found in fourth generation languages to users of NPL. The Niakwa Development Tools product line currently consists of the Niakwa Data Manager for data management and Intelligent Query (IQ) for structured queries and report generation.

Niakwa Data Manager

The Niakwa Data Manager is the heart of the Niakwa Development Tools product line. Using the external call facility of NPL, the Data Manager is an application program interface (API), developed by Niakwa, to provide a consistent, fully portable interface to commonly used ISAM products, such as C-ISAM, Btrieve and RMS. The primary feature of the Data Manager is data independence. Data independence means NPL files are no longer stored in a proprietary format. Therefore, NPL applications can now share files with popular third-party products such as Informix and Focus. In addition, NPL applications can be enhanced with third-party utilities such as IQ because NPL files are stored in a standard format. Other significant benefits of the Data Manager are improved performance, often substantial, and increased productivity because application developers can offload ISAM maintenance and development to the Data Manager.

Intelligent Query

Once the Data Manager is in use, Intelligent Query (IQ), from Programmed Intelligence, offers simple, English-like access to NPL data files--without the time and expense of special programming. IQ allows end-users to present NPL application data in four different ways:

- Preformatted reports, queries, and labels
- Custom reports, queries, and labels
- XY-graphs or histograms for visual interpretation
- Data export capabilities to other programs (i.e., Lotus 1-2-3 and dBase)

IQ also allows end-users to merge NPL application data with standard ASCII, Lotus 1-2-3, or dBase files for even more useful information. Intelligent Query enhances NPL applications by giving end-users new and more effective ways of retrieving company information.

1.2.3 Niakwa's Objectives

Niakwa's mission is to provide high quality, high performance, highly portable application development software and services to application developers worldwide who target small to medium-sized companies or departments. The Niakwa Development Environment and its primary components are designed to meet this goal and provide professional developers the best solution for developing mission-critical applications.

Niakwa's future plans related to NPL and the Niakwa Development Environment are designed to service our existing customer base, while positioning the company for future growth. This is being done by:

- Servicing the core Niakwa market with our established high quality standards applied to products and services.
- Maintaining our technological edge with an aggressive development agenda.
- Continuing to embrace computer industry standards (we will always be on the major operating environments--whatever they may become).

- Expanding our developer/reseller base by pursuing additional market opportunities.

1.3 NPL Technical Features

Continuing in the tradition of its successful predecessors, NPL Release IV is the applications programming language solution for the future. While remaining fully upward compatible with traditional Wang Basic-2, NPL offers significant improvements which can substantially improve application development productivity. In addition, NPL includes a set of powerful utilities, access to commercially available library products, and integration with the Niakwa Development Tools product line.

Specifically, NPL's technical features as described in the following sections.

1.3.1 Performance

Application programs written in NPL exist in a pseudo-code (p-code) format that allows faster execution than a pure interpreter, without sacrificing the convenience. In addition, NPL has added many high-level functions that quickly perform entire operations that, in other languages, require significantly more code. Niakwa also maintains two sets of source code. The first is an assembler set for all Intel-based systems that is optimized specifically for increased performance on the Intel chips sets. The second is an optimized C version used for all, generally faster, non-Intel based chip sets. This C version allows for greater flexibility in terms of future platforms supported than the assembler version.

1.3.2 Interactive Environment

Using incremental compiler technology, NPL is easy to learn and makes coding and debugging rapid and straightforward. The interpretive NPL environment and editing facilities allow developers to interrupt an executing program, examine the state of the application, edit the code and modify variables, and resume execution from the point of interruption. This capability can greatly simplify real-time customer support and customization. NPL also allows the developer to define appropriate character sets and configuration tables, as well as interactive program editing methods.

1.3.3 Debugging Facilities

Niakwa also provides a series of powerful debugging commands that make NPL one of the easiest languages to debug. These commands include

For inspection of program logic:

```
STEP
STEP LINE # RANGE
STEP OFF
CONTINUE RETURN
CONTINUE NEXT
CONTINUE LOAD
TRACE
TRACE #
TRACE '
TRACE V
TRACE OFF
LIST STACK
LIST DT
MODULE
```

For logic modification:

```
GOTO
RETURN
RETURN CLEAR
NEXT
RUN #
RUN
```

For inspection of program variables:

```
PRINT
PRINT USING
MAT PRINT
LIST DIM
LIST STACK DIM
```

For automatic variable inspection:

```
TRACE V
```

For variable modification:

```
LET alpha and numeric
INPUT LINPUT
KEYIN (non-polling)
PACK
UNPACK
CONVERT
MAT COPY
MAT MOVE
MAT SORT
DATA LOAD DA
DATA LOAD DC
DATA LOAD BA
Etc.
```

For inspection of program text:

```
LIST commands (many)
```

For inspection and modification of the program environment:

```
LIST DT
LIST DC
$SHELL
SPACE instruction
LIST PUBLIC
```

For program editing:

```
RENUMBER
RENAME [V, etc.]
LOAD [merge]
CLEAR P
```

For defining and inspecting system variables:

```
$BOXTABLE
$KEYBOARD
$OPTIONS
$MACHINE
$PSTAT
$SCREEN
$PRINTER
$PROGRAM
$REV
$KEEPREMS
#TERM
#PART
#ID
#NETID
```

1.3.4 Portability

Portability has been the key to NPL's success. NPL's virtual machine environment provides absolute portability (binary-level compatibility) of applications and data across all contemporary operating environments including MS-DOS, MS-Windows, Novell NetWare, UNIX, Xenix, SuperDOS, AIX, and VMS. In addition, users of Niakwa's Data Manager are provided a set of utilities to migrate NDM data files from one native ISAM to another.

1.3.5 Multi-User

NPL was specifically designed for multi-user systems such as UNIX, SuperDOS, and Novell NetWare networks. Critical features such as unique terminal, task, and user identification as well as file locking are integral to NPL. The optional Niakwa Data Manger adds another level of data manipulation features, utilities, and performance gains for multi-user installations.

1.3.6 Program Constructs

NPL offers the following structured constructs:

```
WHILE/WEND
REPEAT/UNTIL
IF/ELSE/END IF
SWITCH/CASE
FOR/BEGIN/NEXT
```

1.3.7 Program Modules (Libraries)

Program modules allow development of true NPL "black box" routines. Once a group of functions is fully developed and debugged, it can be incorporated as a module and can be used by any application without concern for identifier names (which are local to the module unless defined as public) or line number (line numbers are always local to the module) conflicts (pointer parameters are supported for transfer of large blocks of data). All the calling application needs to know are the entry points and the calling conventions (parameter lists and return values). Thus, true application-independent libraries can be written in NPL.

NOTE: Niakwa is acting as a clearinghouse for NPL libraries and Niakwa's Library program is designed to promote the distribution of these modules within the NPL community. Contact your Niakwa Sales Representative for more information on the Niakwa Library program.

1.3.8 FUNCTION Interface

The FUNCTION/PROCEDURE interface provides a structured methodology for subroutine development and access. Designed to perform a specific operation, functions and procedures can have a unique identifier name and parameter list, can support value or reference parameter types, may be defined as public or private, can be used to access external routines written in C, and may be used to return error codes.

NOTE: FUNCTIONs and PROCEDUREs are very similar; the main difference is that FUNCTIONS can return values, and PROCEDUREs cannot.

1.3.9 Mixed Language Programming

NPL allows integration to external subroutines written in other non-NPL languages. This offers the NPL developer the ability to increase execution speed for selected processor-intensive functions and the capability to access resources and features of a specific environment. NPL currently supports C, PASCAL (limited), and ASSEMBLER (limited).

1.3.10 Data Conversion Statements

NPL supplies several statements designed explicitly to aid in the conversion of data. These include statements designed to: convert alpha-variables to numeric-variables or vice-versa; store a list of numeric values in an alpha-variable in a packed decimal format defined by an image; pack or unpack a buffer-variable with any number of values in various formats; convert the integer result of a numeric-expression into character string format (binary representation); and convert the binary value of a specified alpha-variable or literal string to a numeric value.

1.3.11 Advanced Math Functions

NPL provides many advanced math functions in NPL itself (i.e., trigonometric functions, boolean statements, etc.). Numeric accuracy is maintained to 13 significant digits.

1.3.12 String Handling

NPL supports dynamic strings of 64K on non-32-bit operating environments and up to 2 GB on 32-bit operating environments. Strings are easily concatenated, subdivided and searched. Data formatting and low-level bit operations are also supported.

1.3.13 Advanced Matrix Manipulation

NPL provided both numeric and string arrays of up to two dimensions. Arrays are dynamic and allow programmer-specified subscript ranges. Array input, output, and manipulation functions are all provided. Matrix arithmetic for numeric arrays is supported.

1.3.14 Device Independent

NPL's virtual machine environment provides not only portability but sophisticated control of all native device types (i.e., terminals, printers, storage devices, serial I/O devices, etc.) across all NPL-supported operating environments.

1.3.15 Native Operating System Calls

$SHELL allows the calling of a native operating system command or a non-NPL executable file from within NPL. After execution, NPL is restored to its previous condition.

1.3.16 Data Verification and Error Handling

Interactive applications require sophisticated error handling. NPL offers error handling at the program, line, function, and command levels. Additional functions make it easy to create generic error traps. Built-in input verification makes it easy to design "user-proof" applications.

1.3.17 HELP Subsystem

The NPL HELP processor is a feature which provides the user with access to a series of information screens and options relative to the operation of a particular application program or the NPL RunTime program itself. Nested entries and index files are all available with full multi-screen and mouse support (on operating environments where NPL supports the operation of a mouse).

1.3.18 $DEMO

The $DEMO feature of NPL provides a mechanism that allows for the generation of "self-demonstrating" software. With this technique, live software is operated by using keystrokes and informational text from an ASCII text file as opposed to the keyboard. This allows for an application software demonstration that uses a prepared "script" to automatically fill in all operator-entered fields. The operator can be provided with informational text along the way and is only periodically required to press the space bar to continue with the demonstration. In addition, automatic tutorials can be created for self-training at end-user sites.

1.3.19 Program Encryption

Sensitive code can be protected with NPL exclusive "scramble-protected" mode. This function can be used from the NPL Interpreter or through the Compiler. This feature uses a complex algorithm to actually encrypt the source program. This encryption does not allow for listing or saving of the program that has been scrambled.

1.3.20 Wang 2200/Basic-2 Upward Compatibility

NPL offers nearly 100% upward compatibility for users moving off Wang 2200/Basic-2 systems. In addition, many of the existing Wang peripherals (i.e., terminal, printers, etc.) are also supported (this is operating system-dependent).

1.4 NPL Limits

The following represents a summary of the major NPL limitations.

NOTE: Although NPL has preestablished theoretical limits, as presented below, many of these values are constrained by one or more possible system limits. Consequently, the actual NPL limits are often operating environment-dependent (actual limits are also subject to limits due to system memory or the NPL handle table). Where notations appear, practical limits are likely to be encountered long before theoretical limits.

Maximum number of users = 512

Maximum number of lines in one program module = 32,118 (H)

Maximum number of variables in one program module = 65,534 (H/2)

Maximum level of recursion FUNCTION/PROCEDURE = 64 or 512 on 32-bit operating environments (W)

Maximum variable size = 64K or 2 GB on 32-bit operating environments (M)

Maximum variable name length = 255 characters

Variable ranges:

String length: 0 to 65,534 or 2 GB on 32-bit operating environments

Signed 8-bit integers: -140,737,488,355,328 to +140,737,488,355,327

Real numbers:

Signed 8-bit: +/-140737488355327

Signed 16-bit: +/-140737488355327 E+/-32767 (exponent of 10)

NOTE: In NPL, the terms "integer", "long integer", and "single/double precision" do not apply. The data type for all NPL numerics is the same, and converted to "floating point" as required. The above numbers represent the largest for which no precision is lost.

Array size (all elements) = 64K or 2 GB on 32-bit operating environments (M)

Number of dimensions allowed = 2

Array subscript value = 0 to 65,535 or 0 to 2 GB on 32-bit operating environments.

Maximum sizes associated with FUNCTION/PROCEDURE size = (M)(H)

Number of arguments passed = 0 to 255

Data file numbers (NPL) = 255 (F)

Number of logical files in one diskimage = 4079

Data file record number (NPL) = 64K normal, 16 MB extended

Data file size (NPL) = 16 MB normal, 4 GB extended

Error message numbers = 0 to 999

$HELP entries per index = 3270 or limited only by physical memory on 32-bit operating environments(U)

Maximum program size = limited only by physical memory

NOTES:

(H) Each program line requires an entry in the handle table. The maximum handle table contains 16,383 entries or 65,536 entries on 32-bit operating environments (requires the use of the /H start-up option to exceed 16K handles).

(H/2) Each variable can require from 0 to 2 entries in the handle table. One entry may be required for the variable data (if not recursively allocated), one entry is required for each distinct long identifier used.

(W) Recursion may be limited by the availability of temporaries used for storage of intermediate numeric expressions and constants, if more than two intermediate results are pending at each level of recursion.

(M) The limiting size of variables in a 32-bit operating environment are limited by available physical memory (or in a VMM environment by the size of the backing store dedicated to virtual memory) and any per-process quotas imposed by the operating environment.

(F) NPL allows up to 256 open data files if a SELECT #nn statement appears in the program. However, most operating environments have a smaller limit to the number of files that can be open concurrently. Consequently, if the files are on different diskimages, the files may not be concurrently accessible.

(U) The limit of help entries is imposed by the indexing utility based on the available array size on the host operating environment.

1.5 The NPL Virtual Machine Environment

NPL's strength is the consistent interface it provides its users from one operating system to another. This is maintained by creating a "virtual machine environment" on each hardware platform NPL supports. Because of this environment, NPL applications can operate on many different types of hardware while still maintaining a consistent look and feel.

NPL provides a generalized method of accessing peripheral devices that is hardware-independent. This method employs the use of unique three-character "device addresses" which are used by NPL programs to reference external devices.

NPL "devices" are logical devices, not physical devices. Each logical device address is mapped to a physical native operating system device or file name by use of the Device Equivalence Table (DET).

In addition to the three-character device address, NPL provides a mechanism for establishing default addresses for subsequent use by NPL programs. This is accomplished by the Internal Device Table (DT).

NOTE: This implementation of device addresses is fully compatible with the Wang 2200.

NPL provides mechanisms for gaining exclusive access to selected devices, an essential capability in multi-user environments.

For NPL I/O routines to function properly, the I/O operation must be directed to the correct native operating system file or device. The Device Equivalence Table (DET) is used to establish an equivalence between the NPL logical device addresses and the equivalent physical files or devices maintained within the native operating system for the particular hardware.

For each NPL device address that is used by a given application, the DET will contain an entry with that particular address, and the equivalent native operating system file or device name. During program execution, whenever the RunTime Program encounters a logical NPL device address in program syntax, the DET is inspected to determine the equivalent physical native operating system device or file name that it references. The I/O operation is then directed to that file or device. If it is the first access to the specified NPL device address since the device equivalence was established, a logical open is performed for the native operating system file or device before the I/O operation is performed. This mapping operation is entirely transparent to the NPL program.

1.5.1 Input/Output

NPL can be used to address any native operating system device (access is limited to simple read or write operations unless a specialized device driver is used). The simplest method of directing output to such a device is to PRINT to it (NPL will permit the automatic translation of characters as they are sent to a print device using a simple look-up table).

NOTE: Print output to an native operating system file is also supported.

The Runtime also contains limited ability to read and write raw data from a serial port. With this functionality, serial devices are easily supported.

Mouse support is also available on many of the NPL supported platforms by use of a RunTime start-up option.

1.5.2 Storage Devices

Disk I/O for NPL applications can be implemented in two ways. The first, and most common way, provides for a high degree of compatibility between the NPL environment and the Wang 2200/CS environment. This also provides a common access method across the various hardware/operating system platforms which are supported by NPL which allows the porting between NPL platforms to be done with little or no change to the application code.

The fundamental way in which information, both programs and data, is stored on disk for use with NPL is the diskimage.

NOTE: Niakwa's Data Manager is often a better solution for storing key-indexed data files.

An NPL diskimage is a logical disk device which may be equated with a physical native operating system file or device by use of the Device Equivalence Table.

The use of NPL diskimages on any machine does not interfere in any way with other data stored on a shared disk. All NPL disk operations are contained within the diskimage.

Every NPL diskimage may contain up to 4079 logical files. These may be either program or data files. Diskimages have a specialized internal format which is used to maintain information about these files within the disk. This format consists of an index area, a catalog area, and an optional non-cataloged area.

The Index contains the name and sector location of every cataloged program and data file on the diskimage. The Catalog Area is where each cataloged program file and data file is physically stored.

The Index is always stored at the beginning of each diskimage, starting at sector zero. Each sector of the Index contains 16 file entry slots. The first sector of the Index uses the first file entry to store the hash method, index size, current end of the Catalog Area, and the upper limit (END) of the Catalog Area. All other file entry slots are used to maintain information about individual files. Files listed in the Index are referred to as CATA-LOGED files.

Refer to Chapter 5 on using diskimage files and Chapter 7 on the technical details of NPL diskimages.

Native Operating System Files

Native operating system files are easily created and/or used by NPL programs. This flexibility is typically used by NPL programmers for generating files which can be used as input to native operating system applications or generating a file that can be printed at a later time (typically ASCII format).

ISAM Files

Through the use of the Niakwa Data Manager (NDM) NPL developers have access to state-of-the-art native ISAM products (i.e., Btrieve by Novell, C-ISAM by Informix, etc.) to store data while retaining full portability. This option allows NPL developers to share data files with popular third-party products that also make use of these industry-standard file formats, improve application performance, and increase programmer productivity. Refer to Section 1.2.2 for more information on Niakwa's NDM product.

1.5.3 Display Devices

As with other I/O operations, the NPL method of screen output for display devices is designed to be compatible across all machines on which NPL operates. However, because of the variety of devices available (controllers, monitors, terminals, consoles, etc.) this area of NPL support is very devices-specific. In general, NPL can address 80 columns and 24 rows, character output is used, the output is displayed at the current cursor position--unless explicitly specified, line wrapping is automatic, and screen scrolling is automatic. Additional features such as color, graphics, local terminal printing, and 132 column support are also available on some devices (refer to the appropriate NPL Operating-Specific Supplement for more information).

To provide this level of compatibility, Niakwa provides a series of generic files designed to allow different display devices to work similarly on all platforms. These include screen driver files that contain a built-in translation table between the devices, available characters, and NPL's character set (refer below), and font files to support the NPL character sets on the various display devices. All of these NPL-supplied device driver files and tables are modifiable by the NPL Utilities and easily customized to a developer's particular needs.

1.5.4 NPL Character Sets

NPL programs typically make use of the standard or alternate NPL character set or a developer modified version of this (as discussed above).

The NPL standard character set is as follows:

NPL Standard Characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1x	â	ê	î	ô	û	ä	ë	ï	ö	ü	à	è	ù	Ä	Ö	Ü
2x		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
6x	°	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	§	£	é	ç	¢
8x		♦	>	<	→		|	¨	’	‘	^	■	!!	↕	ß	¶
9x	â	ê	î	ô	û	ä	ë	ï	ö	ü	à	è	ù	Ä	Ö	Ü
Ax	_	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
Bx	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
Cx	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Dx	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
Ex	°	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
Fx	p	q	r	s	t	u	v	w	x	y	z	§	£	é	ç	¢

The NPL alternate character set is as follows:

NOTE: The lines surrounding characters in the range HEX(C0)-HEX(FF) are not part of the character. They are included to help distinguish the boundaries of each character.

NPL Alternate Characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1x	â	ê	î	ô	û	ä	ë	ï	ö	ü	à	è	ù	Ä	Ö	Ü
2x		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
6x	°	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	§	£	é	ç	¢
8x		♦	t	t	→	¬	|	¨	’	‘	^	■	!!	Ã↕	ß	¶
9x	•	◊	▲	▼	↓	¬	✓	°	{	}	∇					
Ax		-	-	─	|	┐	┌	⊥	|	┌	└	⊥				
Bx																
Cx																
Dx																
Ex																
Fx																

1.5.5 NPL Virtual Keyboard

In addition to the various display options available on different NPL operating environments, the keyboard layouts to these devices differ as well (like the screen driver files, the keyboard files can be modified by the developer with the use of the NPL utilities). Consequently, Niakwa has also provided a standard keyboard layout as shown below.

NOTE: This keyboard layout corresponds to the keyboard layout of the Wang 2200 terminals.

CHAPTER 2

RUNTIME OPERATIONS

2.1 Overview

This chapter discusses the operation of the NPL RunTime programs (RTP and RTI), the various methods of starting and exiting the RunTimes, command line (start-up) options, and the operation of the Immediate Mode facility. In addition, the RunTime's HELP processor and its REDIRECT option for remote support are detailed along with the RunTime's Print Control functions.

Section 2.2 outlines the technical differences between RTP and RTI.

Section 2.3 provides the general format for start-up of the RunTime program.

Section 2.4 discusses the use of various options available in the RunTime program.

Section 2.5 discusses the operation of the Immediate Mode facility.

Section 2.6 discusses the various methods of exiting the RunTime program.

Section 2.7 discusses the HELP processor and associated HELP options.

Section 2.8 discusses the Printer Control feature of the RunTime.

Section 2.9 discusses how the REDIRECT feature of the HELP Processor can be used for remote support.

Section 2.10 discusses how the RunTime program handles program errors.

2.2 RTP Versus RTI

Included in the RunTime Package are two RunTime programs. One program is non-interpretive (RTP), and the other is interpretive (RTI). The non-interpretive version allows for execution of NPL object code, without any capabilities for "Immediate Mode" command entry or functions (refer below). The interpretive RunTime program allows for full development capabilities, such as program text editing and debugging. The interpretive version requires more memory (refer to Chapter 3 or the appropriate operating environment addendum).

Feature	RTP	RTI
Object Code Execution	Yes	Yes
Development Capabilities	No	Yes
Requires ENABLED File	No	Yes
Extended Error Displays	No	Yes

2.2.1 Enabling RTI

Software vendors may choose whether or not end-user sites shall have the optional interpretive capabilities. This is done through the installation of the "ENABLED" file. This file is required for operation of the Interpretive RunTime program. Niakwa recognizes that while interpretive capabilities at the end-user site may be useful for support purposes, these same capabilities allow an end user access to the vendor’s source code, which may not always be desirable.

NOTE: The use of the Interpreter at end-user sites requires execution of the End-User Support Only License Agreement. Refer to the appropriate Operating System-Specific Supplement for more details on the use of the "ENABLED" file.

2.3 Starting the NPL RunTime

There are a variety of methods available for starting the execution of either the non-interpretive (RTP) or interpretive (RTI) RunTime program. However, before the RunTime program can be executed, the user must pass a security check. The RunTime uses a variety of security methods, which are dependent on the operating environment in use. Some versions may request that a Gold Key diskette be installed or be available for the security check. If the Gold Key diskette has been installed, there is no prompt for the Gold Key. If the Gold Key has not been installed, the diskette is requested. The RunTime program will not start without it. The discussion in this chapter assumes that the proper security requirements have been met for each particular operating system. Refer to Section 2.5 of the appropriate NPL Supplement for details on security considerations and implementation for the operating environment in use.

2.3.1 Starting from the Command Processor

The examples provided in this chapter use the IBM MS-DOS operating system as a default. Please refer to the appropriate NPL Supplement for detailed information concerning the operating system being used.

While at the MS-DOS prompt, the general form of a command line to execute either RunTime program is:

```
{RTP} [option] [progname]     (non-interpretive version)

{RTI} [option] [progname]     (interpretive   version  )

            where:

option    = RunTime startup option or combina-
            tion of startup options.

progname =  the filename of the NPL object
            program to be executed.  If no
            progname is specified, the Run-
            Time Program assumes the program
            "BOOT.OBJ" is to be executed. If
            an extension is not supplied, NPL
```

Refer to Section 2.4 for a detailed explanation of these options.

2.3.2 Starting from Batch/Script Files

Batch or script files can be written to execute the RunTime program directly. The batch or script file can perform the various activities necessary to invoke the RunTime program for a specific application in use. Refer to the appropriate NPL Supplement for more details on invoking the RunTime program from batch or script files.

2.3.3 Starting from a Menu

For a more user-friendly approach to starting an NPL application, the RunTime program can be started from a menu system. Many third-party products can be used for this purpose, including some operating systems' menuing facilities.

2.4 Command Line (Start-up) Options

The NPL RunTime environment is set up internally by the RunTime program on execution. A series of start-up options may be specified on execution of the RunTime that give the programmer a limited ability to modify the default RunTime environment. These options may be used to control a variety of RunTime execution features, however, many options are operating environment-specific. Refer to Section 4.4 of the appropriate NPL Supplement for information on which options are supported on the operating environment in use.

NOTE: The format of the command line (start-up) options described below can be either a slash "/" or a hyphen "-", depending on the operating system being used. Refer to the appropriate NPL Supplement for information on which is appropriate.

2.4.1 /B (Background Partition)

The "/B" RunTime option is used to inform the RunTime that it is operating in a background partition.

```
{RTP} [/B=nn]

{RTI} [/B=nn]
```

where:

```
nn =       the background partition number,
           where supported.
```

This option is not available on all versions of NPL (e.g., it is not supported under Novell NetWare). It is extremely operating-system dependent.

The "= nn" specification is not permitted on all operating environments.

Refer to the NPL appropriate Supplement for information regarding availability of this option.

2.4.2 /D (DET Entries)

The "/D" RunTime option specifies the number of device equivalence table (DET) entries defined. The number of DET entries may range from 16 to 255.

```
{RTP} [/B /T=xx]

{RTI} [/B /T=xx]

                where:

 xx =           a defined SuperDOS port number
```

The number of DET entries specified by "/D" is available to the application in byte 16 of $MACHINE in binary format.

If "/D" is not specified, the default of 16 DET entries is used.

Each DET entry above the default of 16 results in additional memory being used. The amount of memory used for DET entries varies widely among different NPL platforms. The programmer must consider the effects of this additional memory use. On systems with limited physical memory, use of the "/D" option to specify more than 16 DET entries may result in the user partition being reduced in size. On systems with paged or virtual memory, the higher amount of memory required by the user's task may cause swapping or paging, that will reduce system performance.

The programmer should also be aware that there may be limits imposed by the host operating system on the number of files that may be open concurrently.

NOTE: The use of /D, or even $DEVICE, in itself does not cause files to be opened. Rather, files are opened when first accessed.

The "/D" option is only valid for NPL Revision 3.0 or greater.

For more details on the "/D" option refer to $DEVICE in the NPL Statements Guide. Refer to Chapter 4 of the appropriate NPL Supplement for details on host operating system open file limits.

2.4.3 /G (Graphics Mode)

The "/G" option is used to invoke graphics mode on an EGA or Hercules controller on the IBM PC under supported operating systems. When invoked, the RunTime attempts to use available true box graphics and user-definable fonts on an IBM PC or compatible system. The "/G" may be immediately followed by a letter indicating the type of graphics display to use ("E" for EGA - "H" for HERCULES).

```
{RTP} [/G[H,E]]

{RTI} [/G[H,E]]
```

For example:

```
RTI /GE [progname]
```

uses the EGA graphics support,

while:

```
RTI /GH [progname]
```

uses the Hercules graphics support.

If no letter is used, then the RunTime attempts to determine if an appropriate adapter (EGA or Hercules) and support files, such as the font download file, are present. If neither adapter is available, the "/G" option is ignored. If the "/G" option is immediately followed by an "E", only an EGA adapter is used (if available). If the "/G" option is immediately followed by an "H", only a Hercules adapter is used (if available). This allows for the use of Hercules adapters on systems which have both EGA and Hercules displays (either on separate adapters or combination adapters).

The /G option is not available on all operating systems.

The /G option is not implemented on NPL revisions prior to 2.01.20.

For more details on the availability of graphics mode, refer to Chapter 4 and 6 of the appropriate NPL Supplement.

2.4.4 /H (Handle Table Size)

The handle table is an internal table used by the RunTime to translate two byte p-code pointers into four-byte memory addresses. Handle table entries are created at program resolution time. An entry is required for:

- Every unique variable
- Each unique line number
- Each DO/ENDDO group
- Each internal DEFFN'
- Each loop construct
- Each PROCEDURE or FUNCTION

It not specified, the RunTime program initially allocates a small amount of memory for the handle table and expands it as required. This can cause difficulties for applications since the routine for expanding the table requires some temporary memory to make a copy of the existing table. If there is not sufficient memory, an error A01- Memory Overflow or 200 - Handle table full / cannot expand could result. This problem particularly affects systems with fixed task sizes such as SuperDOS but could also affect very large applications under MS-DOS.

The /H option can be used to pre-allocate a minimum size for the handle table on all non 32-bit versions of NPL (MS-DOS, SuperDOS, MS-Windows and Intel Xenix). The /H option is used to specify the minimum size of the handle table in number of K entries.

For example:

```
RTP /H4
```

Causes the minimum size of the handle table to be set to 4K entries.

```
RTP /H16
```

Causes the minimum size of the handle table to be set to 16K entries (the maximum value of non 32-bit versions of NPL).

NOTE: The value specified in the /H option represents the number of entries, in kilobytes. Since each entry requires four bytes, the amount of memory allocated to the handle table, in kilobytes, is four times the value specified by the /H option. For example, /H4 causes 16K to be allocated to the handle table.

If /H is not specified, the RunTime continues to initially allocate only a very small handle table and expands it as needed.

```
{RTP} [/Hk]

{RTI} [/Hk]
```

where:

k = the number of entries, in kilobytes.

Developers who have encountered difficulty with A01 or 200 errors due to handle table reorganizations are encouraged to use the /H option to set the handle table size large enough to prevent reorganizations.

Byte 21 of $MACHINE contains the maximum number of entries allocated to the handle table (in K) during a RunTime session. This value is used to determine the proper value to use with the /H option described above. Developers who wish to use this feature are advised to execute every function module of the application before examining this value. This ensures that the value observed actually does represent the highest value possible with the application.

NOTE: The value returned in byte 21 of $MACHINE is in binary, but the value used for the /H option must be in decimal.

For 32-bit versions of NPL (Intel UNIX, and the 386/DOS-Extender), the /H is used to specify a maximum size for the handle table in number of K entries. This value may be up to 64 on 32-bit versions of NPL.

For example:

```
RTP /H32
```

allocates a handle table large enough for 32K entries.

This enhancement is useful for very large applications that may actually exceed 16K entries.

NOTE: Considerable memory can be used by the handle table if this option is used. Each handle table entry requires four bytes. Thus, for example, a specification of /H64 requires 256K for the handle table. As such, do not make this entry larger than required. Byte 21 of $MACHINE can be used to determine the size required as described above.

The /H option is only supported on NPL Revisions 3.20 or greater.

2.4.5 /K (Mouse Support)

On some operating environments, the /K option is used to explicitly request mouse support.

```
{RTP} [/K]

{RTI} [/K]
```

The /K option is required in environments where the NPL internal technique used to provide mouse support may not be 100% compatible with all hardware platforms or operating environment variations.

Refer to Section 7.6 on mouse support in NPL. Refer to Section 5.6 of the appropriate NPL Supplement for information on mouse support and the /K option on specific operating systems.

2.4.6 /L (Leave Overhead Memory)

The "/L" RunTime option specifies that the memory allocated by the RunTime should be contained within that task's own memory, and not use any available overhead memory.

```
{RTP} [/L]

{RTI} [/L]
```

This option is not available on all versions of NPL. It is extremely operating environment dependent.

Refer to Chapter 4 of the NPL Supplement(s) for information regarding RunTime version support of this option.

2.4.7 /M (XMS Memory)

NPL for standard MS-DOS environments supports the optional use of HMA memory. Use of HMA memory can result in up to 64K additional memory being available to NPL applications.

Refer to Chapter 4 of the MS-DOS Supplement for more information on the use of this option.

This start-up option requires Revision 3.20 or greater.

2.4.8 /P (Pre-boot)

The "/P" RunTime option allows for a "pre-boot" configuration program. If the /P option is specified on the RunTime command line, NPL loads and executes a "pre-boot" program.

The purpose of this program is to set programming preference options, (especially $OPTIONS values), and then perform a LOAD BOOT of the specified program. The BOOT program must be a different program than the pre-boot program. Other operations, such defining standard device addresses for library modules, could also be done here. However, operations in general should be restricted to setting values that are globally applicable.

```
     {RTP} [/Pfilename]

     {RTI}  [/Pfilename]
where:
     filename =  the filename of the pre-boot
                 program to be loaded.
```

The name of the pre-boot program may be specified in three ways:

1. Immediately after the /P option, with no space. For example:

   /PSTARTUP

2. If no filename immediately follows the /P option, and the new environment variable NIAKWA_PREBOOT defines the program to be used, this filename is used. If such a name is specified it should usually include a drive and directory.

3. If no filename immediately follows the /P option, and no environment variable NIAKWA_PREBOOT is defined, the filename PREBOOT.OBJ is used (current directory assumed).

In all cases, if the file has no extension, a ".OBJ" is assumed. The pre-boot program is loaded and executed only once. The pre-boot option is available on all NPL platforms, for both the interpretive and non-interpretive RunTime. Operating systems which do not support an environment may not use option (2) to specify the pre-boot file name.

The last statement the pre-boot program should execute is a LOAD BOOT statement (with no parameters). This will load in the boot file (if any) specified on the command line. Other statements which can affect the default boot program name (LOAD BOOT "name" or SAVE BOOT "name") should not be used by the pre-boot program.

For example:

```
0000 ;PREBOOT
 : DIM O$64,S$256,K$32+256+32+256
 : O$=$OPTIONS
 : STR(O$,38,1)=BIN(1)            : ;require all declarations
 : STR(O$,42,1)=OR HEX(01)        : ;turn off error beeping! : $OPTIONS=O$
 : S$=$SCREEN
 : STR(S$,VAL(HEX(5F))+1,1)=HEX(5F) : REM underline fix
 : $SCREEN=S$
 : K$=$KEYBOARD
 : STR(K$,1+32+VAL(HEX(5F)),1)=HEX(5F)  : ;underline generates HEX(5F)
 : STR(K$,1+VAL(HEX(5F))/8,1)=AND HEX(7F): ;not a special function key
 : $KEYBOARD=K$
 : LOAD BOOT                      : ;load specified boot program
```

WARNING--Exercise caution when testing a pre-boot program. If the pre-boot program is loaded (i.e., using LOAD BOOT) then typically RUNning the program will result in an infinite loop, since the LOAD BOOT at the end will reload the same pre-boot program. This can be stopped with the HALT key.

2.4.9 /R (Remote Control)

The /R option is used to force the RunTime program to use generic native operating system screen access techniques instead of direct video mapping. This feature is primarily intended to be used in conjunction with third party communication products for remote control maintenance.

```
{RTP} [/R]

{RTI} [/R]
```

Redirection is supported directly by the RunTime without requiring the use of the /R option. Refer to Section 2.9 for complete details on using the NPL REDIRECT feature.

2.4.10 /S (Separate Program Segments)

The /S option performs no operation. It is supported only for syntactical compatibility with prior revisions of NPL.

```
{RTP} [/S]

{RTI} [/S]
```

2.4.11 /T (Terminal/Port Number)

The "/T" RunTime option is used to inform the RunTime of the value to use for the terminal number. The specified value becomes the #TERM value of the partition. On some systems, in conjunction with the "/B" option, this option specifies the port which can control the background partition.

```
{RTP} /T=xx [progname]

{RTI} /T=xx [progname]
```

where:

```
xx =        a numeric-constant which indicates
            which terminal number to use (over-
            rides default terminal number de-
            termination).
```

This option is not available on all versions of NPL and is extremely operating environment-dependent.

Refer to the NPL Supplement(s) for information regarding availability of this option.

2.4.12 /U (UMB Memory)

NPL for standard MS-DOS environments only, NPL supports the optional use of XMS memory. Use of XMS memory can result in up to 200K additional memory being available to NPL applications.

Refer to Chapter 4 of the MS-DOS Supplement for more information on the use of this option.

This start-up option requires Revision 3.20 or greater.

2.4.13 /X (External Library)

The "/X" option is used to specify that an external subroutine library of functions is to be loaded. The filename for the external library must immediately follow the /X option. An extension of .QLB is assumed (.DLL if the MS-Windows version of the RunTime is being used).

```
{RTP} /X[quicklib]

{RTI} /X[quicklib]
```

where:

```
/X =       an external library must be at-
           tached to NPL and "quicklib" is
           the file name of the quick library
           module.  No spaces should appear
           between /X and the library name.
```

If the /X option is specified without a quick library name, NPL uses the default filename "RTPXSUBS.QLB" for MS-DOS or "RTPXSUBS.DLL" for MS-Windows

The "/X" option is only valid for revisions of the NPL RunTime greater than 3.0.

The method of loading external libraries is extremely operating system-dependent and "/X" may not be supported on all operating systems.

For more details on the implementation of external calls, refer to Chapter 16 of the Programmer's Guide and Chapter 11 of the appropriate NPL Supplement.

2.4.14 Boot Program

The RunTime program assumes that the first program to be executed is not in an NPL diskimage, but rather is a true native operating system file. This first program can be thought of as a "boot" program. Its job generally is to setup or customize the operators Device Equivalence Table and run a start-up program in a diskimage. That is, the "boot" program assigns device addresses to all NPL diskimage files that will be used by the various application programs about to be run. The last statement of this program would then "LOAD RUN" the desired starting program in one of the established diskimages.

The "boot" program does not have to reside in the user's current directory, however, it is a good idea to put it there. This would allow for relocating an entire application system to a new directory, if desired.

Niakwa has provided a sample BOOT.SRC and BOOT.OBJ program in the /BASIC2C directory. This program simply requests the operator to specify the application program name and disk address which is to be executed. This would typically be the "start" program for an application. The vendor may modify and recompile this program to suit a particular need or write a new one. Typical modifications would include:

- Establishing disk or printer addresses using the $DEVICE statement. Refer to the NPL Statements Guide, $DEVICE, for details.

- Automatically running the desired start-up program for an application without operator prompting.

NOTE: Any number of different boot programs can be set up. By executing the RunTime program with different progname options or from different current directories, various applications can be accessed independently.

For example, under MS-DOS an example Boot Program might appear as:

```
10  REM
20  REM SET-UP DEVICE EQUIVALENCE TABLE
30  REM FOR ACCOUNTS RECEIVABLE RUN
40  REM
50  $DEVICE(/D11)="\PROGS\ARPROGS.BS2"   :REM SETUP AR PROGS DISK
60  $DEVICE(/D12)="\DATA\ARMAST.BS2"     :REM SETUP MASTER FILE
70  $DEVICE(/215)="LPT1"                 :REM SETUP PARALLEL PRINTER
80  REM
90  REM RUN A/R Startup PROGRAM
100 REM
110 SELECT DISK/D11                      :REM SELECT AR PROGS DISK
120 LOAD RUN T "ARMENU"                  :REM RUN START PROGRAM
```

Line 50: Sets up device /D11 as the "ARPROGS.BS2" diskimage. Presumably this diskimage would contain all accounts receivables programs.

Line 60: Sets up device /D12 as the diskimage ARMAST.BS2. This diskimage could contain all data files related to the accounts receivable system. This diskimage is stored in a directory called \DATA.

Line 70: Sets up device /215 as the system printer.

Line 120: Loads program "ARMENU" for the accounts receivable application from the ARPROGS.BS2 diskimage.

This boot program would be stored in an MS-DOS directory (the \PROGS directory would be a reasonable choice in this case) with a name like ARBOOT.OBJ. To run it from the MS-DOS command processor, select the \PROGS directory (using the "cd" command) and enter:

```
RTP ARBOOT

    (or)

RTI ARBOOT
```

This loads and runs the RunTime program, which in turn scans the current directory (\PROGS) for the program ARBOOT.OBJ, loads it and runs it. The activities of the boot program as described above are then performed.

2.5 Immediate Mode Operation

Immediate Mode is the doorway to the entire NPL program development support system. That is, all NPL program editing and debugging tools are accessed from this mode. The Immediate Mode facility is available from the Interpretive RunTime only (RTI). The non-interpretive RunTime program (RTP) does not support Immediate Mode.

The term "Immediate Mode" is derived by the fact that commands or program lines which are entered in this mode are immediately parsed, compiled and, in some cases, executed as directed.

The availability of Immediate Mode is signaled when the ":" prompt is displayed during a session with the NPL Interpreter. The presence of the ":" prompt on the display means that processor control has been taken away from the NPL program and is given to the console (programmer). Here, Immediate Mode commands or program lines may be entered (facilitated by the NPL line editor, Section 2.5.3) to be immediately processed by the Immediate Mode processor.

With the Immediate Mode facility, variable values can be examined or modified at any time during program execution. Programs can be executed one statement at a time for pinpoint debugging and logic testing. Programs can be loaded, modified, and saved "on the fly", removing the burden of recompilation after even the smallest changes. Program line syntax errors are detected and reported immediately after each line is entered to maximize the programmer’s efficiency while creating new programs or editing existing programs.

2.5.1 Two Types of Immediate Mode Entries

There are two types of entries which can be made in Immediate Mode. The primary distinction between the two is that one type is immediately executed, the other is not. Other than the allowance for this distinction, the syntax of either type of entry is largely identical.

Immediate Mode Program Lines (Deferred Execution)

Any text entered in Immediate Mode which is prefixed by a number (up to five digits) is considered to be an NPL program line. Here, the program line text is immediately parsed and compiled into p-code, and syntax errors, if any, are reported. However, the program line itself is not executed. Rather it is stored in the NPL program text area in memory for later execution.

This branch of Immediate Mode generally relates to NPL program text editing. With this facility of Immediate Mode, the programmer has full access to the NPL program text currently in memory for editing functions such as inserting new program text, modifying existing text, deleting text and many other functions. To aid the programmer, the Immediate Mode processor at all times performs syntax checking and reporting after each program line is entered.

The program editing facility of Immediate Mode is discussed in detail in Chapter 5.

Immediate Mode Commands (Immediate Execution)

Any text entered in Immediate Mode which is not prefixed by a number is considered to be a "command". Like an Immediate Mode program line, command text is immediately parsed and compiled into p-code, and syntax errors if any are reported. However in addition, the command is immediately executed if no system errors are found.

The power of the Immediate Mode command is employed largely as a debugging tool. This is derived partly from the fact that virtually any statement of the NPL language can be entered as a command, thereby allowing for its immediate execution and the immediate availability of its result (with some exceptions, as documented in Section 2.5.3).

This is further enhanced in that the Immediate Mode command text does not disturb the NPL program currently in memory, since commands are entered, compiled and executed in a separate self-contained buffer from that of the resident NPL program.

With this facility, Immediate Mode commands can be used for variable inspection, modification, program listing, cross-referencing, etc., of the NPL program text and data variables currently defined in memory for the current module.

For example, entering the following command in Immediate Mode:

```
PRINT A,STR(B$,1,3),VAL(C$,3)*2^8R
```

would print the values of A, the first 3 bytes of B$, and the numeric value of C$ times 2 to the eighth power. If the variables A, B$ and C$ are already defined (by the resident and executed NPL program), then the current values of these variables are used in the computation for the final result. Since variable inspection in this manner also allows for computations, the programmer can easily manipulate data in a fashion which renders it more meaningful for debugging purposes.

This branch of Immediate Mode is discussed in full detail in Chapter 6, Debugging Code.

2.5.2 Invoking Immediate Mode

There are many different events within the NPL language which cause Immediate Mode to be invoked. However, generally, these events fall into four categories. Immediate Mode may be invoked:

1. From the keyboard (on demand) during real-time execution of a program.

2. Programmatically by placing certain statements within a program in advance which, when executed, cause Immediate Mode to be invoked.

3. Conditionally on the occurrence of certain pre-determined events.

4. Automatically when a RunTime error condition is encountered.

In all cases, invoking Immediate Mode involves taking control away from an executing NPL program and controlling the partition from the keyboard once in Immediate Mode.

Before invoking Immediate Mode, it is important to choose the method of invoking which best takes into account the program's situation once the Immediate Mode session is complete. This is because some methods of Immediate Mode invocation disturb the RETURN stack and CONTINUE status of the executing program, thereby potentially forfeiting the option of resuming its execution once the Immediate Mode session is complete.

The following is a discussion of each method of invoking Immediate Mode, together with some insight as to the circumstances under which each would typically be used.

Invoking Immediate Mode from the Keyboard

There are three methods of invoking Immediate Mode from the keyboard during the execution of an NPL program. These are the STEP and RESET functions of the HELP key, and the HALT key sequence.

With all three methods, execution of the resident program is halted and Immediate Mode is invoked (i.e., the colon prompt (":") is displayed). However, the availability of each method and the effect that each has on program continuation differ as follows:

STEP and RESET are both functions of the HELP display, and as such may only be accessed when the NPL program is waiting for keyboard input. Therefore, STEP and RESET may only be used to invoke Immediate Mode when an INPUT, LINPUT or KEYIN statement has been executed.

The HALT key, however, invokes Immediate Mode at any point during NPL program execution (under most operating systems). Therefore, the HALT key can be used to invoke Immediate Mode under circumstances where the operating program's logic is out of control (e.g., infinite loop) and is not requesting keyboard I/O which would allow control to be acquired from HELP.

The STEP function and HALT key methods of invoking Immediate Mode are most often used in program debugging, since both of these methods preserve the subroutine RETURN stack and CONTINUE status of the interrupted program. Therefore, this allows that normal program execution may be resumed from the point of interrupt once the Immediate Mode session is completed.

The RESET function, however, clears both the RETURN stack and the CONTINUE status of the executing program, thereby forfeiting the option of resuming normal execution of the interrupted program. The RESET function is generally used to recover control under circumstances where the executing program is "hung". A typical case of this is where I/O to an external device (like printer, disk drive, etc.) is not resulting in a response and the program logic cannot continue.

For further details on RESET and STEP, refer to the NPL Statements Guide .

Refer to the appropriate NPL Supplement for details on the HALT key sequence for the hardware and operating environment being used.

Invoking Immediate Mode from Program Control

Immediate mode may be invoked under program control by use of the STOP, STEP and END statements. At any time that one of these statements is encountered during the execution of a program, program processing is halted and the partition is placed in Immediate Mode.

Strategic placement of these statements in program logic allows for rapid progression through correct program logic to temporarily stop processing at areas where known bugs exist. This allows for detailed analysis of program logic and variables from Immediate Mode commands in only areas of the program code which are of interest.

The STOP and STEP statements are the most commonly used methods for programmatically invoking Immediate Mode. Both of these methods preserve the RETURN stack and CONTINUE status of the interrupted program, thereby allowing for normal resumption of the program once the Immediate Mode session is completed.

The END statement preserves the RETURN stack when executed, but does not preserve the CONTINUE status. Consequently, resumption of program execution may not be possible. For this reason, the END statement is most often used to "formally terminate" execution of a program or system.

There are other statements which may also cause invocation of Immediate Mode under program control, but with the additional attribute that invocation is conditional. These are discussed in the following section.

For further details on STOP, STEP and END, refer to the NPL Statements Guide.

Conditionally Invoking Immediate Mode

There are a number of methods under which the programmer may instruct the Interpreter to conditionally invoke Immediate Mode. That is, the Interpreter normally executes a given program(s) until such time as a certain condition or event takes place. When the condition or event takes place, program execution is halted and Immediate Mode is invoked.

These methods are useful in that they allow execution of program logic to take place at high speed until such time as a point of interest (for debugging purposes) is encountered.

For example, it is possible to instruct the Interpreter to resume program execution until such time as a certain variable is modified in a LET statement, or until a program line in a certain range of line numbers is encountered, or until a certain subroutine completes execution.

The following is a summary of the instructions which provide conditional invocation of Immediate Mode. Each of these instructions may be issued as Immediate Mode commands or as a statement in an NPL program. Further, in all cases the RETURN stack and CONTINUE status are preserved, thereby allowing for resumption of program execution from the point of interrupt.

CONTINUE	Continue execution until a program LOAD statement is encountered, at which point invoke Immediate Mode.

CONTINUE RETURN	Continue execution until execution of the current subroutine is completed, at which point invoke Immediate Mode.
CONTINUE NEXT	Continue execution until the current FOR/NEXT loop is completed, at which point invoke Immediate Mode.
STEP #	Invoke Immediate Mode when a program line number in the specified range is encountered (also places the program in STEP Mode).
TRACE V or # or ’	Invoke Immediate Mode when either one or a range of: variables, program line numbers, or marked subroutines are accessed.

For further details on CONTINUE LOAD/RETURN/NEXT, STEP # and TRACE V/#/’, refer to the NPL Statements Guide.

Invoking Immediate Mode on ERROR Conditions

When a RunTime error condition occurs during execution of an NPL program, the Interpreter displays debugging information and automatically invokes Immediate Mode. The exact sequence of events is as follows:

1. Program execution is halted.
2. The program statement currently executing at the time of the error is displayed.
3. A visual pointer is displayed at the approximate section of code in the statement which caused the error.
4. A descriptive error diagnostic is displayed.
5. Immediate Mode is invoked.

At this point, all Immediate Mode functions are available to the programmer to aid investigation of the error condition.

NOTE: The RETURN stack of the executing program is preserved when an error condition occurs, but the CONTINUE status is not. Therefore, to resume processing after an error condition has occurred, the error condition must first be corrected (if possible) and then the CONTINUE status must be reinstated (e.g., with an Immediate Mode GOTO command). For further details on resuming program execution from Immediate Mode, refer to Section 2.5.3.

Summary Table

The following is a summary of all methods of invoking Immediate Mode, and the effect of each on the RETURN stack and CONTINUE status.

METHODS OF INVOKING IMMEDIATE MODE			
Method	**From**	**Return Stack Preserved**	**CONTINUE Status Preserved**
STEP (HELP) key	Keyboard	Yes	Yes
RESET (HELP) key	Keyboard	No	No
HALT key	Keyboard	Yes	Yes
STOP statement	Program	Yes	Yes
STEP statement	Program	Yes	Yes
END statement	Program	Yes	No
CONTINUE LOAD	Conditional	Yes	Yes
CONTINUE RETURN	Conditional	Yes	Yes
CONTINUE NEXT	Conditional	Yes	Yes
STEP #	Conditional	Yes	Yes
TRACE V/#/’	Conditional	Yes	Yes
Error Condition	Automatic	Yes	No

2.5.3 Immediate Mode Operation

The intended capability of Immediate Mode operation as it relates to program editing and debugging is reserved for discussion in Chapter 5, Creating NPL Programs and the NPL Editor, and Chapter 6, Debugging Code.

This section addresses the symptomatic effects that Immediate Mode or an Immediate Mode session may have on the operation of certain statements and on the operation of a program which resides in memory.

Statements Which Are Illegal As Commands

Although the vast majority of statements in the NPL language are legal as Immediate Mode commands, there are some statements which are not legal in Immediate Mode.

The following statements are never legal as Immediate Mode commands:

```
DATA
DEFFN and DEFFN'
FUNCTION and PROCEDURE
```

Further, any statement which references a program line number is legal in Immediate Mode, but only if the program line exists in the currently loaded program, and only if the program is resolved (refer to RUN, Statements Guide).

For example:

```
:GOTO 2500                      :REM alters next instruction
                                     execution pointer
:PRINTUSING 1000,A,B,C          :REM print using an image defined
                                     in the program text area
:IF A=B*256+1 THEN 100          :REM alter next instruction
                                     execution pointer if
                                     condition true
```

If an attempt is made to execute commands which reference line numbers when the resident program is not resolved, the message "PROGRAM NOT RESOLVED" is issued. A program may be resolved, without executing it by typing the STEP command, followed by the RUN command.

Statements Which Operate Differently As Commands

There are a number of NPL statements which operate differently when executed as an Immediate Mode command as opposed to as a program statement. These differences are largely introduced as a convenience for debugging and generally make use of the statement more appropriate in Immediate Mode.

There are four classes of statements which are affected in this manner:

1. Transfer of Control Statements

Statements which cause a transfer of control (e.g., GOTO) operate differently in Immediate Mode than in a program line. In a program line, this class of statements immediately performs a transfer of control (i.e., jump). In Immediate Mode however, after the statement is entered, the system processes the statement (to register the transfer of control) but then performs an implicit HALT (which causes a display of the statement that will be branched to) and returns to Immediate Mode. To actually execute the jump, the EXECUTE key, or CONTINUE statement must be entered. This provides the programmer with the opportunity to verify a jump before it is executed.

Statements in this class are:

GOTO
RETURN
NEXT

2. Program LOAD Statements

Statements which LOAD program text operate differently as Immediate Mode commands. LOAD in a program line clears all program lines from memory (when line numbers are not specified), clears all non-common variables, then LOADs in the current RUN Module and automatically RUNs the loaded program. In Immediate Mode, LOAD does not clear the existing program or variables from memory. The specified program is merged with existing program text in the current LIST Module. Further, the resultant program is not automatically RUN. An explicit RUN statement is required.

These differences are implemented primarily to facilitate program text merging and editing.

Statements in this class are:

LOAD [DC]
LOAD DA
LOAD BOOT

NOTE: The LOAD RUN command is an exception to this rule. It operates identically both as a program line and as a command.

3. PRINT Statements

PRINT statements operate differently as Immediate Mode commands. All PRINT commands (entered in Immediate Mode) print to the screen irrespective of the current SELECT PRINT address. This facilitates program debugging by allowing inspection of program variable values from Immediate Mode even while programmed output is directed to a printer or spool file.

Statements in this class are:

PRINT
PRINTUSING

4. $SHELL Statement

The syntax of the abbreviated form (!) of the $SHELL command is different when issued in Immediate Mode. Specifically, enclosing quotes about the command argument are not required (in program mode they are mandatory).

Preserving a Program's "Execution Status"

Immediate Mode is often entered with the intention that program execution be resumed from the point at which it was interrupted once the Immediate Mode session has been completed. There are, however, certain Immediate Mode commands which disturb the execution status of the interrupted program. When it is intended that a program be resumed, the programmer must take care to avoid, or at least be aware of, their effect.

To preserve the execution status of an interrupted program while in Immediate Mode:

- Immediate Mode must be originally invoked using a method which preserves the subroutine RETURN stack and the program CONTINUE status of the interrupted program. For further details, refer to Section 2.5.2.

- Use of commands which clear the RETURN stack or CONTINUE status of the interrupted program should be avoided. Commands which cause this are:

CLEAR [P or V]	Clear program or variables
RENUMBER	Renumber program lines
RESET (help function)	Reset the partition
LOAD	Merge other program text

- Further, editing of the interrupted program text of any kind while in Immediate Mode (i.e., text additions, deletions or changes) should be avoided as this forfeits the ability to resume the program.

NOTE: These events only indicate circumstances under which a program may not be resumed from the point at which it was interrupted. They do not imply that execution of the remaining program cannot be restarted from its beginning (or some other predetermined point).

There are other Immediate Mode commands which, although they do not clear or even modify the RETURN stack or CONTINUE status of the program, may alter the normal logic path of the program when execution is resumed.

An example of this is the contents of a variable being modified by the programmer in an Immediate Mode assignment. This external influence on the variable's contents may cause operations or logic to take place within the program which otherwise would not occur.

Another example of this is the issuing of a GOTO command from Immediate Mode. This could bypass code which would otherwise be executed in normal program logic.

NOTE: Most often, actions such as these are intended by the programmer and are a useful debugging tool when used properly.

2.5.4 Exiting Immediate Mode

There are several methods by which Immediate Mode may be exited when an Immediate Mode session has been completed.

CONTINUE Command

The CONTINUE command (either as entered on the command line or selected from the HELP display) terminates Immediate Mode and resumes program execution from the point at which the program was interrupted.

NOTE: If the CONTINUE status of the program has been modified (e.g., by a GOTO), then execution resumes from that point. If the CONTINUE status of a program has been cleared (for further details refer to Section 2.5.2), then the CONTINUE command is inoperable.

Use of CONTINUE implicitly turns STEP Mode off. If TRACE V, TRACE # or TRACE ' is in effect at the time CONTINUE is entered, exit from immediate mode may only be temporary (for further details on TRACE statement, refer to the NPL Statements Guide).

EXECUTE Key

The EXECUTE key exits Immediate Mode and resumes program execution in the same manner as the CONTINUE command, with one exception. If the program has been placed in STEP Mode, then execution resumes to execute only the next single instruction of the program, after which Immediate Mode is reentered.

This is intended as a convenient method of single-instruction-stepping through program execution.

The EXECUTE key works this way only when the command lines in Immediate Mode is empty and the cursor is at column 1.

RUN Command

The RUN command exits Immediate Mode and starts execution of the program currently in memory. The RUN command always causes resolution of the program before execution. That is, the RETURN stack is cleared and the entire program is scanned for syntax errors.

If no line number is specified in the RUN command, all non-common variables are cleared and execution proceeds at the first line in the program.

If a line number is specified, all currently defined variables remain intact and execution proceeds at the specified line number.

The RUN command does not allow for resumption of a program indicated by the current RETURN stack or CONTINUE status. Execution always begins from the point specified by the command itself.

$END Command

The $END command exits Immediate Mode and also terminates the NPL session by returning to the host operating system. All program text and data are cleared from memory by this command.

2.6 Exiting NPL

There are several methods available to either the programmer or end user for exiting from the RunTime program (and consequently the application it is executing). In most cases, exiting the RunTime returns to the point at which it was invoked. That is, if the MS-DOS command processor was the starting point, exiting returns there. If a menu was the starting point, exiting returns to the same menu. A brief discussion of the various exiting methods follows.

2.6.1 Exiting under Program Control

The RunTime program may be exited under program control by any one of the following events:

- When working in the interpretive RunTime (RTI), execution of the NPL "END" or "STOP" statement. With either of these statements, a ":" is placed on the screen with the cursor immediately following.

  When in the non-interpretive RunTime (RTP), pressing the CANCEL or HELP key with the KILL NPL option at that point exits the RunTime program (refer to Section 2.7 for more details).

- By program logic which falls through the logical end of the program. In this event, the RunTime is automatically exited without further action required. This is only true for the non-interpretive (RTP) RunTime.

- The non-interpretive RunTime (RTP) also exits if an application error condition is encountered.

- By execution of a $END statement. This statement causes the RunTime to end operation, returning control to the native operating system. This is the preferred method of exiting under program control. Refer to the Statements Guide, $END for more details.

2.6.2 Exiting Using the HELP Key

The end-user may call for cancellation of program execution using the HELP key. Pressing the HELP key suspends current program execution. The screen is saved and the HELP screen is displayed, with various options. The end user may at this point select the KILL NPL option to terminate program execution unless this option has been disabled by the application. If the LEAVE HELP option is selected the screen is restored and program execution resumes.

NOTE: The HELP key is active only when the application program is polling for keyboard input (e.g., executing a KEYIN, INPUT or LINPUT). For full details on the HELP key, refer to Section 2.7.

2.6.3 Exiting Using the Interrupt Key

All operating systems provide their own methods for terminating program execution. The interrupt key (e.g., MS-DOS--pressing CTRL-C) termination method is one of these. When the interrupt key is pressed, the usual HELP display is provided, and program execution may be terminated or resumed using the same methods as described for the HELP key.

NOTE: The interrupt key only takes effect when the application program issues a disk I/O request (e.g., DATALOAD, DATASAVE, LOAD DC, etc.). If the interrupt key is pressed when the application is waiting for keyboard input, it is treated as a normal keystroke.

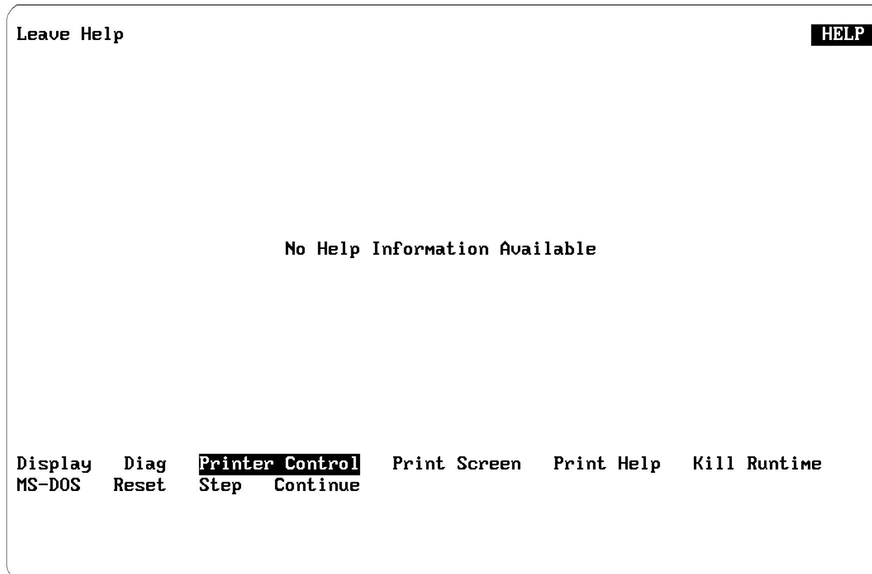
The interrupt key for MS-DOS is the CTRL-C combination, or CTRL-BREAK (sometimes labeled SCROLL LOCK).

NOTE: The interrupt key can be disabled on some operating systems. Refer to the appropriate NPL Supplement for more details.

2.7 The HELP Processor

The HELP processor can be accessed by pressing the HELP key. The HELP key is enabled whenever the RunTime is waiting for a key from the keyboard. Upon depression, application program execution is suspended, the contents of the screen are saved, and the following is displayed:

The Help Display under the interpretive RunTime (RTI) appears as:



At the top right hand corner of the display, the word HELP is displayed. This lets users know they are in the HELP processor (as opposed to the ERROR processor, which looks similar). Notice the series of highlighted phrases on the display, LEAVE HELP, DISPLAY, DIAG, PRINTER CONTROL, PRINT SCREEN, PRINT HELP, KILL NPL, O/S, RESET, STEP and CONTINUE. These are the HELP options and they are available to the user on all HELP screens.

NOTE: The RESET, STEP and CONTINUE options are not available under the non-interpretive RunTime (RTP). The use of each is discussed below.

The user may cancel HELP processing and return to the application program at any point by pressing the CANCEL key, or by selecting the LEAVE HELP or CONTINUE (not available under RTP) options. Upon depression of either of these keys, the application screen is restored and program execution resumes.

2.7.1 Selecting HELP Options

A HELP option is selected by moving the input block to the desired option and pressing EXECUTE.

NOTE: $OPTIONS byte 46 can be used to treat the RETURN key as if it were the EXECUTE key. Developers can enable this feature be setting byte 46 of $OPTIONS to HEX(01).

The input block is controlled by any of the following keys.

RETURN, TAB & EAST	Moves to the next option on the screen (row order).
BACKSPACE & WEST	Moves to the previous option on the screen (row order).
NORTH	Moves to the previous option on the screen (column order).
SOUTH	Moves to the next option on the screen (column order).
BACKTAB SHIFT-TAB	Moves to the previous option on the screen (row order)
PREV	Page backward through help entries with greater than 19 lines of text
NEXT	Page forward through help entries with greater than 19 lines of text.

Letter keys move the acceptance block to the next highlighted option with the first letter matching the letter keyed in.

All movement keys "wrap" when the top or sides of the screen are encountered.

The HELP processor automatically supports the use of a mouse and all mouse functions, where supported by NPL. Refer to Section 7.6 for a discussion on NPL mouse support.

2.7.2 HELP Options

LEAVE HELP	Selection of this option cancels HELP processing and continues program execution. Upon selection, the application screen is restored and program execution resumes. This is the default HELP option.

DISPLAY	The DISPLAY option displays the application screen as it was before HELP was executed. This is useful when reviewing User Defined HELP Screens (refer to text below).
DIAG	The DIAG option allows the user to invoke the REDIRECT feature to transfer system control to a remote terminal for an on-line troubleshooting session or Enable or Disable Keyboard Logging. The REDIRECT function allows the user to transfer control of their system to a remote terminal. For a complete discussion of the REDIRECT option, refer to Section 2.9. The Enable/Disable Keyboard Logging feature allows the user to turn on and off the Keyboard Logging feature of NPL. Display of this feature can be suppressed using byte 33 of $OPTIONS. Refer to the Statements Guide, $OPTIONS for details. Refer to Chapter 12 of the Programmer's Guide, and the Statements Guide, SELECT LOG, for a detailed discussion of Keyboard Logging.
PRINTER CONTROL	Selection of this option invokes the printer control options screen. Refer to Section 2.8 below for details on printer control.
PRINT SCREEN	Selection of this option causes the contents of the application screen to be printed on the system printer.
PRINT HELP	Selection of this option causes the contents of the HELP screen to be printed on the system printer.
KILL NPL	The KILL NPL option terminates execution of the application program and returns to the native operating system.
O/S	Selection of this option effects a temporary exit from the application program to the native operating system. Upon return, control is passed back to the HELP processor.

NOTE: The text displayed on the HELP PROCESSOR screen is the name of the native operating system (e.g., MS-DOS, SuperDOS, Xenix, etc.).

RESET	Selection of this option invokes Immediate Mode and prevents continuation of the program currently in memory. This may be useful for viewing a program listing, or inspecting the contents of one or more program variables. Refer to the Section 2.5.2 for more details on RESET.
STEP	Selection of this option invokes Immediate Mode following completion of the NPL statement being executed within the program currently in memory. This may be useful for then "stepping" through the program one statement at a time, for debugging program logic or variable inspection. Refer to Chapter 6 of the NPL Programmer's Guide for details on STEP mode.
CONTINUE	This option appears on the HELP display only after returning to HELP after previously selecting the STEP option. Selection of this option cancels HELP processing and continues program execution. Upon selection, the application screen is restored and program execution resumes.

NOTE: The RESET, STEP and CONTINUE options are only available under the interpretive RunTime (RTI).

2.7.3 Disabling Immediate Mode Options

From the HELP display under the Interpretive RunTime (RTI), the programmer can enter Immediate Mode processing, with development and debugging commands such as RESET, STEP and CONTINUE available for use. In addition, under both the interpretive (RTI) and non-interpretive (RTP) RunTimes, the O/S , DIAG and KILL NPL options appear on the HELP display by default.

All of the above options can be suppressed under program control by setting byte 12 and/or byte 33 of the $OPTIONS system variable appropriately. Refer to the NPL Statements Guide, $OPTIONS for details on the exact syntax and use of the $OPTIONS variable.

2.8 Print Control

The print control functions allow various printer functions to be easily modified by the operator at program execution time. The intention is to allow the use of printers which have very limited front panels but which can be programmed for different characters per inch, lines per inch, or fonts by sending proper control sequences.

Through the print control feature, characters per inch (pitch), lines per inch, form feed, line feed, and other printer control codes can be sent to the system printer.

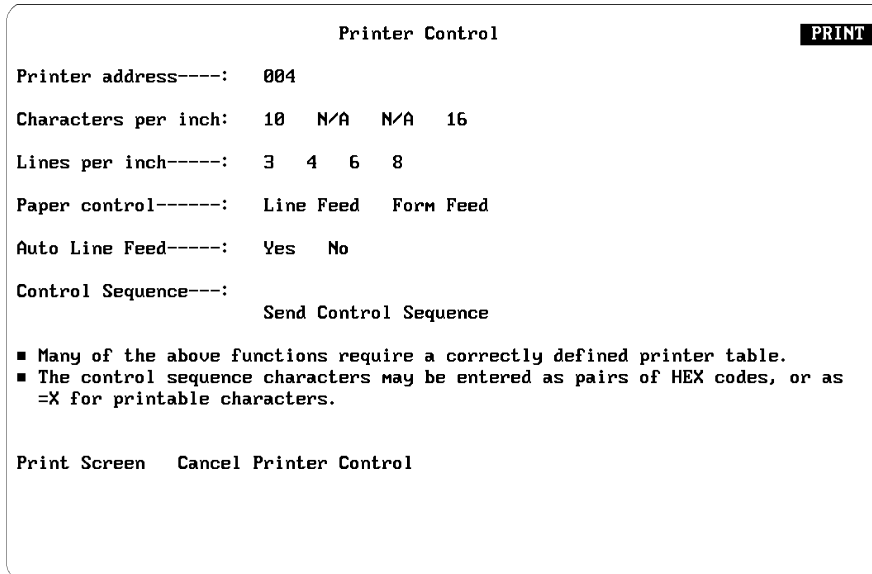
2.8.1 Accessing Printer Control

Printer control is a function of the HELP processor and may be accessed whenever the HELP processor is active. Refer to Section 2.7 above for details on the HELP processor.

2.8.2 The Print Control Screen

When the Printer Control function is accessed, a screen similar to the following is displayed.

NOTE: Refer to Section 2.8.4 below for an explanation of default values.


Printer Control PRINT

Printer address----: 004

Characters per inch: 10 N/A N/A 16

Lines per inch-----: 3 4 6 8

Paper control------: Line Feed Form Feed

Auto Line Feed-----: Yes No

Control Sequence---:
Send Control Sequence

■ Many of the above functions require a correctly defined printer table.
■ The control sequence characters may be entered as pairs of HEX codes, or as =X for printable characters.

Print Screen Cancel Printer Control

Printer control may be exited at any time by pressing CANCEL or by selecting the CANCEL PRINTER CONTROL option.

Options on the printer control screen may be selected simply by moving the highlighted acceptance block to the option desired and pressing EXECUTE. Refer to Section 2.7 above for detailed instructions on how to move the acceptance block.

2.8.3 Printer Control Options

The printer options which can be set are:

PRINTER ADDRESS — This option directs Help Processor and Print Control Output to the specified NPL print address. This must be a configured NPL device address such as 204 or 215. The initial value of this field at boot time is 004, but can be changed at any time for the duration of the current RunTime session.

CHARACTERS PER INCH	This option allows the pitch (number of characters printed per inch) to be changed. By moving the acceptance block to the desired characters per inch and pressing EXECUTE, the pitch is set as specified.

NOTE: Not all printers have the ability to print at the different pitches. Refer to Section 2.8.4 below for details on default values.

LINES PER INCH	This option allows the vertical spacing of reports to be modified. Simply move the acceptance block number to the desired lines per inch and press EXECUTE.

NOTE: Not all printers have the ability to print at the different values for lines per inch. Refer to Section 2.8.4 for details on default values.

PAPER CONTROL	This option advances the paper in the printer. LINE FEED advances the paper one line each time it is executed. FORM FEED advances the paper to the top of the next page each time it is executed. This option is not dependent upon a printer driver.
AUTO LINE FEED	This option tells the RunTime program whether or not to perform special end-of-line handling for the specified print address. Refer to Section 7.8.9 for further information on special end-of-line handling.

CONTROL SEQUENCE

Many printers have sophisticated functions which can be controlled by certain control sequences. This option is used to enter the control sequences to be sent to the printer. These control sequence characters may be entered in pairs of hex codes or as "= x" where "x" is a printable character. The "=" as a prefix indicates that the character(s) following are normal printable characters, not hex codes.

For example, the Okidata model 2410 printer has a correspondence quality mode. The hex codes 1B37 cause the 2410 to print in correspondence quality. The hex codes 1B38 cause the 2410 to print in draft mode. To activate correspondence quality, simply enter 1B37 on the control sequence line.

If EXECUTE or HELP is pressed while the acceptance block is at CONTROL SEQUENCE, HELP information from the file CTRLSEQ.HLP is displayed, if available. This HELP information should be set up by the application software vendor and should contain a description of the features of the printer being used with the corresponding hex codes to be used for control sequences.

SEND CONTROL SEQUENCE

This function, when executed, sends the control sequence entered above to the printer.

2.8.4 Print Control Values

It is clear that no standards exist for printer control protocol. Consequently, printer control specifications vary dramatically from printer to printer. The print control feature of the RunTime program has built-in defaults. These default control codes vary by operating system. Refer to Section 4.5 of the appropriate NPL Supplement for default printer control codes on specific operating systems.

When the RunTime program is invoked, it always looks for a table of printer control values under the name "PRINTCTL.TBL" (first in the current directory, then in the default BASIC2C directory). If no such table is found, the built-in defaults are used.

The Edit Printer Control Codes program allows the user to define up to 4 values for "characters per inch" and 4 values for "lines per inch" for the printer. In addition, the sequences used to perform Line Feed and Form Feed operations may be defined.

If these values are saved in the file "PRINTCTL.TBL" in the current directory or the default BASIC2C directory, the next time the RunTime program is invoked these values will be loaded for use by the Printer Control screen.

2.9 Redirect Feature for Remote Support

The REDIRECT option of the HELP Processor assists in support of end-user systems, by allowing "on the fly" turnover of system control.

There are essentially four steps in the remote support of an end user using the REDIRECT option:

1. Establishing the communications link.

2. Turning control over to the remote terminal.

3. Troubleshooting from the remote site.

4. Relinquishing control back to the end-user site for continuation of processing.

Each of these steps is discussed in detail in the following sections.

2.9.1 Establishing the Communications Link

Once it is decided that remote support is necessary, and both vendor and end-user sites have compatible telecommunications equipment (i.e., a modem), the communications link should be established.

To establish the communications link the user must ensure that the modem is properly connected to the system and to the telephone line. The user should also know the following information:

Terminal Type	Refer to Section 2.9.2 for supported terminal types.

System name of TC port	Device name of the port used to establish communication link (i.e., under MS-DOS either COM1 or COM2).
Baud Rate	1200 baud is recommended. If using higher baud rates, the system requires that the terminal and the driver have a compatible flow control (i.e., XON/XOFF) and have sufficient buffering to avoid loss of data.
Parity	This is necessary since redirecting to a nonexistent or improperly set-up device (e.g., baud rate or parity incompatibility) causes the system to "hang".
Stop Bits	Usually set to 1.

Once the end users know this information, they should set their communications ports using the appropriate operating system specific commands. The vendor should then establish the communications link in the normal way (i.e., dial the user's number, perform the necessary steps to establish the carrier signal). Once the communications link is established, it is a good idea to test the link prior to redirecting.

NOTE: The discussion above refers to use of a serial terminal as the remote device. Use of a PC as the remote device requires that the PC be executing a terminal emulation program. Niakwa does not support the use of terminal emulation programs. However, periodically Niakwa may review terminal emulation programs in the Niakwa Newsletter.

2.9.2 Turning Control Over to the Remote Terminal

Once the connection has been established, the user releases control of the RunTime to the remote terminal by entering the HELP processor, selecting DIAG, and then selecting REDIRECT (a sub-item of DIAG). The REDIRECT screen contains user-modifiable entries for the device name to which the RunTime's screen and keyboard I/O are to be switched (e.g., COM1), and the type of terminal to be used at the remote site.

As of this release the following terminals are supported:

Terminal Name	Terminal Type
ALTOS III	altos
ALTOS V	al5
Bull HDS I	wy50
Bull HDS III	vt100
DEC VT100	vt100
DEC VT200	vt200
IBM 3151	i3151
PC Monitor with ANSI.SYS Driver	IBMPC
PC Monitor function as a console under UNIX or Xenix	IMON
NCR 4970	vt200
Spectrix SPX 701	spx
Wang 2110a	w2110a
Wang 2x36 DE/DW	w2236
Wyse 50	wy50
Wyse 60, 150, 160	wy60
Wyse 370	wy370

NOTE: Use the exact terminal type name given above. An example of the screen is shown below:

```
                              Redirect Console                    REDIRECT

Redirect Device Name: COM1

       Terminal Type: IBMPC

Please Note:
■ Redirection passes control of your system to another device,
  normally to allow remote diagnosis of a problem by your software vendor.
■ The device must be ready before proceeding.









Proceed   Cancel Redirect
```

This example shows the default values for an IBM PC.

The user must ensure that the appropriate values are entered in the "Redirect Device Name" and the "Terminal Type" fields. The default values in these fields are rarely correct.

NOTE: If the HELP key is pressed when in the "Redirect Device Name" field, the system performs a normal HELP look up for the $HELP value "REDDEVIC". If the HELP key is pressed when in the "Terminal Type" field, the system performs a normal HELP lookup for the $HELP value "REDTTYPE". If this feature is to be used extensively with the application, the user may wish to provide .HLP files which specify typical values to use in these fields. The Redirect Device Name is typically the name that would be given to the port if it were used as a serial printer under NPL. Refer to Chapter 13 for detailed information on the use of HELP files.

Once the correct values have been entered, the user then must move the acceptance block to PROCEED and press the EXECUTE key. The RunTime validates the terminal type and device name (as far as possible), and then switches subsequent screen and keyboard I/O control to the specified device. At this point, assuming that the device baud rate, parity, stop bits, terminal type, and port are all properly set, the REDIRECT display should appear on the vendor’s terminal screen at the remote site.

2.9.3 Example of Redirect Feature

Listed below are the steps necessary to use the Redirect option. This example assumes that the end-user site has an IBM PC and the remote support site is using a VT100 terminal running at 1200 baud, 8 data bits, 1 stop bit and no parity.

1. Ensure that the communications port is functioning with the same communication parameters as the VT100. On the IBM PC under MS-DOS this can be accomplished by use of the "MODE" command shown below:

   MODE COM1:1200,N,8,1

   where:

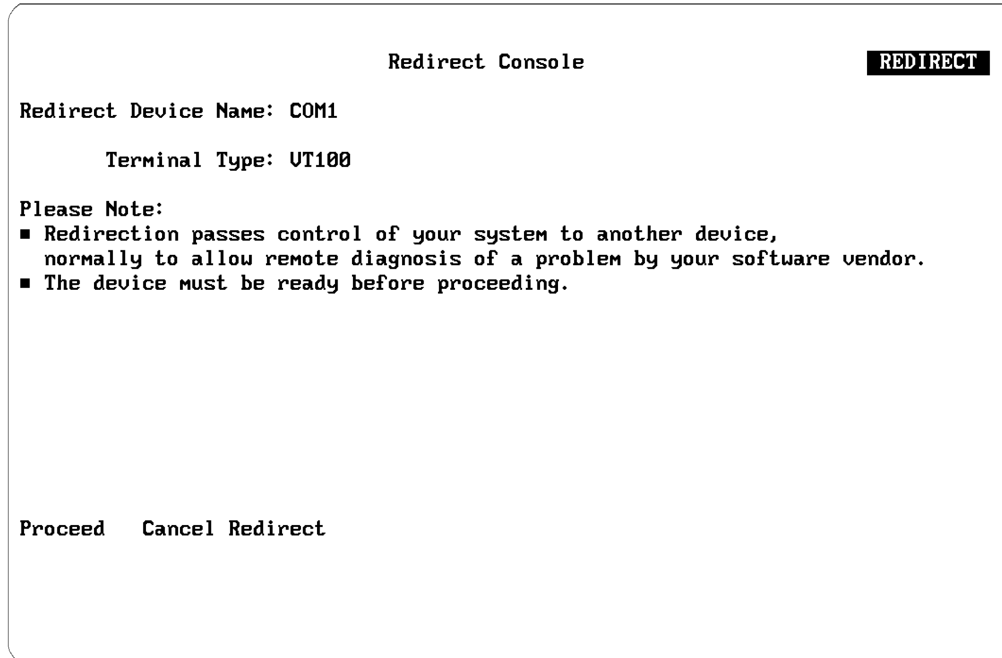
COM1	indicates which serial port to set
1200	indicates baud rate
N	indicates no parity
8	indicates 8 data bits
1	indicates 1 stop bit

2. Make sure that the modems at the remote support site and the end-user site are functioning properly. This can be accomplished by the "ECHO" command shown below:

   ECHO "HELLO" > COM1

   This should print the word "HELLO" on the VT100 terminal.

3. After executing the RunTime, enter the HELP Processor, select the DIAG option, and then select the REDIRECT option (a sub-item of DIAG). The end user should then enter the Redirect Device Name and Terminal Type. For this example, the screen should appear as follows:



The end-user should then execute the Proceed option. This turns over control to the remote sites VT100 terminal.

2.9.4 Troubleshooting from Remote Sites

Once the REDIRECT display appears on the remote sites terminal, the user may leave the redirect display as if the remote site were the actual monitor/keyboard on the system. Troubleshooting can then commence, providing the following restrictions are kept in mind:

- Communications parameters cannot be changed while connected. These parameters should be set prior to linking and should be used for the entire session.

- The appropriate screen and keyboard tables must be available on the host system for the remote terminal in use.

- The HALT key may not be supported on the remote terminal in use. Care should be taken to avoid processing programs which require halting before completion, but do not poll the keyboard. Halting can be accomplished by using the STEP option available on the HELP screen. In some cases, if the HALT key does not respond, pressing the HALT key at the original monitor may cause a HALT to be executed.

- Use of $SHELL or Invoke (!) should be done with caution. Some programs (especially under MS-DOS) may not operate properly with output redirected to a device.

- Exiting the RunTime for any reason (using $END or KILL NPL on HELP screen) automatically returns control to the host (IBM PC) system keyboard.

- If the attached terminal supports a local printer using an LCL= Y type $DEVICE statement, this can be used to get printouts. Please refer to Appendix D for information on terminals that support local printers. Refer to Section 7.8.8 for implementation of the local printing convention (LCL= Y).

2.9.5 Relinquishing Control Back to the End-User Site

The easiest (and recommended) way of returning control back to the end-user is to exit the RunTime. If this is inconvenient (e.g., because of a long process which must be completed or cause recovery of files), it is possible on most systems to transfer control back to the end-user "on the fly".

Returning control back to the end user "on the fly" requires essentially the same steps (only in reverse). First, users at the remote site should select themselves back to the Redirect command in the DIAG section of the HELP screen. Then they should select the following options:

Redirect Device Name :	CON
Terminal Type :	IBMPC

Since the communication link was previously established, the transfer of control back to the Host uses that same link. By selecting EXECUTE, control of the host computers screen and keyboard I/O is passed back to the host computer.

The following consideration should also be noted.

IBM PC compatible systems must be configured with the ANSI.SYS driver to regain control in this way. The type used for the normal host screen and keyboard on IBM systems is "ibmpc". Here, the user the equivalent of running the RunTime with the /R option. This means that some capabilities will be lost (e.g., faster speed, box graphics, downloadable fonts), and normal requirements for operation under /R must be met (e.g., ANSI.SYS driver configured). Because of these restrictions it is often more suitable to return control back to the user by exiting the RunTime Package, rather than redirecting back. Refer to Section 2.4.6 for details on invoking the RunTime with the "/R" option.

2.10 RunTime Errors

When application program errors are detected by the RunTime program, one of two possible error handling procedures is used, depending upon whether the interpretive or non-interpretive RunTime version is in use. The discussion of how each version handles errors follows.

2.10.1 Error Handling Under RTP

Under the non-interpretive version of NPL, when the RunTime program encounters a RunTime error, the following screen is displayed:

```
                                                                    ERROR
       Application Program Error Detected by Niakwa RunTime Package

Niakwa Error Code = P48 on line 0020: offset 0010, SI=0010, IP=2712.
MS-DOS Error code = 0002.  File not found.
PLATTER.BS2
Illegal Device Specification.
Program Load Sequence:MIKE









Display   Diag   Printer Control   Print Screen   Print Error   Kill Runtime
```

The contents of the fields displayed on the error screen are as follows:

RunTime error #	This is the RunTime error code. For ease of use, it is equivalent to the error codes as produced on the Wang 2200 system.
On line #	This is the line number of the original source program that caused the error condition.
Offset	This is the offset of the sub-statement in number of bytes from the first byte of the line number. This is useful in determining the exact statement in a multi-statement line on which the error occurred.
SI & IP	These fields refer to internal RunTime program routines and are intended for use by Niakwa only.
O/S error code	The operating system error code. Refer to the operating system documentation for a detailed listing of error codes.

NOTE: The error code's meaning, in this example "File not found", is supplied in the RTIx-ERR.HLP file and appears following the error code, provided that this file is in the default NPL directory. Refer to Appendix B and Section 2.10.4 below for information on Error Code files.

Message line	Wherever possible, the RunTime program provides further diagnostic information about the run time error on this line. This verbiage is supplied in the ERRORMSG.HLP file. In this example, this line states "Illegal device specification". Refer to Appendix B and Section 2.10.4 below for information on Error Code files.
Program Load Sequence	This field lists the names of the first program in memory, plus the last 5 programs overlaid at the time of the error.

NOTE: Releases prior to 2.01.18 displayed the names of the last program overlayed in memory, plus the first 5 programs overlayed at the time of the error.

User-defined information	The error screen displays the contents of the user-defined HELP file called RUNERROR.HLP in the current directory, if present. This file may created or modified by the programmer. It is expected that this file contains general information about what to do in case of a program error (e.g., whom to call). Message screens may be nested like HELP screens. Refer to Section 11.4 for details on nesting.

NOTE: The RUNERROR.HLP file must be limited to 8 lines of text. Exceeding 8 lines causes error information to scroll off the screen and be lost.

The options (DISPLAY, DUMP, PRINTER CONTROL, PRINT SCREEN, PRINT ERROR, and KILL NPL) can be accessed and executed on the ERROR screen as in a HELP screen (refer to Section 2.7.2 - HELP options) with the exception that the LEAVE HELP option is not available.

2.10.2 Error Handling Under RTI

Because of the nature of an Immediate Mode environment, with all development and debugging capabilities available, all program error displays under the Interpreter are "programmer friendly". This means that the somewhat limited error screen generated by the non-interpretive version of the RunTime is not used. Instead, the entire program line of code where the error occurred is displayed, along with the appropriate error code, and an arrow pointing to the approximate position of the actual error.

For example, assume that the following short program is being executed:

```
10  A$="ABC"
20  CONVERT A$ TO X
```

This program generates an execution phase error because the CONVERT statement requires that the contents of the alpha variable being CONVERTed contain only numbers. The error display looks like:

```
20 CONVERT A$ TO X
       ^
ERR X75: Illegal number
```

At this point, all Immediate Mode aids for debugging or variable inspection and modification are at the disposal of the user, such as full Immediate Mode, HALT/STEP, LIST, TRACE, EDIT/RECALL and more.

Refer to Chapter 5 of the Programmer's Guide for more details on error handling. Refer to the Statements Guide, HALT/STEP, LIST, TRACE, EDIT/RECALL for syntax and usage of these commands.

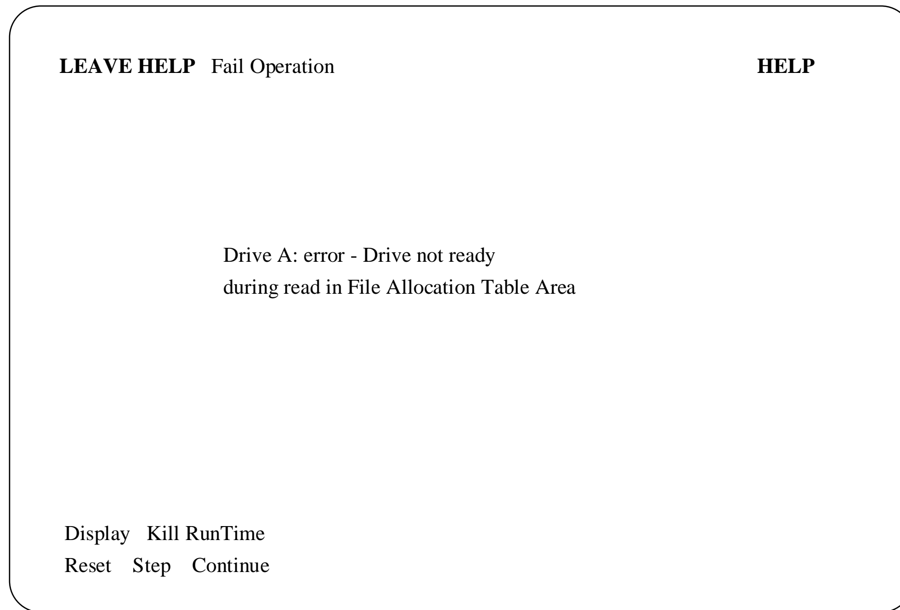
2.10.3 Device Not Ready Conditions

For certain I/O related errors, generation of standard RunTime errors is not possible. Here, the RunTime program attempts to displays a special error screen referred to as the "Device Not Ready" screen.

For example, under MS-DOS, attempting to access a diskimage file on a diskette when no diskette is in the drive, as in:

```
:0010 $DEVICE(/D20)="A:PLATTER1.BS2"
:0020 LISTDCT/D20,
:RUN
```

would produce the following screen:



This screen operates very much like the standard ERROR or HELP screen in that listed options may be selected by moving to the desired option and pressing the EXECUTE key.

The options at the bottom of the screen perform the same functions as on the HELP screen.

NOTE: The Native O/S option is not available on this screen because exiting to the native operating system while an error condition is in effect is not possible.

The Reset, Step, and Continue options are available only in the Interpretive version (RTI).

The LEAVE HELP option operates differently depending on the type of I/O operation that was being performed when the error occurred. If the operation was a disk operation, specifying LEAVE HELP causes the operation to be retried. This allows for recovery from diskette-not-mounted situations. For instance, in the example above inserting a diskette with a file named PLATTER1.BS2 and then specifying LEAVE HELP would allow the LISTDCT statement to proceed normally. For print class operations, specifying Leave Help causes the operation to be skipped. RTP attempts to continue with normal processing.

NOTE: Most print class statements actually generate a series of print class operations. Thus, to get past one statement may require pressing LEAVE HELP many times.

The Fail Operation causes the RunTime to abort the attempted operation and return control to the application. In the case of disk class operations, the Fail Operation usually results in a disk class error being generated. For instance, in the example above specifying Fail Operation would result in a P48 - Illegal Device Specification error with an MS-DOS error code of 0D02 (drive not ready) being generated. If RTP were being used, the standard RTP error screen would be generated.

In some cases it is necessary to specify the Fail Operation twice to get the standard RunTime error. For print class operations, specifying the Fail Operation generates a RunTime error only if the ERR= Y clause is specified in the $DEVICE statement for the device address used.

NOTE: The Fail Operation option should only be used in cases where generation of an NPL error code is essential to determining the cause of the problem.

The fail operation option is only available under NPL Revision 3.0 or greater.

NOTE: The conditions which generate the Device Not Ready screen may vary widely from one operating system to the next. On many operating systems, detection of device not ready conditions is not possible at all at the program (RunTime) level.

2.10.4 Error Descriptions

In addition to displaying the NPL error code, a description of the error code is displayed if files containing these error descriptions are present.

NOTE: The native operating system error message will also be displayed, when appropriate if the proper help files are available. Refer to Section 8.5. for more information.

Located on the RUNTIME PACKAGE diskette are four files, labeled ERRORMSG.HLP, RTIxERR.HLP, ERRORMSG.IDX, and RTIxERR.IDX which should be copied to the default directory on the hard disk where the RunTime is installed. The files with the extension of .HLP contain the literal description of all NPL error code and applicable native operating system error codes respectively. The files with the .IDX extension are the Indexed Help files required to "find" the proper entry in the descriptive file.

When these files are installed, the error code is accompanied by the literal translation, and in cases where the error generates an operating system error, that code and literal translation are also provided.

For examples:

```
20 CONVERT A$ TO X
       ^
ERR X75: Illegal number
```

NOTE: This error does not generate an operating system error code. Therefore, no further information is reported.

```
10 $DEVICE(/D12)="LISTTEST":      REM "LISTTEST" DOES NOT EXIST
20 LISTSDCT/D12,
                        ^
ERR P48: Illegal device specification
    "LISTTEST"
O/S error code 0002: File not found
```

In the example above the file "LISTTEST" was not a valid native operating system file. Therefore, a native operating system error code (0002) was generated along with the descriptive message "File not found" contained in RTIxERR.HLP (x refers to the host operating system designation).

Refer to Chapter 11 for details on the use of Help files. Refer to Chapter 13 for details on the Indexed Help File Processor. Refer to the NPL Statements Guide, $OSERR, ERR$ for information on the literal descriptions of error codes.

CHAPTER 3

RUNTIME MEMORY USE

3.1 Overview

This chapter discusses how the RunTime allocates memory and handles program and variable storage.

Section 3.2 discusses dynamic partitioning under NPL.

Section 3.3 discusses the determination of memory using the SPACE statements.

Section 3.4 addresses the issue of p-code size relative to code on the Wang 2200.

3.2 Dynamic Partition Size

All NPL program code and defined variables reside within a section of memory defined as the "user partition". As of Release III, the size of the user partition is limited only by the available physical memory or logical task size of the host operating environment.

Upon execution of NPL, the RunTime interrogates the operating system environment and determines the amount of memory available for the user partition. The amount of memory available to the RunTime is dependent upon, but not limited to the following elements:

- The revision of the native operating system being used.
- Any operating environment loaded on top of the operating system.
- The number of files, buffers, processes and other operating-system dependent parameters defined in the system configuration.
- Which RunTime program is being executed: RTP or RTI.
- RunTime options invoked at startup.
- Operating system task size limitations.

On most NPL operating environments, memory allocation is dynamic (refer to Chapter 8 of the appropriate NPL Supplement for details). Due to this dynamic nature of memory allocation, the RunTime (on most platforms) allocates an initial 141K to the user partition. As such, SPACEW returns the minimum size of the user partition because attempting to allocate the full amount of the memory available would cause performance degradation. As additional memory is needed by the application, the RunTime automatically attempts to allocate another 64K of memory to the user partition. The process continues, based on the applications memory need, until all available system memory is used.

NOTE: Once memory is allocated in this fashion, it is not released until the Runtime is exited.

On other environments, memory allocation may be static. In this case, the RunTime allocates all available memory at startup and SPACEW shows the total amount of memory available to the application.

3.3 Space Functions

NPL provides a number of functions for determining memory available to the RunTime. The following is a list of these available SPACE functions:

SPACEW	This reports the minimum size of the user partition on those operating environments with a dynamic partition size or returns the total size of the user partition in environments with static partition size. SPACEW always returns a number of bytes.
SPACEF	Returns the current amount of space available for additional program code or variables in bytes. On those operating environments that support a dynamic partition size, when the value of SPACEF drops below 64K, the RunTime automatically attempts to allocate another 64K of memory to the user partition. Values greater than 64K may be returned.
SPACE	Returns the minimum of SPACEF or 65488 bytes.
SPACEK	Returns the total size of the user partition in kilobytes (bytes/1024), up to a maximum of 61.
SPACEP	Syntactically supported for compatibility with previous releases of NPL. Returns same value as SPACE.
SPACEV	Syntactically supported for compatibility with previous releases of NPL. Returns same value as SPACE.

Refer to the NPL Statements Guide for further details on these statements.

3.4 P-Code Size Relative to Wang 2200

When compiling Wang 2200 format (atomized) code to NPL (p-code), be aware that memory requirements for each program may increase. As a point of reference, expect anywhere between a 10 to 20 percent increase in the size of the code. This is due to the radically different internal storage format of p-code as opposed to atomized code.

NOTE: The actual difference in memory required may vary widely from one program to another.

for each program may increase. As a point of reference, expect anywhere between a 10 to 20 percent increase in the size of the code. This is due to the radically different internal storage format of p-code as opposed to atomized code.

NOTE: The actual difference in memory required may vary widely from one program to another.

CHAPTER 4

NPL PROGRAMMING CONSTRUCTS

4.1 Overview

This chapter discusses the programming constructs offered in NPL Release IV that aid in a programmer's ability to develop well-structured, modularized code.

Section 4.2 discusses the general topic of logical program organization.

Section 4.3 discusses the use of line numbers and statement labels along with their effect on the order of execution and transfer.

Section 4.4 discusses data types.

Section 4.5 discusses the definition and use of literals and constants.

Section 4.6 discusses the different types and classes of variables and their allocation and scope.

Section 4.7 discusses operations and expressions for numeric and string (alpha) variables.

Section 4.8 introduces FUNCTIONs and PROCEDUREs.

Section 4.9 discusses the function call interface. It also covers different types of parameter passing and the referencing of a function.

Section 4.10 discusses the concept of program modules.

Section 4.11 discusses the various logical constructs introduced with NPL Release IV.

Section 4.12 discusses the use of RECORDs and FIELDs.

4.2 NPL Logical Program Organization

The design of a computer program involves three fundamental steps:

- Identifying the task to be carried out
- Devising an algorithm to carry out the task
- Implementing the algorithm in a programming language

Structured programming techniques and code modularization enable developers to accomplish both design and programming simultaneously. Through features and tools available in NPL Release IV, NPL developers, in particular are able to make use of these constructs and methods, resulting in more logical program organization (structured programming, code modularization and program lines are discussed in the following sections).

4.2.1 Structured Programming

To understand the application of different programming elements of Release IV, it is important to recognize the varied interpretations of the term "structured programming." For example, a structured program can be a program that is constructed using smaller programs as "building blocks" or modules. A structured program can also use certain programming statements or constructs such as conditional branches or loops that make the program well organized. A structured program can sometimes be called a "GOTO-less" program. For further illustration, any of the following can be considered a structured program:

- A null program.
- A statement which does not cause a branch (other than function calls).
- A call to a function or procedure whose body is a structured program.
- A conditional test which executes one structured program if the test is true or another structured program if the test is false, and each case continues at a single exit point of the condition.
- A conditional test which executes a structured program if the test is true, then repeats the same test, or if the test is false continues at a single exit point.
- A structured program followed by another structured program.

Several programming elements are offered in NPL Release IV that allow the developer to implement virtually any program in a structured style.

These include:

- IF...ELSE...ENDIF structures. These are used as a true/false conditional statement.
- WHILE...WEND loop structures. These continue only while a condition, specified at the start of the body, is true.
- REPEAT...UNTIL loop structures. These execute once, then continue only while a condition, specified at the end of the body, is true.

- SWITCH...CASE...END SWITCH. This is a conditional for multiple and mutually exclusive true/false cases, eliminating the need for nested IF statements.
- FOR/BEGIN...NEXT loop structures. These execute using an index or counter that regularly changes with each iteration of the loop.

Refer to Section 4.3 for more information on these program elements.

NOTE: **Some structured programming elements are implemented for expediency, and in some respects contradict the spirit of structured programs. Over using of these facilities can make the logic of an otherwise structured program difficult to follow. These expediency elements include:**

- BREAK allows an unconditional exit from a WHILE...WEND or FOR/BEGIN...NEXT structure.
- LOOP allows skipping over the body of a loop, to perform the next iteration.

NOTE: **Refer to Section 4.3 for more on the use of BREAK and LOOP.**

In addition to these statements, NPL Release IV supports the use of Long Identifier Names, important in structured programming. Long Identifier Names, or LINs, can be up to 255 characters in length and can consist of any combination of alpha-numeric characters or underscore characters. The use of LINs in a program also aids in clarity by allowing for more descriptive variable names, as well as a virtually unlimited number of identifiers. Variables are discussed in more detail in Section 4.6.

4.2.2 Code Modularization

The idea of modularized code is an extension of structured programming. While the use of structured constructs aids in the design of logical easy-flowing programs, division of the code into specific tasks (FUNCTIONs and PROCEDUREs) improves upon that goal. Each division, or module, is written to perform some specific task. It may receive values (parameters) as input and may return results as output. The internal operation of the module used to obtain the results has no dependence or effect on the rest of the program. This type of modularization is sometimes referred to as "black-box" programming. When a parameter is passed to a function or procedure, it doesn't matter how the calculation or task is performed, as long as a result (if applicable) is produced.

NPL Release IV provides a number of facilities to aid in code modularization:

Program modules allow the creation of separate address spaces for line numbers, variable names, and marked subroutine names. This allows for separate program workspaces where programs can be written which are completely independent of each other, using the same line numbers or variables, while still allowing communication with each other. Modules are discussed in more detail in Section 4.10.

Statement labels may be used to identify the start of subroutines, and can also be used as an entry point for transfer of execution from a GOTO or GOSUB. Statement labels are discussed in more detail in Section 4.3.

A function interface allows parameter passing with variable identifiers, references (pointers), and permits recursive use of variables. This also allows for a standard method for return values and error indication. The body of the function provides a local scope, allowing variables to be declared and accessed only from within the function. FUNCTIONs, PROCEDUREs, and the function call interface are discussed in more detail in Sections 4.8 and 4.9.

4.3 Program Lines

At a very low level, an NPL program can be viewed as a set of numbered program lines, each containing one or more statements delimited by a statement separator. A statement is the smallest executable unit of NPL. A program line containing more than one statement is termed a multi-statement program line. Line numbers can be any number in the range 0 to 32117. The same line number can be used in any NPL module without conflict. A REM statement may be used to indicate a comment until the next colon or program line. A semi colon may be used to indicate a comment up to the end of a physical text line.

For example:

```
0100 REM ignored until the next colon : PRINT "This will print."
0110 ; until the end of line is ignored : PRINT "This won't print."
```

4.3.1 Statement separators

Each line of an NPL program can contain one or more statements separated by a statement separator. By default the statement separator is a colon. To make programs even more readable, it is possible to embed "soft" carriage returns in the program line before a statement separator. The statements after a soft carriage return appear on the next line. This makes NPL code more readable and easy to follow. Refer to the appropriate hardware supplement for the proper key sequence to insert soft carriage returns when editing programs. The following is an example of a program without soft carriage returns, and the same program with soft carriage returns.

```
0010 FOR Number=1 TO 10 : Cube=Number^3 : PRINT "Cube of ";Number;"
is:";Cube : NEXT Number

0010 FOR Number=1 TO 10
    :   Cube=Number^3
    :   PRINT "Cube of ";Number;" is:";Cube
    : NEXT Number
```

In some cases, a soft carriage return can be used without a statement separator. In DIM statements and FUNCTION and PROCEDURE declarations, soft carriage returns can be used after commas to separate variable declarations, such that variables can be grouped on a line, with comments added. For example:

```
0005 DIM Buffer$256, Matrix(3,3), _CompanyName$="Acme", Count1, Count2

0005 DIM Buffer$256,                ; Record buffer variable
    Matrix(3,3),                    ; 3x3 matrix array
    _CompanyName$="Acme",           ; Constant String
    Count1,Count2:                  ; Number counters
```

4.3.2 Order of Execution

Program lines are executed sequentially in line number order and statements within a multi-statement line are executed left to right. This order of execution is only altered by an instruction which transfers control to another program location or into another module, such as a GOTO, GOSUB or call to a FUNCTION or PROCEDURE (refer to Section 4.3).

4.3.3 Statement Labels

Identifiers may be used to mark locations in the program, making code more readable. If a GOTO or GOSUB is required, a descriptive label, rather than a line number, should be used for its target location. This makes the code more self-documenting and reduces the number of line numbers required in a program.

For example:

```
0010 Counter=10
0020 =Top_of_Loop
    : GOSUB prt_counter
    : Counter-=1
    : IF Counter > 0 THEN GOTO Top_of_Loop
0030 END
0040 =prt_counter
    : PRINT Counter
    : RETURN
```

Statement labels which occur inside a function body are local to the function. Statement labels which occur outside all function bodies are not accessible from inside any function. The same label may be used in multiple functions within a module without conflict, or may be used in both the mainline and a function without conflict.

Statement labels are not permitted in immediate mode. An immediate mode reference to a statement label (e.g., GOSUB process_record) is executed in the context of the currently executing function (if any).

4.3.4 Transfer of Control

The GOTO/GOSUB or ON...GOTO/ON...GOSUB statements may be used to transfer control from one area of a program to another. Where possible, these statements should be replaced with a suitable structured construct making the logical flow of the program more apparent. Subroutines called with GOSUB statements that perform reasonably large amounts of logic, and that do not reference many local variables may be easily replaced with a PROCEDURE that has a few well-defined parameters. ON...GOTO/ON...GOSUB statements can be easily replaced with a SWITCH...CASE construct.

NOTE: If a GOTO or GOSUB statement is unavoidable, it is recommended that a descriptive label rather than a line number be used for clarity.

The BREAK and LOOP statements also transfer control, but can be used only inside logical constructs. These statements contradict the spirit of structured programming because they provide an alternate exit point from a module. However, they are provided for maximum flexibility. When used in a FOR/BEGIN...NEXT loop, the BREAK statement offers the best "clean" way of terminating a loop without leaving entries in the program stack.

The BREAK statement allows an unconditional exit from WHILE...WEND, REPEAT...UNTIL, and FOR/BEGIN...NEXT loops.

For example:

```
0010 Counter=0
    :; Check first 30 sectors
    : REPEAT
    :   DATA LOAD BA T/D11,(Counter) Buffer$
    :   Counter+=1
    :   IF Counter>30 THEN BREAK
    :   GOSUB CheckSector
    : UNTIL FALSE
0020 PRINT "Program continuing..."
```

In the above example, The FALSE statement is a keyword that can be used as an expression that always carries a false condition, thus the REPEAT...UNTIL will always continue, since the expression is never true. Thus, the only way to terminate the loop is with the BREAK statement.

The LOOP statement passes execution to the end of the current loop structure. The ending statement then evaluates whether to continue at the beginning of the loop or to exit the loop.

For example:

```
0010 FOR F=1 to 20 BEGIN
    :   IF F=13 THEN LOOP    ; Skip the 13th floor
    :   PRINT "Floor #";F
    : NEXT F
```

4.4 Data Types

NPL supports two data types: strings and numerics. Each data type is discussed below.

4.4.1 String

A string is any sequence of characters in which each character may have an ASCII value of 0 to 255 inclusive. In general, strings have a maximum length of 64K, however, this limitation is not imposed on 32-bit operating environments. If requested, string allocations up to the largest signed 32-bit number (2048 megabytes) will be attempted. Restrictions imposed by the operating environment will determine whether such allocations can succeed.

4.4.2 Numeric

All numbers are represented in floating point format with approximately 14 (fourteen) digits of precision. There is no explicit integer type. Numbers are stored internally with a signed 47 bit mantissa and a signed 15 bit exponent. A numeric value is represented as:

Mantissa * 10^{Exp}

where Mantissa = 2^{47} and Exp = 2^{15}

The range for positive numbers is:

1.0E-32767 to 140737488355329E+ 32767

and the range for negative numbers is:

-1.0E-32767 to -140737488355329E+ 32767

4.5 Literals and Constants

As used in NPL, literals and constants are values or expressions which do not change during program execution. Literals are expressed as a specific value, while constants are expressed using an identifier which was previously declared with a specific value.

The following sections provided detailed descriptions of literals and constants, respectively.

4.5.1 Literals

A string literal may be either a sequence of ASCII characters enclosed in double quotes (") or a HEX() literal.

NOTE: To specify a double quote within a string, enter it twice. For example, the string A$= """""" consists of one double quote character.

Some examples of string constants include:

```
"This is a string literal"     "The title is ""A TITLE""."
 HEX(4142430D)  represents "ABC" followed by a carriage return.
```

4.5.2 Numeric Constants

A numeric constant is a sequence of decimal digits followed by an optional decimal point, one or more digits and/or an exponent. An exponent consists of either the letters "e" or "E" followed by one or more decimal digits. Both the number and exponent may be preceded by a plus or a minus sign. The number preceding the decimal point may be omitted.

Some examples of numeric constants are:

```
1            1.          1.2              1.2e-34            0.2
-.2E34                +  4.1234 e+ 56
```

4.5.3 Constant Variables

Constant variables are defined as identifiers which can only be assigned a value once, during declaration. Constant variables are a special class of variables which can only be assigned a value when they are created. They may not be assigned a new value at execution time.

A constant is defined using the DIM statement. A constant identifier must begin with an underscore "_" and can use any combination of alpha numeric characters (A-Z, a-z, 0-9) or underscores. Constant identifiers can be used anywhere a variable identifier can, except on the left-hand side of an equal sign.

For example:

```
DIM _TaxRate = 0.0625
DIM _Flavor$16 = "BananaStrawberry"
PRINT Price*_TaxRate
```

4.6 Variables

The following sections detail the variable options available under NPL.

4.6.1 Naming Conventions

NPL Release IV supports long identifier names (referred to as LINs), which can be up to 255 characters in length, for the naming of program variables, statement labels, and FUNCTIONs. Identifiers must start with an alphabetic character (a-z, A-Z), followed by any combination of alphanumeric characters (a-z, A-Z, 0-9) and the underscore characters.

NOTE: NPL displays the underline as a left-pointing arrow under the standard NPL character set. On IBM-PCs, this default can be changed by using the Screen Translation Utility to map NPL HEX(5F) to a native HEX(5F), which is the underscore character.

Significance of Spaces in Syntax

To enter long identifiers, separate the start and end of the identifier from keywords by a space or other punctuation character which clearly distinguishes the identifier. For example, on previous releases of NPL, the statement:

```
FOR I = A TO B
```

could be entered without any spaces between the identifiers and keywords, as:

```
FORI=ATOB
```

However, since NPL supports long identifier names, the second form (without spaces) has a different meaning, equivalent to an assignment statement. Instances of this type exemplify possible syntactic ambiguities.

When a syntactic ambiguity occurs, the NPL compiler assumes the new rules apply. In the above example, NPL treats the second form as an assignment statement under Release IV. In such cases, if the first form was intended, reenter the line with the necessary additional spaces.

Identifier Abbreviations

To help developers avoid typographical errors when entering long identifier names, the "Recall" key may be used (in immediate mode) to complete the name of the identifier after entering a partial name at the current end of a line.

For example: (the "|" represents the cursor position)

```
:10 DIM SausageName$27
:20 PRINT Sau|
```

At this point, if "Recall" or the right-arrow is pressed, NPL looks for a unique identifier beginning with "Sau..." If only one identifier exists, NPL completes the name with the standard case. If more than one identifier starts with the partial name entered, the system beeps and no change is made to the line.

HINT: Developers using LINs may also find it useful to set byte 38 of $OPTIONS to HEX(01), which requires all identifiers to either be declared as common (COM) or in a DIM statement before any other reference. A P55-Undefined Variable error occurs at resolve time for the first reference not preceded by a declaration.

This can help avoid programming errors such as:

```
10  DIM SausageCount
20  SausageCoutn = 12     ; Oops, meant SausageCount
```

With byte 38 of $OPTIONS set to HEX(01), the above example would produce an error P55-Undefined Variable on line 20 at resolve time, indicating that a variable appeared before its declaration in a DIM or COM statement. Otherwise, the misspelled variable identifier causes a new variable to be implicitly declared. Refer to the Statements Guide for more information on the use of $OPTIONS.

4.6.2 Case Conventions

Differences between upper and lower case are ignored by NPL. Enter an identifier as any combination of lower and upper case. An identifier may not contain embedded spaces.

NPL displays all instances of an identifier using consistent case.

Short Identifier Names

NPL always displays short variable identifiers (supported on previous releases) in upper case.

NOTE: Short variable names are defined as all variable names supported prior to NPL Release IV.

```
:10 dim a61
:list
0010  DIM A61
```

The above example illustrates the declaration of a short variable name. In this case, NPL converts it to upper case.

This example illustrates the declaration of a LIN and, thus, NPL retains its lower case.

```
:10 dim sd9
:list
0010  DIM sd9
```

Long Identifier Names (LINs)

While short variable names are always displayed in upper case, long identifier names (LINs) retain the case as entered most recently. Continuing the above example:

```
:10 dim sd9  ;will display the case of its last reference in the module
:20 Sd9=2
:list
0010 DIM Sd9
0020 Sd9=2
```

The variable was most recently referenced with a capital "S" so the entry in the variable table was updated. Now all references in the program reflect the change accordingly.

NOTE: For LINs, the first instance in a (B2C) compiled program determines the case. Any manual program modifications (made in RTI) change the case convention to that which was last entered. Array variables and scalar variables which use the same root identifier can have different case conventions. String variables and numeric variables which use the same root identifier can also have different case conventions. In fact, each different context in which a variable identifier appears (statement label, function name, interface package label) may use different case conventions. For example, MainLine, MAINLINE(X) and Mainline$ could all appear in the same program listing and represent different variables. Immediate mode commands do not implicitly change the case.

HINT: Due to the large number of reserved words in NPL, it is recommended that at least some lower case letters be used in long identifiers. This may help minimize any confusion between identifiers and reserved words as NPL will always display a reserved word entirely in upper case. Although reserved words are displayed in upper case, they may be entered in any mix of upper and lower case.

4.6.3 Variable Types

The following section discusses the various NPL variable types.

String Variables

String variables are used for representing a sequence of characters. Identifier names are always followed by the type specifier "$". String variables can be either scalars or arrays. The term 'alpha-variable' is used throughout the documentation to refer to a more general string class which may be either a scalar string variable, an element of a string array variable, an entire string array, a substring of another alpha-variable (using STR() function), a string function, or a string field-expression. Arrays and field-expressions are covered later in this section. Identifier names should follow the naming conventions outlined in section 4.6. Examples of some valid string variable names include:

```
A$  b2$   char$   Employee_Name$  R2D2$
```

String variables can be allocated an initial size when declared. If a size is not specified, it defaults to sixteen characters. Strings can later be dynamically increased in size if necessary. String variables may be initialized upon declaration with a valid literal. Non-initialized string variables are filled with all bytes containing HEX(20) (blanks).

String variables do not have a "current" length maintained and it is not possible to have an "empty" string. The LEN() function can be used to return the number of characters in a string ignoring any trailing blanks. LEN(STR()) can be used to determine the defined length of a string.

For example:

```
COM CommonStr$1 = HEX(ff)
DIM aString$6 = "ab cd" , ControlCodes$10 = HEX(030A0D)
```

then the following:

```
PRINT LEN(aString$)              would display 5 and
PRINT LEN(STR(aString$))         would display 6
```

The total size of a string variable may not exceed 64KB on many NPL platforms. Programs which must be portable should avoid the use of variables larger than this. With NPL Release IV in 32-bit environments, larger variables are permitted. Refer to section 4.6.7 for details on restrictions.

Numeric Variables

A numeric variable is used to represent a floating point number. There is no explicit integer type. Numeric identifier names should follow the naming conventions outlined in Section 4.6. Numeric variables do not require or permit a type specification character "$". Some examples of valid identifier names include:

```
A  X6  hourly_wages  Account_Num_1234
```

Numeric variables may be initialized upon declaration with any valid numeric expression or constant. Uninitialized numeric variables are assigned the value zero.

For example:

```
COM CommonNumeric         ; default value is zero.
DIM aNumber = 10, Sqr_Root_2 = SQR(2.0)
```

Array

Arrays may be either numeric or string with identifier names following the naming conventions outlined in Section 4.6. The array type is determined the same way as for ordinary variables (i.e., with string arrays having a "$" appended to the identifier name). Arrays are one or two dimensional tables of values of the same type. Arrays can also be constants. Constant arrays must have an underscore "_" at the beginning of the identifier. The same name may be used for both an array and an ordinary variable. Array names are generally followed by an opening parenthesis.

The following are valid numeric array variables:

```
A()   B2()   Matrix()   Salaries()
```

The following are valid alpha array variables:

```
A$()   B2$()   SillyStrings$()   Names_of_Sisters$()
```

The following are valid constant arrays (numeric and alpha examples):

```
_A()   _B2$()    _PostalCodes$()    _Cloud_Types$()
```

String or numeric array elements are accessed by positive numeric subscripts starting at 1 (fractional subscripts are truncated to the integer value). All elements of a string array are of the same length.

Single dimensional arrays may have a maximum of 65535 elements and two dimensional arrays may have up to 65535 by 65535 elements. On 32-bit platforms this restriction is not imposed and arrays may be dimensioned up to the largest signed 32-bit number provided there is enough memory. A two dimensional array requires two subscripts separated by a comma.

For example:

```
DIM Strings$(100)80, Numbers(20), Matrix(100,10)
DIM Buffer$(20)_MaxBufferSize
Strings$(1) = "This is a sentence."
Strings$(2) = STR(Buffer$(CurrentBuffer),1,80)
Matrix(i,j) =  Numbers('index(i,j))
PRINT _Month$(7)
```

Statically allocated arrays may be expanded beyond their initial allocation with the use of the MAT REDIM statement. Refer to Section 4.6.8 for details and restrictions.

Pointer

Arguments to FUNCTIONs or PROCEDUREs are passed either by value or by reference. Parameters passed by reference are called pointer variables and are indicated by the type specification keyword /POINTER. Here, any reference to the parameter within the function body refers to the argument directly. Refer to Section 4.8 for more information on FUNCTIONs and PROCEDUREs.

For example, given the following function:

```
0010 PROCEDURE 'Whatever(Argument1, /POINTER Argument2 )
    : Argument1=0
    : Argument2=0
    : END PROCEDURE
0020 ; Start of main program
0030 A=1:B=2
    : 'Whatever(A,B)
    : PRINT A; B
```

would show that A was still equal to one but that B had been changed to zero.

NOTE: Pointers are designed to be used when referencing very large sets of information or where multiple result values are produced by a PROCEDURE.

Constant Variables

Constant variables are variables that are given a value upon declaration and are not allowed to be modified during a program's execution. Constant variables are indicated by an underscore character preceding the identifier name.

For example:

```
COM  _a_Common_variable=9
DIM  _MaxTableLength=100
DIM  _PI_DIV_2=#PI*0.5
DIM  _Str_Constant$4="ABCD"
```

Constant variables are also covered in Section 4.5.

Field Identifiers and Logical Records

A field identifier is a variable type which provides easier examination and modification of values in logical data records. A logical RECORD is defined as a block of code, starting with the RECORD keyword and ending with the END RECORD keyword. Field identifiers follow the same conventions as variable identifiers, but are grouped together using the logical RECORD structure and use relative offset, length, and format specifications to determine placement within a RECORD.

Refer to Section 4.12 for a more detailed discussion of Logical Records and the use and definition of Field Identifiers.

4.6.4 Allocation Classes of Variables

The allocation class of a variable refers to the method used by NPL to allocate and release memory for the variable. NPL supports two allocation classes:

Static Allocation

Memory for statically allocated variables is reserved when a program is resolved (when first RUN, or before an overlay).

Memory for statically allocated variables is released when either a CLEAR N or CLEAR V command is executed, to clear non-common or all variables, respectively. The CLEAR N command occurs implicitly each time an overlay occurs.

Recursive Allocation

Recursively allocated variables are always associated with a function.

Memory for a new copy of recursively allocated variables is reserved each time the associated function is called. While executing the body of the function, the most recently allocated copy is referenced by the variable name.

Memory for recursively allocated variables is normally released when the function returns to the calling program.

Within a module, any variable which has not been declared within a function or procedure body will have an implicit storage class of Static. Any variable within a module, including those within a function or procedure body, may be explicitly given a Static storage class by using the /STATIC keyword in the variable's declaration. All variables declared within either a function or procedure are implicitly given a storage class of Recursive. The only time one really needs to be concerned with a variable's storage class is when a variable's value must be retained across successive function calls. In such cases the variable must be explicitly given a Static storage class upon its declaration within the function or procedure.

For example:

```
0010 ;
    : PROCEDURE 'anyProc /FORWARD
    : COM string$10,n             :; always implicitly static
    : DIM letter$1 :; implicit static
    : DIM /STATIC age=1           :; explicit static, not necessary
    : ;
    : char$="c"    :; undeclared variable, implicit static
    : ;
    : 'anyProc
    : 'anyProc
    : 'anyProc
    : END
    : ;
0020 PROCEDURE 'anyProc
    : ; recursive storage class always implied
    : DIM height=0
    : ; recursive storage class specified explicitly, not necessary
    : DIM /RECURSIVE width=0
    : ; explicit static storage class, required to preserve value
    : ; of variable 'weight' across multiple function calls.
    : DIM /STATIC weight=0
    : ;
    : height+=1
    : weight+=1
    : PRINT "Height = ";height;" Weight = ";weight
    : END PROCEDURE

:RUN
Height =  1  Weight =  1
Height =  1  Weight =  2
Height =  1  Weight =  3
```

In the above program, each call of the procedure 'anyProc modifies the value of the variable "weight," by adding 1 to its previous value that was calculated during the last call. Although the variable "height" also has its value modified within each call, the value is not preserved since it is of recursive class.

4.6.5 Scope of Variables

The scope of a variable refers to the range of statements within a program where it is permitted to use a previously declared variable name, and have it refer to the same declaration.

It is possible, within a program, to declare the same variable name for different purposes, either by accident or design. For example, a parameter common to several related functions might, in all functions, be called "Buffer". In such cases, each reference to the variable by name is, at least potentially, ambiguous. However, by clearly defining the scope of each declaration of a variable in such a way that one declaration takes precedence, this potential ambiguity disappears.

A variable declaration is said to be "in scope" at a given point in a program if a reference to the variable is legal there and refers to the variable intended (not to another variable with the same name declared for a different purpose).

NPL supports three variable scope types, Function-private, Module-private, and Public.

Function Private Variables

The declaration of the variable occurs within a function body.

The declaration is "in scope" on all statements which appear within the same function body and after the declaration.

Module Private Variables

The declaration of the variable does not occur within a function body or PUBLIC section, or in any COM declaration. The declaration may not have the /PUBLIC keyword.

The declaration is "in scope" on all statements in the same module and after the declaration, unless a function-private variable declaration with the same name is already "in scope."

Public Variables

The declaration of the variable occurs in a PUBLIC section, or has the /PUBLIC keyword.

The variable is "in scope" on all statements in the module following the declaration, unless a function-private variable or module-private variable declaration with the same name is already "in scope." A specific variable name may only be declared PUBLIC by one module (its "owner.")

In addition, the variable is "in scope" on any statement in any other modules that INCLUDE the variable’s "owner" module, provided:

1. The statement is after the INCLUDE statement.

2. No function-private or module-private variable of the same name is already "in scope."

3. If the declaration is in a named PUBLIC section, the statement must also be after a USES statement for the named section.

NOTE: If a PUBLIC section contains other nested INCLUDE or USES statements, the public variables that become "in scope" due to these statements are also brought "in scope" when the containing section is INCLUDEd.

4.6.6 Variable Declaration

Variables can be declared either explicitly or implicitly, and can be declared with an initial value.

Explicit declarations are performed using the DIM and COM statements, and also using parameter syntax in FUNCTION and PROCEDURE declarations. Refer to the Statements Guide for specific rules on the use of these commands. Refer to Section 4.8 for more on FUNCTIONs and PROCEDUREs. The following are some examples of explicit declarations:

```
0010 COM Buffer$30                 ; Declares common alpha var.
0020 COM _CommonNumber=14          ; Declares constant with initial value
0030 DIM SillyString$40            ; Declares 40-byte alpha var.
0040 DIM Array$(10)24              ; Declares alpha array with 10
                                   :; elements of 24 bytes each.
0050 DIM Number1,Number2           ; Declares two numeric vars.
0060 DIM Month$10="January"        ; Declares alpha with initial value.
```

Explicit declaration also takes place when parameters to FUNCTIONs or PROCEDUREs are specified.

For example:

```
0010 FUNCTION 'PieceOfString$(String$40,Byte,Length)
```

The parameters are explicitly declared and by default, are recursive and private to the function. There are other ways to declare variables in a parameter list. This is covered in Section 4.8 and 4.9.

Undeclared Variables

For historical reasons, NPL permits string and numeric scalar variables (and, under some circumstances, numeric and string arrays) to be used without declarations. An implied declaration occurs when a previously undeclared variable is referenced in a program (for arrays, only references in some MAT statements qualify to force an implied declaration). The variable is allocated as if a DIM statement (with default sizes for strings and arrays) preceded the statement.

NOTE: Although undeclared variables are still permitted in some cases for upward compatibility with previous releases, their use is discouraged.

Undeclared numeric and string variables whose default declaration would make them PUBLIC (in PUBLIC section) or RECURSIVE (in function bodies) are diagnosed as errors.

NPL can be forced to treat all references to undeclared variables as errors if $OPTIONS byte 38 is set to HEX(01), This would require all variable references to be preceded by a declaration in a DIM or COM statement; otherwise, a P55-Undefined variable error occurs.

4.6.7 Variables Larger than 64KB

With NPL Release IV under 32-bit environments, the RunTime attempts any variable allocations up to the largest signed 32-bit number (2048 megabytes). However, restrictions imposed by the operating environment will determine whether such allocations can succeed.

In particular, operating environments which do not use virtual memory have a limit due to the amount of available real memory. Operating systems which use virtual memory have a limit due to the amount of backing store in the system swap area(s). Additional quota limitations per process may be present in multi-tasking sites which are designed to prevent accidental or intentional overuse of virtual memory allocations that could impair other users' operations.

In virtual memory environments, allocating extremely large variables may result in "thrashing", depending on the amount of available real memory. NPL, by default, performs initialization of all variables to either blanks (for strings) or zeroes (for numerics). This can result in significant amounts of I/O to the swap area.

Restrictions on Use

Some NPL statements are designed to return string positions as a 2-byte binary value, under the assumption that no variable exceeds 64K bytes. These statements include:

- MAT SEARCH

  When an alpha array is specified as a locator, a maximum index of 64K-1 bytes into the search array can be returned. If MAT SEARCH is executed (which would return a result out of range), an error P34-Illegal value error occurs. The recommended correction in this case is to use a numeric locator array for the result, which does not have this restriction. If the STEP value is bigger than 1, using the ELEMENT keyword can also reduce results so that the range error does not occur.

- $GIO returned length

  Return lengths of output or input using $GIO in the arg-2 variable may not exceed 64K. Attempts to use I/O with buffers in excess of this size, when arg-2 is specified, result in an error P52-Variable or value too large.

- PRINTUSING TO/ PRINT TO buffers

  The initial 2 bytes used to contain the "amount of buffer used" as a two-byte binary value restricts the total size of a PRINTUSING TO or PRINT TO buffer to 64K.

- MAT SORT array size limitations

  The design of the MAT SORT pointer array normally limits the use of this statement to two-dimensional arrays with up to 255 rows and 255 columns, and to one-dimensional arrays with up to 65535 elements.

  NPL Release IV allows the use of a pointer array with 4 bytes per element, permitting one-dimensional arrays larger than 65535 elements and two-dimensional arrays of up to 65535 by 65535 to be sorted. Here, the work array requirement also increases to 4 bytes per element. The resulting pointer array contains BIN(,4) format array indices for one-dimensional arrays, or BIN(,2)&BIN(,2) array indices for two-dimensional arrays, and may be used in MAT MOVE.

WARNING--The sort algorithm performs multiple non-linear passes through the key array. In virtual memory environments, this can result in excessive paging on very large key arrays.

- MAT MOVE array size limitations

  The design of the MAT MOVE pointer array normally limits the use of this statement to two-dimensional arrays with up to 255 rows and 255 columns, and to one-dimensional arrays with up to 65535 elements, when a pointer array is involved.

  Release IV allows the use of a pointer array with 4 bytes per element, permitting one-dimensional arrays larger than 65535 elements and two-dimensional arrays of up to 65535 by 65535 to be moved. The pointer array must contain BIN(,4) format array indices for one-dimensional arrays, or BIN(,2)&BIN(,2) array indices for two-dimensional arrays, as produced by MAT SORT.

- MAT MERGE array size limitations

  When used with 2-byte pointer arrays, the design of the MAT MERGE control array continues to limit the use of this statement to two-dimensional arrays with up to 255 rows and 254 columns.

  If four-byte pointer arrays are used, the specifications for a control array change, requiring two bytes per row and two bytes for the status information. The value HEX(FFFF) is reserved to mean "row empty" in the extended control vector format. In addition, the work-variable requirements increase to 4 bytes per row plus 3 bytes (to ensure an aligned work area is always available).

4.6.8 Expanding Array Size

Arrays can be expanded beyond their initial allocation with the use of the MAT REDIM statement, provided the following two conditions are met:

1. There must be sufficient memory to allocate both the old variable and the new larger allocation.

2. There may not be any pending /POINTER or stack references to the variable.

If either condition is not met, an error 304 - Cannot expand array variable (which is a recoverable error) occurs.

Newly allocated space in the variable contains either blanks (if a string) or zeroes (if numeric). Previously allocated space is unchanged.

This extended capability of MAT REDIM also applies to implied or explicit array redimensioning operations in the matrix math statements. Refer to the NPL Statements Guide for other matrix operations with the MAT statement.

The scope of the statically allocated array may be either PUBLIC, module private, or local to a function (refer to Section 4.8.5 for more information on "scope.")

4.7 Numeric and String Expressions

The following section details the use of numeric and string expressions in NPL.

4.7.1 Numeric Operators

The following is a list of the numeric operators in order of priority:

Operator	Description
^	Exponentiation
-	Negation (unary minus)
* , /	Multiplication, division
+ , -	Addition, subtraction

For example:

```
A = B + C / D
```

In the above expression, C / D is evaluated first and then added to B, because the division sign "/" has higher priority than addition.

4.7.2 Numeric Expressions

A numeric expression may contain a series of terms separated by arithmetic operators. A term may consist of scalar variables, array elements, constants, functions and other expressions, all of which must be of the numeric type. Evaluation of expressions is from left to right except where affected by operator priority. The order of evaluation may be changed by using parentheses.

```
A = ( B + C ) / D
```

Continuing the example from the previous section, adding the parentheses changes the order of evaluation so that B + C is evaluated first, then the result is divided by D.

The operands for numeric functions may themselves be numeric expressions. Here, the evaluation of these nested expressions starts with the innermost expression. Where more than one expression exists at the same level of nesting, evaluation proceeds from left to right, following the rules stated above.

For example:

```
X = MAX( A * 2 , ( B - C ) / D )
```

consists of four numeric expressions which are evaluated as follows:

1. A * 2
2. (B - C)
3. Result of (B - C) divided by D
4. MAX of the results of expressions 1 and 3

In addition, the numeric functions may also be part of a numeric expression.

For example:

```
X=5*SQR(2)
```

where the SQR(2) (square-root of two) function is part of the numeric expression, multiplied by five.

The following is a list of numeric functions that can be used in a numeric expression:

ABS(n)	ARCSIN(n)	ARCCOS(n)	ATN(n)	COS(n)	ERR
EXP(n)	FIX(n)	#GOLDKEY	#ID	INT(n)	LEN(...)
LGT(n)	LOG(n)	MAX(...)	MIN(...)	MOD(m,n)	NUM(...)
#PART	#PI	POS(...)	RND(n)	ROUND(m,n)	SGN(n)
SIN(n)	SPACE	SPACEK	SPACEP	SPACEV	SQR(n)
TAN(n)	#TERM	VAL(...)	VER(...)		
#FIELDLENGTH		#FIELDSTART	#RECORDLENGTH		

4.7.3 Matrix Support

The NPL MAT statement provides extensive support for both one and two dimensional arrays. Refer to the Statements Guide for more information on the MAT statement.

4.7.4 String Operators

String operators operate differently from numeric operators. String operators are strictly evaluated left to right, and have no order of priority assigned. The following is a list of operators:

&	Concatenation
ADD	Binary addition
ADDC	Binary addition with carry
AND	Bitwise logical AND
*BOOLx	Boolean operations
DAC	Decimal addition with carry for BCD data
DSC	Decimal subtraction with carry for BCD data
OR	Bitwise logical OR
SUB	Binary subtraction
SUBC	Binary subtraction with carry
XOR	Bitwise logical XOR

* where x is any hexadecimal digit 0-F.

4.7.5 String Expressions

In NPL, there are two distinct string expressions which are handled separately. These include string concatenation and the order of evaluation of alpha-operators.

The string concatenation combines two string operands into a single string, using the concatenation operator "&". String concatenation is handled specially and may not be combined with any other alpha operator except itself.

A string expression may contain any number of alpha-operands and alpha-operators. An alpha operand may be a string-literal, an alpha-variable, a string-function or a system variable. The expression is always evaluated from left to right on an operator-by-operator basis. The result of the first operation is placed in an alpha-receiver and each subsequent operation is conducted left to right on the subsequent new values of the receiver. The order of evaluation cannot be changed with parentheses.

For example, in the statement

```
A$ = B$ AND C$ OR D$
```

the contents of the alpha operand B$ is first assigned to the alpha receiver A$. C$ is then ANDed with A$ and the result is ORed with D$. The final result is in A$.

The first alpha operand after the assignment operator is optional in all string expressions except those involving concatenation.

For example, in the statement

```
A$ = AND B$ OR C$
```

the contents of A$ is ANDed with B$ and the result is ORed with C$. This statement is equivalent to A$ = A$ AND B$ OR C$.

If more than one alpha receiver is specified, then the assignment is made to the last receiver first, with the evaluation proceeding as if several assignment statements were made in series.

Some examples of string expressions include:

```
A$ = "ONE" & HEX(2B) & STR("**TWO**",3,3)  = "ONE+TWO"
A$(1) = STR(A$,1,3) OR 'a_string_fnc$
A$, STR(B$(1),3,4) = "fred" AND HEX(1010) XOR "sam"
```

4.7.6 Data Conversion

The CONVERT statement allows general conversion between displayable alpha to numeric or numeric to alpha. Thus, NPL provides an easy why to convert between numeric and string expressions.

For example:

```
0010 CONVERT Price$ to P
```

This line translates ASCII digits in Price$ into a single number and stores it in the variable P.

The BIN and VAL functions allow conversion between binary alpha and numeric data types also.

Data can also be converted using $PACK/$UNPACK statements. Refer to the Statements Guide for proper use of these statements.

4.8 FUNCTIONs and PROCEDUREs

A PROCEDURE is a set of NPL statements designed to perform a specific task, given a number of parameters. A FUNCTION is a special type of PROCEDURE, for which the specific task is to determine a RETURN value of either numeric or string type.A FUNCTION reference can be included within any expression which would otherwise use a constant of the same type (numeric or string.) A PROCEDURE is called by name in a single statement, and has no return value. A FUNCTION or PROCEDURE may have values passed to it in the form of a parameter list, which immediately follows the FUNCTION identifier.

The following is an example of a FUNCTION:

```
0010 FUNCTION 'Highest(Var1,Var2)
    :   IF Var1>Var2
    :     RETURN(Var1)
    :   ELSE
    :     RETURN(Var2)
    :   ENDIF
    : END FUNCTION
```

The following is an example of its use in an expression:

```
0100  PRINT 'Highest(A,B)
0110  BigValue=10+'Highest(Value1,Value2)*0.5
```

The following is an example of a PROCEDURE:

```
0010 PROCEDURE 'Chars(Byte$1,Number)
    :   FOR I=1 TO Number
    :     PRINT Byte$;
    :   NEXT I
    :   PRINT
    : END PROCEDURE
```

The following is an example of its use as a statement:

```
0100 PRINT "Main Menu"
    : 'Chars("=",9)
```

4.8.1 Parameter Types of Functions

Parameters to the function specify the names of variables which receive values required by the function. If the identifier is preceded by an underscore "_", the parameter is treated as a constant value in the function body and modifications to the parameter are inhibited (as with constant variables, each reference to the parameter must include "_" preceding the identifier).

NOTE: All parameters of a function are function private /RECURSIVE class variables.

4.8.2 Parameter Passing Conventions

Function parameters with the /POINTER keyword are passed by reference. All other parameters are passed by value (a copy of the value is made at the time of the call.) String parameters which are passed by value (not /POINTER type) must have a declared maximum length. This length is evaluated once only at resolve time. If a string argument to the function exceeds this length, the value received by the parameter will be truncated to the maximum length. Array parameters passed by reference may not specify dimensions. Array parameters passed by value can only accept arguments with identical dimensions.

4.8.3 Parameter Specifying Modifiable Arguments

The values of Function parameters which both have the /POINTER keyword and are not constant may be used both as input and output in a function. Changes made to parameter variables of this kind are made to the argument used in the corresponding function call. For this reason, values specified as arguments for this type of parameter must be modifiable scalar or array variables, array elements, or STR() functions. Constants, evaluated expressions and constant class variables are not permitted as arguments to this type of parameter.

4.8.4 Passing Large Arguments by Reference

Passing a large argument by reference (using /POINTER) can be more efficient than by passing by value. This is possible because unlike passing arguments by value, passing by reference does not involve copying the arguments when the function is called. Therefore, passing by reference may be faster and less memory-intensive.

If the parameter is passed by reference only for reasons of efficiency, it is good programming practice to declare the parameter as a constant (preceded by a "_") so that it is clear that the parameter is not modifiable (changes to the argument are not expected within the function body). This also avoids the requirement that arguments for the /POINTER parameter must be modifiable variables.

Passing a parameter by reference could have side effects if other functions (called from inside the function body) were able to modify the value of the argument, for example if the argument were a /STATIC or /PUBLIC variable.

Declaring a string parameter as both constant and /POINTER allows arguments to the function to be either a string variable of any length or a literal-string, which may be desirable in many cases.

4.8.5 Scope of Functions

The scope of a function refers to the range of statements within a program where it is permitted to use a function name, and have it refer to the same function body.

It is possible within a program to declare the same function name for different purposes, either by accident or design. For example, a function called "Initialize" might be defined in each of several modules. In such cases, each reference to the variable by name is at least potentially ambiguous. However, by clearly defining the scope of each declaration of a function in such a way that one declaration takes precedence, this potential ambiguity disappears.

A function declaration is said to be "in scope" at a given point in a program if a reference to the function is legal there and refers to the function intended (not to another function with the same name declared for a different purpose).

NPL supports two function scope types: module private, and public.

4.8.6 Module Private Functions

Module private Functions are functions declared without the /PUBLIC keyword and outside of a public section. Only one module private function of a given name may be defined in the same module.

A module private function is in scope only inside the module in which it is declared, provided the declaration of the function precedes the reference.

To avoid any possible ambiguity, it is advisable to position /FORWARD declarations of all module private functions immediately after any PUBLIC sections, INCLUDE and USES statements in the module.

4.8.7 Public Functions

Public functions are functions declared with the /PUBLIC keyword. When a function declaration occurs within a PUBLIC section, it is /PUBLIC by default. Only one public function of a given name may be defined in the workspace (all modules).

A public function may be in scope in any module in the workspace, provided the declaration of the function precedes it in the module, an INCLUDE or USES statement which declares the function precedes it and no module private function with the same name is in scope.

To avoid any possible ambiguity, it is advisable to position /PUBLIC function declarations statements, INCLUDE statements and USES statements before any other types of function declarations in the module.

4.8.8 Scope of Variables Declared within Functions

Parameters and all variables declared with a DIM statement within a function may only be referenced within the body of that function.

4.8.9 Coding Restrictions

The following section discusses coding restrictions for FUNCTIONs and PROCEDUREs.

Location of the Body of a Function

The /FORWARD or /EXTERNAL keywords in a FUNCTION or PROCEDURE statement indicate that this function header is a prototype only, used to declare parameter types and interface specification so that these may be checked at resolve time. No function body (or END FUNCTION or END PROCEDURE statement) follows the prototype.

If the /FORWARD keyword is used, the actual function body must appear later in the module, preceded by a second function header statement. The second function leader statement must contain either a complete set of parameters (all types must match) or the keyword /BEGINS, which indicates that parameter types must be previously declared by a /FORWARD declaration.

If the /EXTERNAL keyword is used, the FUNCTION is implemented externally and no later declaration need appear in the module. At resolve time, all /EXTERNAL declarations are checked to ensure that the named FUNCTIONs are available and have conforming parameter and return types. The /EXTERNAL FUNCTION declarations are required. It is not possible to access functions in an external library unless an appropriate /EXTERNAL function declaration is "in scope".

A function body or prototype must always precede any use of the function in a module.

Branching Restrictions in a Function

While executing the function, branches to outside the body of the function (using GOSUB' or function calls) are permitted, but parameters and other /RECURSIVE or /STATIC variables which are declared within the function body can only be referenced within the body of the function.

Branches into the body of a function from outside (using GOTO or GOSUB) are not permitted. These are flagged as errors at resolve time.

Nesting of Function Bodies

A second FUNCTION or PROCEDURE declaration may not occur within the body of a FUNCTION or PROCEDURE.

Function Body Skip-over

It is not possible for execution to "fall into" a function body. If a function declaration is encountered during execution, the function body is skipped and execution proceeds following the matching END FUNCTION (or END PROCEDURE) statement.

DEFFN' Declarations

DEFFN' statements are not permitted in the body of a function.

4.9 Calling a Defined Function

The function call interface adds flexibility in the way information is passed from a calling program to a function. Information can be passed as a string or numeric, scalar or array, and can be passed directly by value, or indirectly as a pointer.

4.9.1 Argument Type Checking

Arguments must agree in number, order, and type with the parameters declared in the corresponding function header. These values are checked at resolve time for direct calls, and at execution time for indirect calls.

4.9.2 Use of Constant Parameters

Parameters which begin with and underscore "_" are treated as constant values for the duration of the function (changes to the value of the parameter are inhibited within the body of the function).

4.9.3 Use of /POINTER Parameters

Arguments without the /POINTER keyword are passed by value at the time of the function call (a copy of the value is made). Arguments with the /POINTER keyword are passed by reference. In this case, any reference to the parameter refers to the argument directly.

Restrictions on Arguments for POINTER Parameters

If a parameter in the function header is declared as a (non-constant) POINTER type, the corresponding argument must be a variable (numeric expressions, literals, and constant class variables are not permitted and are flagged at resolve time as errors). Array elements and STR() functions are permitted. This restriction does not apply to constant parameters which are POINTER type.

Arguments are evaluated left to right. All arguments are evaluated before the function call is made.

4.9.4 Resolution of Function Identifier

Directly named function calls may reference either module private function or public functions.

NOTE: In situations where both module private and public functions are declared, only one can be in scope.

4.9.5 Indirect Specification of Function Name

Public function names may also be specified indirectly by entering alpha-variable instead of an identifier. At execution time, the alpha-variable must contain the identifier of a valid public FUNCTION or PROCEDURE.

NOTE: Module private FUNCTIONs and PROCEDUREs may not be called in this manner.

The identifier stored in the alpha-variable is not case-sensitive. If it refers to a function returning a string, it may end in a "$". The alpha-variable expression must be followed by a "$" (after the right bracket.)

For example:

```
0327 FUNCTION 'Ignore$ /PUBLIC
   :   RETURN(" ")
   : END FUNCTION
   : INP_Handler$ = $NAMEOF(FUNCTION Ignore)
   : INP_String$ = '<INP_handler$>$
   : ...
```

NOTE: When function names are specified indirectly, type checking of parameters is not done until execution time, and there is a performance penalty.

4.10 Modules

NPL program modules provide a way of compartmentalizing the operations performed in the NPL workspace. Each NPL workspace may be subdivided into many modules. The number of such modules and the size of each one is controlled implicitly by the application.

4.10.1 Module Concepts

Modules have the following characteristics:

- Identifier names (variables, FUNCTIONs/PROCEDUREs, and DEFFN's) are local to the module (unless defined as public). Thus, different modules can define the same identifier name without conflict with identifier names used by other modules.

- Line numbers are also local to the module, thus allowing different modules to use the same line number range without conflict.

- Each module may contain a series of PUBLIC and private FUNCTIONs, PROCEDUREs, DEFFN's, and variables. A PUBLIC section can be used to define all PUBLIC entities.

- Control is transferred from one module to another by issuing a FUNCTION, PROCEDURE, or DEFFN' call to a defined PUBLIC entity of the same type.

- Data is transferred from one module to another either by the use of PUBLIC variables or by the use of function parameters. Large blocks of data can be passed by the use of POINTER parameters.

- Modules are loaded automatically by the RunTime as required by the application. Once loaded, the module remains resident as long as subsequent programs reference it. An overlay in one module does not require that other modules be resolved, thus substantially improving load and resolve time.

- Modules may be independently scramble-protected. The fact that one module is scramble-protected does not prevent debugging or development in other modules.

- Each module may contain procedures that are automatically executed when the module is first loaded or just prior to being unloaded.

- Immediate mode commands operate only on the currently Executing Module or the currently selected List Module. Other modules may be present but are invisible to immediate mode commands. The List Module can easily be changed in immediate mode by use of the MODULE command.

The module concept allows development of true "black box" routines. Once a group of functions is fully developed and debugged, it can be incorporated as a module and can be used by any application with no concern about identifier name conflicts or line number conflicts. All the calling application needs to know are the entry points and the calling conventions (parameter lists and return values). Thus, NPL developers have the ability to develop true application-independent libraries written entirely in NPL.

4.10.2 Module Categories

To understand how to work with modules, it is necessary to define four categories for modules--Root, Run, Execution, and List. Only one module may be in each category at a time. The same module may be (and often is) in more than one category at the same time.

Root Module

The module in which the boot program is originally loaded. The Root Module has no module name assigned to it.

Run Module

The module in which the next RUN command will begin execution. This module is typically the Root Module.

Executing Module

The module in which the next program statement is to be executed. The Executing module will always be the Run Module except when control has been transferred to another module by means of a FUNCTION, PROCEDURE, or DEFFN’ call and program execution is halted by a HALT key, STOP statement, END statement, or unrecovered error prior to completion of the subroutine.

List Module

The module to which any operations affecting program text are to be directed. The List Module will always be the same as the Executing Module except when explicitly changed by use of the immediate mode MODULE command, or if a resolution error occurs in a module other than the Run Module (List Module is set to the name of the module where the resolution error occurred). Any change made to List Module is temporary. List Module will be changed to Executing Module as soon as program execution is begun or resumed.

The current Run Module, Executing Module, and List Module names are displayed by the LIST DT command.

4.10.3 Choosing Module Names

Module names are not case-sensitive and should be chosen with care. Any subsequent INCLUDE which references the same module name does not reload the module, even if the reference filename or device is completely different.

For example:

PROGRAM1:

```
0010 INCLUDE T "LIBRARY"
    : LOAD T "PROGRAM2"
```

PROGRAM2:

```
0010 INCLUDE T#2,"BIGLIB" TO "Library"
```

The device and filename in the INCLUDE statement in PROGRAM2 are effectively ignored, since a module called "LIBRARY" is already loaded.

NOTE: If the program text of a module has been deleted in immediate mode (from CLEAR P), or marked as invalid when the module loads but cannot be resolved, any future INCLUDE statements will reload the specified program file (any remaining text and non-common variables in the INCLUDEd module are deleted).

4.10.4 Reviewing Loaded Modules

The $PROGRAM and "Program Load Sequence" information displayed in LIST DT applies to the current RUN module only, and does not show modules loaded from INCLUDE statements.

NOTE: The Executing Module is not displayed unless the program is currently HALTed and could be CONTINUEd or STEPped.

Following the Program Load sequence, LIST DT displays all modules (except the root module) which are currently loaded, in the format shown below:

```
INCLUDE T/xxx, "FILENAME" TO "ModuleIdentifier" :status[:RUN][:COM]
```

Here, the T/xxx and FILENAME fields identify the device and filename used to initially load the module, and ModuleIdentifier is the internal module name.

The "status" field displays the current modification state of the module and is one of the following keywords:

LOAD	Module unchanged since loaded from INCLUDE
ERROR	Loaded by INCLUDE but could not be resolved
MERGE	Module overlaid or otherwise modified since load
CLEAR	Module is clear of program lines or
SAVE	Module unchanged since last saved (in full)

The RUN indicator shows that the module is currently resolved (public functions may be called either indirectly or directly, if the module is currently INCLUDEd).

The COM indicator is shown for modules which have common variables declared. This prevents them from unloading automatically.

4.10.5 Modifying an Active Module

If a program has been RUN that includes a library module (e.g., "LIBRARY"), and it is determined that a change to "LIBRARY" is required, the correct procedure is:

1. Select the currently loaded library as the LIST module using the MODULE command.

   For example:

   ```
   :MODULE "LIBRARY"
   ```

2. Make program changes to the module, and SAVE it. If you want to test that the module resolves correctly, a RUN[,STOP] command will do this, assuming that the root module still INCLUDEs "LIBRARY".

NOTE: RUN also resets the current module to the Root (no name) Module, so if there are errors, the current List Module will have to be reset again.

The usual inclination for programmers accustomed to versions of NPL prior to Release IV is to do a CLEAR P, LOAD the LIBRARY module (without using the MODULE command), fix it (SAVE it) and then try to RUN it to check for syntax errors. This doesn't work properly because:

- The "LIBRARY" module is still loaded and resolved. All PUBLICs will be flagged as duplicates if an attempt to RUN a second copy in the root (no name) MODULE is performed.
- If, after making the change, you do a CLEAR P, reload the normal root module and try to RUN it then the changes made to "LIBRARY" are not detected. The old module is still running!

4.10.6 Nested Included Modules

Modules can INCLUDE other modules, but INCLUDE statements do not permit circular references. For example, the following INCLUDEs would be considered errors:

```
(Module A)
0010 INCLUDE T "B"

(Module B)
0010 INCLUDE T "C"

(Module C)
0010 INCLUDE T "A"
```

However, repeat references are permitted. The following example is allowed:

```
(Module A)
0010 INCLUDE T "B"
```

```
    :   INCLUDE T "C"
    :   INCLUDE T "D"

(Module B)
0010 INCLUDE T "C"

(Module C)
0010 INCLUDE T "D"
```

In the above example, only one module C and one module D are loaded.

4.10.7 Module Initialization and Cleanup

If a library module has initialization that must be done before other functions can be called, a procedure should be declared with the /MAIN attribute which performs this initialization. When a module is first INCLUDEd, the /MAIN procedure is executed immediately.

If a library module must be cleaned up before it is deleted from the workspace, a procedure should be declared with the /EXIT attribute. The /EXIT procedure is executed automatically any time a module is about to de-resolve, either because it is about to be deleted or modified.

Make the logic in /MAIN and /EXIT procedures as simple and failure-proof as possible.

Exiting the runtime with $END de-resolves all modules and ensures all /EXIT procedures are run.

4.10.8 Discarding Modules

Modules are checked for discardability at the end of a RUN (or program overlay) and after a CLEAR (with no parameters). The module is discarded if all of the following conditions are met:

- No other modules INCLUDE it anymore (or have RUN it).
- It has no common (COM) variables.
- It does not contain unmodified changes, i.e., it has not been modified by overlays or immediate mode changes since it was initially created, it has been SAVEd in its entirety since such changes were made, or the program text has been cleared from memory or is invalid.

The second condition is intended to make a module stay resident more easily.

The third condition is intended to avoid deleting modules that have been changed but not saved during program development.

NOTE: If a group of overlaid programs all use a particular library, each must have the INCLUDE statement.

Values assigned to /PUBLIC variables survive only as long as the defining module remains resolved.

4.10.9 Effect on Immediate Mode Commands

The effect of the current Run Module, Executing Module, and List Module on immediate mode commands can be summarized as follows:

- All references to variables will refer to variables within scope at the point of execution within the current executing module. For example, assignment statements:

  ```
  A$="XYZ"
  X=Y
  ```

  Display statements that display the value of a variable:

  ```
  PRINT
  LIST DIM
  LIST DIM*
  LIST STACK DIM
  ```

- All references to executable subroutines (DEFFN's, FUNCTIONs, PROCEDUREs, GOSUB #) refer to subroutines within scope at the point of execution within the current Executing Module.

- All references to program text will refer to program text in the current List Module. This includes:

  Line editing
  LIST (text), LIST D, LIST V, LIST #, LIST ', LIST T
  CLEAR [P,V,N]
  RENUMBER
  RENAME V
  STEP #

- Debug commands (STEP, TRACE) refer to all modules. Refer to Chapter 6 of the Programmer's Guide for further details.

- Immediate Mode LOAD, LOAD BOOT, SAVE, SAVE BOOT, and RESAVE commands refer to the current List Module. An immediate mode LOAD RUN statement executes in the current Run Module.

WARNING--Use of DEFFN' literals to perform program SAVEs can give unexpected and unpleasant results if the program is currently resolved (HALTed while running), since DEFFN' functions are always referenced in the currently EXECUTING context.

4.10.10 Execution Time LOAD and SAVE

Execution time LOAD statements refer to the current Run Module. That is, the overlay operation (i.e, non-common variables cleared, lines cleared, and lines loaded) occurs in the Run Module.

Execution time SAVE, and RESAVE statements refer to the current Executing Module.

4.10.11 Guidelines for Writing Modules

For new programming, use FUNCTIONs and PROCEDUREs, not DEFFN's.

Use the option ($OPTIONS byte 38=HEX(01)) that requires all variables to be dimensioned.

For maximum reliability, within FUNCTIONs and PROCEDUREs use local variables. In general, the more local and private the scope of a variable the better, since what other procedures cannot see, they cannot change. Consequently, if an error is determined to be related to a wrong value in a specific variable, there are fewer variables to examine.

Use local (recursive) variables where possible, and local STATIC variables only if information clearly needs to survive from call to call. Non-local variables should only contain information that is truly one of a kind, and should never be used as a convenient way of passing "hidden" information between functions.

When a variable of unknown maximum size is required, make it the responsibility of the caller to declare it and pass it to the library routines. If the caller doesn't know how big something should be, provide library functions that will return this information, so that they can be used in the caller's DIM statements.

4.11 Logical Constructs

NPL Release IV introduces several block-oriented constructs that can be used to manage program logic in a more structured fashion. This structured design is both easier to maintain and more familiar to programmers trained in other structured languages such as C or PASCAL. All structured constructs consist of a statement to define the start of the logical block and a statement to define the end of the logical block.

All structured loop constructs support the use of LOOP and BREAK statements. A LOOP statement bypasses execution of the current iteration of the loop and transfers control to the loop evaluation statement. A BREAK statement terminates the loop entirely and transfers control to the statement immediately following the loop terminator (WEND, UNTIL, or NEXT).

4.11.1 Structured IF...ELSE...ENDIF

The structured IF...ELSE...ENDIF is a structured statement block allowing conditional execution or branching based on the evaluation of a logic expression. Any number of statements may be in an IF block. The block of statements to be executed is determined by the block terminator which is either an ELSE statement or an ENDIF statement. Use of DO/ENDDO is not required.

There are no rules regarding use of line numbers within IF blocks. All statements may be on a single line, each statement may be on a separate line, or any mixture of the two is allowed.

The structured IF statement is distinctive (compared to older forms of the IF statement) in that it does not require or permit the THEN keyword. Only a logical expression follows the IF keyword.

Some examples are:

```
0010 IF X=Y
    :   PRINT "x = y"
    :   X-=Y
    :   PRINT "x is now zero"
    :END IF
    :random_num = RND( seed )
0020 IF random_num < 0.5
    :   PRINT "moving left or right"
    :   IF random_num < 0.25
    :     'move_left
    :   ELSE
    :     'move_right
        END IF
    : ELSE
    :   IF random_num > 0.5
    :     PRINT "moving up or down"
    :     IF random_num < 0.75
    :       'move_up
    :     ELSE
    :       'move_down
    :     END IF
    :   ELSE
    :     PRINT "random number = 0.5, not moving"
    :   END IF
    : END IF
```

4.11.2 WHILE...WEND

The WHILE...WEND is a structured loop. The WHILE statement evaluates an expression and executes the loop body until the expression evaluates to false.

For example:

```
0010 I=0
    : WHILE I<10
    :   'any_func(I)
    :   I+=1
    : WEND
0020 WHILE TRUE
    :   char$='read_a_char(Eof)
    :   IF Eof<>0 THEN BREAK
    :   PRINT char$
    : WEND
```

The loop in line 10 continues to execute while I is less than 10, thus the highest value passed to 'any_func(I) is 9. Line 20 uses the TRUE keyword, which essentially continues the loop indefinitely. The loop is only broken with the BREAK statement.

4.11.3 REPEAT...UNTIL

The REPEAT..UNTIL is also a structured loop. The loop body following the REPEAT statement is executed until an expression evaluated by the UNTIL statement is true. The REPEAT...UNTIL loop always executes at least once.

For example:

```
0010 I=0
    : REPEAT
    :   'any_func(I)
    :   I+=1
    : UNTIL I> 79
```

This loop is a rewritten version of the WHILE...WEND example. Again, the highest value passed to 'any_func(I) is 9.

4.11.4 FOR/BEGIN...NEXT

The FOR/BEGIN...NEXT is a structured loop with a clearly defined entry and exit point. The statements contained within the loop's body are executed a specified number of times.

If the parameters of the FOR/BEGIN statement indicate that the initial value exceeds the final value (for a positive STEP value) or is less than the target (for a negative STEP value,) the loop body will not be executed.

For example:

```
0010 FOR Counter=1 TO 10 BEGIN
    :   PRINT Counter
    : NEXT Counter
0020 FOR Fraction=0.0 TO 0.5 STEP 0.1 BEGIN
    :   Integer_part=Fraction*10
    :   PRINT Integer_part
    : NEXT Fraction
0030 I=-0.35
    : FOR row_index=1 TO max_row_index BEGIN
    :   J=0.4
    :   FOR col_index=1 TO max_col_index BEGIN
    :     array(row_index,col_index)=I*J
    :     J+=0.8
    :   NEXT col_index
    :   I-=0.7
    : NEXT row_index
```

4.11.5 SWITCH...CASE

The SWITCH/CASE structure is used where a decision results in more than two mutually exclusive cases. The structure provides multi-decision branching and can be used in place of ON...GOTO, ON...GOSUB or complex IF statements. Three forms of SWITCH are provided: String, Numeric, and Logical. Numeric and String forms may use expressions on both the SWITCH and CASE statements.

Some examples are:

```
0010 SWITCH char$
    :   CASE "a"
    :     PRINT "a"
    :   CASE "b", "c"
    :     PRINT "b or c"
    :   CASE        : ; default case must always be last
    :     PRINT "Not a,b or c"
    : END SWITCH
0020 REM Numeric example
    : SWITCH 'any_numeric_fnc
    :   CASE 3,5,7
    :     PRINT "odd number"
    :   CASE 2,2*2
    :     PRINT "even number"
    :   CASE
    :     PRINT "value too large"
    : END SWITCH
0030 REM Alpha example
    : SWITCH last_name$
    :   CASE "Brown"
    :     SWITCH first_name$
    :       CASE "Bob"
    :         PRINT "Bob";
    :       CASE "Bill"
    :         PRINT "Bill"
    :       CASE "Betty"
    :         PRINT "Betty"
    :       CASE
    :         PRINT "Don't know any Browns with first name ";first_name$
    :     END SWITCH
    :   CASE "Green"
    :     char$ = STR(first_name$,1,1)
    :     REM Logical example
    :     SWITCH
    :       CASE char$ = "A"
    :         PRINT "First name starts with A"
    :       CASE char$ < "M"
    :         PRINT "First name starts with letters B thru L"
    :       CASE
    :         PRINT "First name starts with letters M thru Z"
    :     END SWITCH
    :   CASE
    :     PRINT "Don't know any one with the last name ";last_name$
    : END SWITCH
```

4.12 RECORDs/FIELDs

The use of RECORDs/FIELDs gives NPL programs the ability to define logical record layouts. RECORD/FIELD have the following characteristics:

- Each RECORD layout is defined by a block of statements starting with a RECORD statement and ending with an END RECORD statement.
- A logical RECORD can be used for multiple record buffers. The logical RECORD itself allocates no storage space.
- Any number of fields can be defined within a logical record.
- Field characteristics (type, length) can be defined using traditional $PACK specifications or by mnemonic terms using the PCKFIELD module provided by Niakwa. Mnemonics for $PACK specifications and Niakwa Data Manager field types are supported.
- References to FIELDs cause implicit data conversion to occur. Use of $PACK/$UNPACK can be greatly reduced or eliminated.

The following statements support use of RECORDs and FIELDs:

$FIELDFORMAT returns the $PACK specification for the specified field.

#FIELDLENGTH returns the length of the specified field.

#FIELDSTART returns the starting position within the logical RECORD for the specified FIELD.

LIST RECORD, LIST FIELD statements.

#RECORDLENGTH returns the length of the specified logical RECORD.

Indirect reference for FIELD variables is supported. That is, the name of a field can be contained in a variable. This, in conjunction with modules, permits a very high level of abstraction within data base applications.

4.12.1 Field Identifiers

A field identifier is a variable type which provides easier examination and modification of values in logical data records. Associated with each field identifier is a relative offset, length and format specification which defines a field in a logical record. These characteristics can be described using $PACK specifications or by the mnemonics in the PCKFIELD module provided by Niakwa.

For example:

```
10 INCLUDE T "NPLDEV"
 : INCLUDE T#_NPLDEV,"PCKFIELD"
 : USES PackFormats
20 RECORD a_record
 :    FIELD letters$10
 :    FIELD characters$ = HEX(A00A)
 :    FIELD string$ = 'StringLength$
 :    FIELD name$(10)25
 :    FIELD initials$(10)=HEX(A003)
 :    FIELD age = HEX(B003)
 :    FIELD address$ = 'FieldType$( _PACK_ALPHA_STRING_FORMAT,0,4)
 :    FIELD numbers(10) = 'FieldType$( _PACK_SIGNED_BINARY_FORMAT,2,5 )
 : END RECORD a_record
```

defines the logical record "a_record".

Field identifiers are always entered and displayed preceded by a period. When a field identifier is appended to a string variable, NPL treats that string variable as a record of the appropriate type. The field identifier becomes equivalent to a single field of that record. For example, given the previous record definition:

```
DIM record_buffer$#RECORDLENGTH( a_record )
record_buffer$.name$(1) = "John Black"
record_buffer$.age = 30
```

When assigning a value to a string field, string concatenation (&), logical string operations (AND, OR, BOOL, ADD[C] etc), BIN() and ALL() functions are permitted on the right hand side of the equal sign. For example:

```
record_buffer$.letters$ = 'MyFunc$(X$)
```

A multiple string assignment may include a number of string field specifications on the left hand side of the equal sign, e.g., the following is legal:

```
MyRec$.field1$, MyRec$.field2$ = ALL("X")
```

As is the case with non-field string assignments, each member on the left hand side of the equal sign is treated separately.

For example, the above is equivalent to:

```
MyRec$.field2$=ALL("X")
MyRec$.field1$=ALL("X")
```

When assigning a value to numeric a field-expression, any valid numeric expression is permitted on the right hand side of the equal sign.

For example:

```
MyRec$.field1=12
```

Multiple assignments are not permitted with numeric fields. Only one numeric field may be specified on the left hand side of the equal sign of a numeric field-expression assignment statement.

4.12.2 Field Expressions

String field-expressions can be used in any context where a STR() function of a buffer alpha-variable is acceptable.

For example:

```
BufferName$.fieldname$
```

is equivalent to:

```
STR(BufferName$, #FIELDSTART( fieldname$), #FIELDLENGTH(field name$) ).
```

Consequently, the following syntax is allowed:

```
DATA LOAD DC #1,Buffer$.String$
'ProcName(Buffer$.String$) ;ok even if parameter is /POINTER
$UNPACK(F=Y$)B$ TO Buffer$.String$
MAT SEARCH Buffer$.String$,=Value$ TO L$()
LOAD T<n> Buffer$.String$
PRINT TO Buffer$.String$;X;
DEFFN'Doit(Feeble$.Mindedness$)
DATA LOAD BA T#1,(Rec,Buffer$.Next$)Buffer$.Sector$
CONVERT X TO A$.Output$,(###)
```

A numeric field expression may be used:

- In the special numeric field-expression assignment statement, and
- Wherever any numeric value (general numeric expression) is permitted.

However, it is not always legal to replace a numeric variable with a numeric field-expression.

A numeric variable is a special type of expression that can be used in a number of contexts where not all numeric expressions are allowed. By comparison, a numeric field expression is also always an expression, but it can only be used in a context where all numeric expressions are allowed. Since NPL performs special handling to store a numeric value into a numeric field, a numeric field expression does not qualify as a variable.

For example, the following is illegal:

```
FOR Buffer$.Invoices$(2).Amount=1 TO 10
```

A simple test can be performed to help decide when a numeric field expression can be used in cases other than simple assignments to the field. The numeric field expression can be replaced by another non-variable expression, such as SQR(2).

For example:

```
Buffer$.Invoices$(2).Amount=SQR(2)      :;works great
FOR SQR(2)=1 TO 10                      :;generates syntax errors
```

The following examples would also be flagged as illegal syntax.

```
DEFFN'Doit(Feeble$.Number)                  :;illegal field!
$UNPACK(F=Y$)B$ TO Buffer$.Number           :;illegal field!
DATA LOAD BA T#1,(Rec,Buffer$.Next)Sector$ :;illegal field!
CONVERT X$ TO A$.Output                     :;illegal field!
```

Field identifiers offer no new functionality that $PACK and $UNPACK do not also allow, but provide a more convenient way of accessing related data.

The following statements show the equivalence between assignment of a numeric field and a $PACK statement:

```
Buffer$.field = numeric_expression

format$ = $FIELDFORMAT(.field)
start = #FIELDSTART(.field)
$PACK(F=format$) STR(Buffer$,start) FROM numeric_expression
```

The following statements show the equivalence between evaluation of a numeric field and a $UNPACK statement:

```
numeric_var = Record$.field
format$ = $FIELDFORMAT(.field)
start = #FIELDSTART(.field)
$UNPACK(F=format$) STR(Buffer$,start) TO numeric_var
```

NOTE: The internal format of fields defined in a RECORD is always a data format in which the byte order is well-defined. This means that data defined using RECORD specifications can be accessed on any NPL platform. The resolver does not attempt to ensure that the instance of a record is associated with the named field.

4.12.3 Indirect Field References

Wherever field identifiers are permitted in expressions, the specific field identifier name may also be indirectly entered as <alpha-variable>. To ensure uniqueness the field name must be associated with a PUBLIC record. When the identifier is needed, the specified alpha-variable must contain the name of a valid field identifier (if a string, "$" is permitted at the end of the identifier).

This permits table-driven access to a record specification, at some small cost in performance.

NOTE: Matching FIELDs to RECORD declarations is not enforced, because the record instances are string variables or substrings whose provenance is not easily established.

4.12.4 Example of RECORDs/FIELDs

The following example illustrates some of the characteristics of the RECORD/FIELD implementation.

```
0010 ; AR001 - AR Amount Due Report
0020 DIM TotalAmount,Amount,T$32="Last_Pay"
0030 INCLUDE T/D21,"PCKFIELD"
0040 USES PackFormats
0050 RECORD /PUBLIC cr
   : FIELD name$=HEX(A020)
   : FIELD amount=’FieldType$(_PACK_IBM_PACKED_DECIMAL_FORMAT,2,5)
   : FIELD last_pay=’FieldType$(_PACK_FLOATING_POINT_FORMAT,
     _PACK_BASIC2C_INTERNAL_NUMERIC_FORMAT,8)
   : END RECORD cr
0060 DIM Cust_Record$#RECORDLENGTH(cr)
0065 IF ’Open_File("ARFILE")<> 0
   :    ’Display_Error
   :    ’Exit_Program
   : END IF
0070 WHILE ’Get_Next_Record("ARFILE",Cust_Record$)=0
   :    PRINT Cust_Record$.name$,Cust_Record$.Amount,
        Cust_Record$.last_pay
   : WEND
0080 PRINT "Total ";TotalAmount
0085 ’Close_File("ARFILE")
0090 END
```

Program AR001 illustrates the typical use of RECORDs/FIELDs. The program is a simple program that reads through ARFILE, prints the Name, Amount and Last_Pay fields for every record, and totals Amount. It performs the following tasks:

1. Line 20 dimensions two work variables used by the program. With Long Identifier Names, it is a good idea to explicitly dimension all variables. In addition, variable T$ is dimensioned for later use in illustrating indirect reference to a field.

2. Line 30 includes module PCKFIELD. This is a module provided in the NPL Development Package that contains the ’FieldType function which allows the use of mnemonic definitions of field types for each field.

3. Line 40 specifies that the PckFormats public entities are to be used. This allows use of mnemonics equivalent to standard NPL $PACK formats. Alternatively, public entities could be used, which permit use of mnemonics equivalent to Niakwa Data Manager data types.

4. Line 50 defines a Record and assigns the name "cr" to the record. The use of the PUBLIC keyword allows use of indirect references to fields defined within this record (although this feature is not used in this example).

   Three fields are defined. The Name$ field is an alpha field, 32 bytes in length, as defined by the explicit $PACK specification (A020). Amount and Last_Pay field types are defined by use of the 'FieldType function. Amount is defined as packed decimal, two decimal positions, five bytes in length, equivalent to the $PACK specification (5205). Last_Pay is defined as NPL internal numeric (F108).

5. Line 60 dimensions a record buffer. The #RECORDLENGTH function is used to determine the size of the record buffer.

6. Line 65 calls a function (not shown) to open the ARFILE.

7. Line 70 defines a loop that reads all records in the file. This logic assumes that the function 'Get_Next_Record (not shown) will return zero if successful.

8. After all records are processed, the accumulated total is printed, the file is closed, and the program is ended.

CHAPTER 5

CREATING NPL PROGRAMS AND THE NPL EDITOR

5.1 Overview

NPL program editing is easily performed by the combined use of two different features of the NPL Interpreter: Immediate Mode commands and the Line Editor.

Immediate Mode commands are integral to NPL program editing in performing such tasks as loading and saving programs, listing programs, cross-referencing variables and line numbers, searching for text strings, renumbering and merging of program text and many other functions.

HINT: This chapter discusses the makeup of NPL programs and various methods of creating them, however, an understanding immediate mode is essential to creating and editing NPL programs. Refer to Chapter 2 for a complete discussion of Immediate Mode.

The Line Editor is used for the initial creation of program text and for editing existing program text which resides in memory on a line-by-line basis. The line editor is closely coupled with the Interpreter, and as a result allows for immediate syntax checking of commands and program lines as they are entered. Line editing is enhanced by the use of Special Function keys which aid in the rapid correction of program text in a minimum of keystrokes. This chapter discusses these topics in detail.

Section 5.2 discusses the creation of NPL programs.

Section 5.3 discusses the loading of NPL programs.

Section 5.4 discusses the NPL line editor.

Section 5.5 explains how to edit NPL programs.

Section 5.6 explains how to save NPL programs.

5.2 Creating NPL Programs

This section discusses the creation of NPL programs.

5.2.1 NPL P-Code

An NPL program is defined as any file of codes (called pseudo-code or p-code) which is directly executable by the NPL Interpretive (RTI) or non-Interpretive (RTP) RunTime Package. The special p-code storage format of NPL programs is used due to its efficient memory usage and excellent execution performance.

NPL programs can be created using the NPL Interpreter, the NPL Compiler, and dynamically by using the NPL $SOURCE/$OBJECT program functions.

5.2.2 The NPL Interpreter (RTI)

The most common method of creating, editing and maintaining NPL programs is using the NPL Interpreter (RTI). NPL programs are created by entering new source program lines in Immediate Mode with the line editor.

NOTE: It is advisable to first CLEAR the program text area of program lines which may be resident from a previous edit or RunTime session before entering new program lines.

As the source program lines are entered, they are immediately compiled by the Interpreter into executable p-code and stored in program text memory. The original source line is then effectively discarded. Whenever the original source line must be subsequently viewed (for listing, editing, debugging, etc.) it is regenerated from the stored p-code by a built-in de-compiler.

Once a program has been created in this way, the completed p-code program in memory can be saved to disk using a number of Immediate Mode SAVE commands. These are discussed later in this section.

5.2.3 The NPL Compiler (B2C)

NPL programs can be created with the compiler by reading source program lines from a file, compiling them and then saving the resultant p-code into another file. Using the compiler, the source program storage format can be in one of several different forms. ASCII format, Wang 2200 atomized format and p-code format itself, are all valid input source program formats to the compiler.

NPL programs cannot be directly edited by native operating system editors since they are stored in p-code format and native editors generally operate on ASCII format files. However, the NPL compiler can be used to easily convert p-code into ASCII (on a batch basis), thereby enabling the use of native editors or any other native utilities to manipulate NPL source ASCII files. Once the manipulation is complete, the compiler is used to convert from ASCII back to p-code for execution.

NPL programs are not directly executable on a Wang 2200 since the 2200 operates on a special atomized format. In this case, the compiler could be used to convert p-code programs (on a batch basis) into Wang 2200 atomized format for execution on that machine.

A complete discussion of the compiler is provided in Chapter 14.

5.2.4 $SOURCE/$OBJECT

$SOURCE and $OBJECT are programmable instructions that can be used by an NPL program to create other NPL programs dynamically at execution time. The $OBJECT function receives a specified line of program source (in ASCII) and returns the executable p-code for the given source to a specified variable. $SOURCE performs the opposite function. By saving p-code generated in this way to disk, the resulting p-code program may then be loaded and executed like any other NPL program.

For further details on this method of creating NPL programs refer to the Statements Guide, $SOURCE and $OBJECT.

NOTE: With the addition of Long Identifier Names in NPL Release IV, routines which use $SOURCE and $OBJECT need to be adapted to handle Long Identifiers in the event they are introduced into existing routines. To simplify this process, Niakwa has provided developers with two libraries, SOURCEIO and OBJECTIO. Refer to Chapter 3 of the Statements Guide for a complete discussion of these library routines.

5.3 Loading Programs

Existing NPL programs are loaded into memory for editing purposes by use of the LOAD command. Programs can be loaded from both diskimages and native operating system directories. If a program is currently loaded in memory, subsequent LOADed programs are merged together with the resident program text in that module.

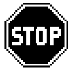
Once a program has been loaded into memory, it may be edited by use of the line editor as discussed in the next section.

NOTE: The LOAD statement as it relates to program editing is discussed in this section. The LOAD statement is used for many tasks in addition to loading programs for editing. For further details refer to LOAD in the NPL Statements Guide.

5.3.1 Program Text Merging

Creation or editing of a program is often assisted by adding existing program text from standard subroutine libraries to the current program text in memory.

This is facilitated by the LOAD command (as distinguished from the LOAD statement). Within the current module, the LOAD command does not clear program text from memory before loading the specified program. Rather, the specified program text is merged with program text currently in memory. Merging is performed on a program line number basis.



WARNING--Care should be taken with this procedure since program lines currently in memory are replaced by program lines of the loading program which have matching line numbers.

Therefore, when merging of program text is not desired, the program text area should be CLEARed before loading new program text.

When program merging occurs, a warning message is generated. This is not an error message, but merely a confirmation that programs have been merged.

For example:

```
:LOAD T "PROGRAM1"
:LOAD T "PROGRAM2"
Warning: Programs merged.
```

NOTE: An immediate mode LOAD command will always merge code into the currently selected LIST module. This will usually be the RUN module, but may also be an INCLUDEd library module.

5.3.2 Loading from Diskimages

A program file is loaded from a diskimage for editing using either the LOAD or LOAD DA command. The LOAD command is the most common method used for loading programs.

The LOAD command loads a program by its name whereas the LOAD DA command loads a program by its starting absolute sector address in a diskimage ("DA" stands for Diskimage Absolute mode).

For example:

```
:LOAD T/D10,"PROGRAM"            Loads a program by searching
                                 the catalog index of
                                 diskimage D10 for program
                                 name "PROGRAM".

:LOAD DA T/D10,(200)             Loads the program starting
                                 at sector 200 of diskimage
                                 D10.
```

A list of all programs within a diskimage can be obtained using the LIST DC command. This command produces a listing of all programs in a diskimage and other pertinent information such as program size, date stamp, time stamp and absolute sector address in the diskimage, etc. The length of diskimage listings can be reduced with optional wildcards and relational parameters to the LIST command.

A list of diskimages available within the native operating system directories can be obtained while in the Interpreter by executing the appropriate $SHELL (or "!") command in Immediate Mode. The command is passed to the native operating system for processing (presumably a directory listing request). The results of the command are then displayed and control is returned to the Interpreter.

For further details, refer to the Statements Guide; LOAD, LOAD DA, LIST and $SHELL.

5.3.3 Loading from Native Operating System Files (BOOT and PREBOOT)

NPL programs may be loaded from the native file system for the limited purpose of a BOOT program. NPL supports two types of boot programs.

A BOOT program is an NPL program stored in a native operating system file system and is automatically loaded and executed by the RunTime.

A PREBOOT program is also stored in a native operating system file system, but is only loaded in response to the NPL /P command line startup option. The PREBOOT program is useful for setting up programming preference options (especially $OPTIONS and then performing a LOAD BOOT).

For example:

```
C:\BASIC2C\RTI BOOT              Execute Standard BOOT Program

C:\BASIC2C\RTI /PSTARTUP         Execute PreBoot Program
```

In addition to directly executing a BOOT or PREBOOT program file from the native operating system command line, these programs may also be loaded directly into the interpreter from the native operating system by using the LOAD BOOT command. When executed, the program name as specified is loaded into memory.

When loading programs from the native operating system file system, the $SHELL command may be useful for locating the program within the native file system or any other preparation required in advance of executing a LOAD BOOT command.

NOTE: **Exercise caution when testing a PREBOOT program. If the PREBOOT program is loaded for editing using the LOAD BOOT command, a subsequent RUN of the program will typically result in an infinite loop, since the LOAD BOOT at the end of the PREBOOT program will reload itself. This can be stopped with the HALT key.**

For further details refer to the Statements Guide, LOAD BOOT and $SHELL. Refer to Chapter 4 of the appropriate NPL Supplement for details on the BOOT and PREBOOT programs or Section 2.4 of this manual.

5.3.4 Loading Program Text at RunTime

Program text which is currently resident in memory and executing may be accessed for editing by suspending program execution through invoking Immediate Mode. Once Immediate Mode is invoked (i.e., the colon prompt is displayed) the resident program text is available for editing.

NOTE: **After editing a program interrupted in this way, execution of the program may only be restarted by the RUN or GOTO commands. Program execution may not be resumed by the CONTINUE command (for further details on Immediate Mode, refer to Chapter 2).**

Further, care should be taken before SAVEing a program edited in this manner. Many NPL applications dynamically merge program text from several distinct program files on disk at run time to form a larger executable program. Therefore, a program in memory may be significantly modified from its original form on disk as a result of this run-time activity. SAVEing this modified form of a program on top of its original form is most likely undesirable.

5.3.5 Loading Modules

This section discusses the process of loading and accessing an NPL program module. Refer to Chapter 4 for more information on NPL modules.

Programmers can load and save modules to and from an NPL diskimage just like any other program. However, modules differ from NPL programs in how they are loaded and edited under program control.

Modules are loaded using the INCLUDE statement within a program. Program overlays are LOADed into a program.

For example:

```
(MODULE "A")
0010 INCLUDE T "B"

(MODULE "B")
0010 INCLUDE T "C"
```

An INCLUDEd NPL program file is loaded into the workspace in an area (a "module") which is separate from the program containing the INCLUDE statement.. As the above illustrates, after running MODULE A, there would be 3 program files loaded into memory as distinct modules in the workspace: MODULE A, MODULE B and MODULE C. Each module resides in it’s own area and does not conflict with other modules in memory. Each module can be selected, edited and saved individually.

This modularity simplifies the development process by reducing the need to rely on complex overlay schemes which can easily mask logic errors. Refer to Chapter 4 for more information on NPL modules.

5.4 The Line Editor

The NPL Line Editor is used to enter and/or edit NPL Immediate Mode commands, Immediate Mode program lines and data values in memory. Immediate Mode commands are generally used for program debugging, and Immediate Mode program lines are generally used for program text editing as described here. For further details on NPL debugging, refer to Chapter 6.

The line editor aids in the entry of text for these purposes by providing such functions as character insertion, deletion, replacement, line insert, line delete and all directions of cursor movement.

The NPL line editor is automatically activated and controls entry of text whenever:

1. The Interpreter enters Immediate Mode. This is indicated by the presence of the colon prompt (":") at the beginning of a line on the display:

   For example:

   ```
   :_
   ```

2. Whenever data input is requested from the keyboard by an executing NPL program (using INPUT or LINPUT):

   For example:

   ```
   INPUT AMOUNT?
   ```

As text is entered with the line editor, each character entered is displayed on the screen and placed in a temporary "edit buffer". This process continues until entry of the text line is terminated by depression of the RETURN key. At this point, the contents of the edit buffer is processed by the appropriate routine as follows:

- If the text line is an Immediate Mode program line (i.e., text is prefixed by a program line number), then the text is checked for syntax errors and is submitted for incorporation into the existing program in memory.

- If the text line is an Immediate Mode command (i.e., text is not prefixed by a program line number) then the text is checked for syntax errors and submitted for immediate execution.

- If the text line is entered in response to an INPUT or LINPUT statement, then the text is analyzed and assigned to the receiving variables of the input statement.

The line editor's use of a temporary edit buffer allows that no changes to the program text area are effected until such time as entry of the full text line is completed and the RETURN key is depressed. This allows for the opportunity to confirm an entry before committing to its intended action.

The size of the edit buffer is important as it represents the maximum text line length that may be entered or edited by the line editor. If editing a program line with the line editor, this would be the maximum program line length that is enterable or editable. The size of the edit buffer is variable since it is dynamically allocated from the current free space in the partition. The current approximate size of the edit buffer is returned by the SPACE function. However, at all points in time, the minimum guaranteed size of the edit buffer is 1024 bytes, regardless of the value of SPACE. For further details on SPACE, refer to the NPL Statements Guide.

It is often the case that a "text line" may exceed the length of a physical line of the display (usually 80 characters). In this case, the line editor simply performs a carriage return and line feed, thereby allowing entry to continue at the beginning of the next line. This is an automatic system function and does not generate characters to the edit buffer. All lines of a text line are accessible by edit functions regardless of how many physical lines are required for one text line.

When entering program text, it may be desirable to force a carriage return and line feed prior to encountering the physical end of line. The LINE INSERT function performs this action. This is particularly useful for documentary purposes when entering program text and for multiple character insertion as described later. For further details on LINE INSERT, refer to Section 5.4.

For example, consider the following two program lines:

```
1000 FOR I=1 TO 1000: PRINT LOG(I): NEXT I

1000 FOR I=1 TO 1000<
   : PRINT LOG(I)<
   : NEXT I
```

Both program lines perform the same function, however the second one is somewhat more readable. Here, a carriage return and line feed were forced after each program statement with the LINE INSERT function. The presence of this is indicated by the return graphic ("< ") at the end of each statement.

Even though the program line requires 3 physical lines it is considered to be only one text line and is stored as such in the edit buffer. Further, the text line is not processed until all three statement segments were entered and the RETURN key depressed.

NOTE: The return graphic is an actual character and does occupy one character position in the edit buffer. The return graphic is subject to rules of syntax and is therefore only legal in certain contexts as discussed in Section 5.4.

5.4.1 Multi-Command Buffering

In addition to containing the most recent command or program line edited, the "edit buffer" may contain up to 20 previous entries, depending on available memory. Once 20 entries have accumulated, the oldest entries are removed from the buffer as new entries are made. If insufficient memory is available to store a new entry, it is not retained. No error or warning is issued when this occurs. All entries in the edit buffer are purged by a CLEAR (no parameters) command. In addition, entries may also be purged if an attempt is made to dimension a variable or load program text that would otherwise generate an A01 - Insufficient memory error.

These entries can be accessed by using the SHIFT PREV key (or SF 18 in Edit Mode) to scroll backwards through the list, or the SHIFT NEXT key (or SF 19 in Edit Mode) to scroll forward.

When SHIFT PREV or SHIFT NEXT are used to scroll between different entries, the command or program line displayed is not executed, and any changes that have been made are not retained.

When the desired entry is located, it may be edited and processed as though it were the current entry. When RETURN is pressed, the entry displayed is then resaved in the edit buffer as the newest entry. The original entry in the edit buffer remains unchanged unless it is purged due to being the oldest entry. Then, if the entry is an Immediate Mode command, it is executed, or, if the entry is a program line, it is submitted for incorporation into the existing program. The cursor is then returned to a colon prompt.

5.4.2 EDIT Functions

In general, the edit functions allow for the correction of text entered with the line editor. These edit functions are activated by one or more keystrokes from the keyboard.

The physical location and labeling of these keys vary depending on the physical keyboard hardware and the keyboard mapping used (user definable). For discussion purposes here, the terminology used for key names refers to the virtual key names as defined on the NPL Virtual Keyboard (refer to Section 7.5.2). To determine the physical location of these virtual keys on the keyboard, refer to the appropriate hardware specific NPL Supplement.

The line editor uses keys from both the standard key group and the special function key group of the virtual keyboard to aid in text editing.

5.4.3 Standard Keys

There are four virtual keys from the standard key group which are relevant to line editing. These virtual keys are active at all times during entry of text with the line editor.

BACKSPACE Depression of the BACKSPACE virtual key deletes the last character entered from the edit buffer, blanks the character on the display, and moves the cursor back one position.If the editor is in insert mode, the last character entered is simply deleted (no blank replaces it).

LINE ERASE Depression of the LINE ERASE virtual key deletes all characters in the edit buffer, blanks all characters entered on the current text line, and returns the cursor to the first position. Effectively, depressing LINE ERASE causes all text on the current text line to be removed so that it may be reentered.

EDIT Depression of the EDIT virtual key switches the mode of the special function key pad from between DEFFN Mode and EDIT Mode. The EDIT key and the special function keys are discussed in the following section in detail.

RETURN Depression of the RETURN virtual key while editing a text line indicates completion of the edit session. After RETURN is depressed, the text line is submitted for processing by the Immediate Mode processor.

5.4.4 Special Function Keys

The majority of the line editor edit functions are implemented through the special function keys (virtual). As defined on the NPL virtual keyboard, there are 32 special function keys. In addition, the line editor implements two "modes" of the SF key pad to allow for 64 possible edit functions.

The first mode is termed "DEFFN Mode" and the second is "EDIT Mode". Each mode may support up to 32 unique edit functions. Generally, in DEFFN Mode the SF keys are user definable, and in EDIT Mode the SF keys perform system pre-defined editing functions.

At any time during entry of text with the line editor, the user may switch SF key modes by depressing the EDIT virtual key. The EDIT virtual key acts like a toggle switch. If the SF keys are in DEFFN Mode, depressing EDIT places them in EDIT Mode. If the SF keys are in EDIT Mode, depressing EDIT again places them in DEFFN Mode.

The current mode of the SF keys is identified by the cursor. A steady cursor means the SF keys are in DEFFN Mode. A blinking cursor means they are in EDIT Mode.

NOTE: This is true on most terminals. Refer to Appendix D for details.

The SF key mode is always returned to DEFFN mode after a text line is entered by keying RETURN, or removed by keying LINE ERASE.

5.4.5 Special Function Keys - DEFFN Mode

The special function keys may be "programmed" to enter user defined strings of text on a text line, with a single keystroke as if entered directly from the keyboard. This programming is done using the DEFFN' Statement (for further details refer to the Statements Guide, DEFFN'). These text definitions of the special function keys are accessible to aid text entry only when the SF keys are in DEFFN mode (steady cursor).

When in DEFFN mode, depression of any numbered SF key causes the line editor to search for the corresponding DEFFN' routine in the current program text area, with an integer number equal to that of the SF key depressed. If the DEFFN' statement contains literal text information immediately after the function number, that text is then passed to the line editor as if entered from the keyboard. Once completed, the line editor awaits entry of further standard keys or special function keys. If a DEFFN' of that number is not found, a beep is sounded to indicate the condition and further entry is awaited.

WARNING--If the function does not contain literal text information or parameters, the current line entry is canceled and the DEFFN' subroutine is executed.

DEFFN Mode is useful programming tool for assigning frequently used verbs or entire command lines to SF keys, thus allowing them to be entered as a single keystroke.

For example, the following would program SF Key 10 to become the PRINT verb:

```
100 DEFFN'10 "PRINT "
```

Once entered, touching SF key 10 followed by "A" and "$" would produce the statement:

```
:PRINT A$
```

in only 3 keystrokes.

The following DEFFN' is useful for saving intermittent copies of a program (for backup) as it is being developed.

For example:

```
DEFFN'31"RESAVE T";HEX(22);"PROGNAME";HEX(22);HEX(0D)
```

Depression of SF key 31 (when in DEFFN mode) causes the currently resident program named "PROGNAME" (or whatever other name is desired) to be saved on disk in place of its previous version. This presumes that the currently selected diskimage already contains the file "PROGNAME".

WARNING--When working with multiple modules in memory, use of DEFFN' literals to perform program SAVEs can give unexpected and unpleasant results if the program is currently resolved (HALTed while running), since DEFFN' functions are always referenced in the currently EXECUTING context. Refer to Section 5.3.5 for details.

Other frequent uses of this capability are:

- Printing the values of specific variables or lists of variables during debugging.
- Frequently LISTing a section of program code.
- Entering frequently used debugging commands in Immediate Mode (e.g., "CONTINUE RETURN", "CONTINUE NEXT", "STEP", "STEP OFF", etc.).
- Entering multiple spaces for quick "tabbing".

5.4.6 Special Function Keys - EDIT Mode

When in EDIT mode (blinking cursor), the special function keys perform system pre-defined editing functions.

The special function keys are labeled from 0 to 31. Of these, the keys numbered 0, 1, 16 and 17 are reserved and perform no function. The balance of the keys perform the functions as described in the following table when the special function key group is in EDIT mode:

SF Key	Label	Function
2	PREV	Displays the previous screen (23 physical lines) of the text line (assuming the text line occupies more than one physical screen).
3	NEXT	Displays the next screen (23 physical lines) of the text line (assuming the text line occupies more than one physical screen).
4	END	Moves the cursor to the end of the current text line.
5	SOUTH	Moves the cursor down to the next line of the text line (as summing the current text line occupies more than one physical line on the display).
6	NORTH	Moves the cursor up to the previous line of the text line (assuming the current text line occupies more than one physical line on the display).
7	BEGIN	Moves the cursor to the beginning of the text line.
8	ERASE	Erases all text from the current cursor position to the end of the text line.
9	DELETE	Deletes the single character at the current cursor position.
10	INSERT	Inserts a blank at the current cursor position.
11	EAST 5	Moves the cursor five positions to the east.
12	EAST	Moves the cursor one position to the east. Also,, EAST duplicates the function of the RECALL key when depressed at the end of a text line.
13	WEST	Moves the cursor one position to the west.
14	WEST 5	Moves the cursor five positions to the west.
15	RECALL	Recalls a program text line from memory,, or the last Immediate Mode line entered, or a data value from memory to the current text line for editing. More on this below.
18	SHIFT-PREV	Scroll to previous entry in the edit buffer.
19	SHIFT-NEXT	Scroll to next entry in the edit buffer.
25	LINE DELETE	Deletes all text from the current cursor position up to the end-of-text line or up to and including the next return graphic, whichever comes first.

SF Key	Label	Function
26	LINE INSERT	Performs one of two basic functions, depending on where in the text line the cursor resides when it is depressed.
	1	When depressed at the beginning or end of a physical line, a RETURN graphic is inserted followed by a statement separator (":") and one space at the current cursor position. The cursor is then advanced to the position following the space. This is useful for inserting a new program statement between two other statements on a multi-statement program line formatted with return-graphics.
	2	When depressed anywhere else on a text line, LINE INSERT inserts a return graphic only at the current cursor position and the cursor is not advanced. This is useful for multi-character insertion (discussed below).

NOTE: SF keys 11 an 14 are "sticky" keys. They must be pressed 5 times to skip past a return graphic. This facilitates getting to the start or end of a program statement when return-graphics are used.

NPL Release IV has implemented several additional enhancements to the Line Editor with regard to statement separators and return graphics when working with multi-statement program lines. Refer to Section 5.4.9 below for details.

5.4.7 Alternative to EDIT Mode SF Keys

Some physical keyboards have keys for which the label matches or nearly matches the functionality of the EDIT mode SF keys (e.g., arrow keys, INSERT key, DELETE key, etc.). These keys are often mapped in a manner which allows them to be used as an alternative to, or in addition to, the special function key group.

Keys of this type have the advantage that:

- EDIT mode functions may be invoked without the SF key group and without concern for the current mode of the SF key group.
- They are often conveniently located on the keyboard.
- They have appropriate labeling.

The following is a list of these virtual keys and a description of their functions in Immediate Mode when not in EDIT mode.

INSERT	Insert one character to right of cursor (default), or switch between insert and overstrike modes. The exact behavior is controlled by $OPTIONS byte 44.
DELETE	Delete one character at cursor location
NORTH	Up one line
SOUTH	Down one line
WEST	Left one character
EAST	Right one character
SHIFT-WEST	Same as WEST 5. Refer to Section 5.4.6
SHIFT-EAST	Same as EAST 5. Refer to Section 5.4.6
SHIFT-INSERT	Same as LINE-INSERT. Refer to Section 5.4.6
SHIFT-DELETE	Same as LINE-DELETE. Refer to Section 5.4.6
SHIFT-PREV	Scrolls backwards through previously entered commands stored in the edit buffer. Refer to Section 5.4.1 for details.
SHIFT-NEXT	Scrolls forward through previously entered commands stored in the edit buffer. Refer to Section 5.4.1 for details.
TAB	Cursor is positioned right to the next defined tab stop. Refer to Section 5.4.12 for details.
SHIFT-TAB	Cursor is positioned left to the next defined tab stop. Refer to Section 5.4.12 for details.

NOTE: The EAST arrow key, when pressed at the colon prompt, functions as a recall key to retrieve the last program line or Immediate Mode command entered. Alternatively, this key allows the user to retrieve any line of code, provided the line has a line number, by entering a line number and depressing the EAST key.

5.4.8 Multi-Character Insertion

The default mode of immediate mode, INPUT and LINPUT routines is "Overstrike" mode. Entering a key overstrikes the character at the current cursor position (except soft-carriage return), and the backspace key blanks the previous character.

Optionally, developers may choose to customize the NPL environment by invoking an "INSERT" mode of operation.

The following section discuss the editing process of character insertion using the default "Overstrike" mode. This is followed up with a discussion of "Insert" mode and it's functionality.

Overstrike Mode

Insertion of a few characters in between other characters of a text line is easily performed with the Insert function. Simply position the cursor to the point of insertion and depress the INSERT key once for each character to be inserted. This inserts enough blank space for the new characters to be entered.

However, this is inconvenient when more than just a few characters are to be inserted. Here, a better method is to use LINE INSERT. LINE INSERT allows the insert operation to be performed in fewer keystrokes (and with less character counting) as illustrated with the following example:

Assume the following text is on the current text line, the input cursor is at the end of the line, and the SF keys are in EDIT Mode.

```
175 = 1000 PRINT A,B,SGN(A+B)_
```

To change this statement to a PRINTUSING statement, with the print image on line 5000, we must insert the characters "USING 5000" after the verb "PRINT".

To do this, move the cursor to the blank following "PRINT" using one of the cursor movement keys (e.g., repeatedly depress WEST). Now touching LINE INSERT at this point results in the following change to the text line:

```
1000 PRINT≤
      A,B,SGN(A+B)
```

Notice that the RETURN graphic has been inserted at the cursor position and all subsequent text has been moved to the following line. This now allows new characters to be entered without overwriting subsequent text since the second line of text is now independent of the first line of text. Now enter the characters "USING 5000".

NOTE: The return graphic always follows the cursor automatically. This results in the following:

```
1000 PRINTUSING 5000,≤
      A,B,SGN(A+B)
```

To concatenate the two physical lines of text back together, simply delete the return graphic (using DELETE), resulting in:

```
1000 PRINTUSING 5000,A,B,SGN(A+B)
```

Insert Mode

Immediate mode, LINPUT and INPUT routines, support an "Insert" mode of operation in addition to, or in place of, "Overstrike" mode.

Byte 44 of $OPTIONS has been implemented to give developers selective control over this new feature. Byte 44 is used to control the start-up and editing characteristics of LINPUT/INPUT statements, immediate mode commands and program lines, and the characteristics of the "INSERT" key. Refer to the Statements Guide, $OPTIONS for a complete discussion on setting byte 44.

While insert mode is active, pressing keys automatically inserts a space at the current position, and the backspace key deletes the previous character (except soft-carriage return's).

By comparison, when overstrike mode is active typing a key overstrikes the character at the current position (except soft-carriage return's), and the backspace key blanks the previous character.

The insert key may be configured to either perform as before (inserting a single character) or to switch between insert and overstrike modes.

The following table defines all valid values for Byte 44 of $OPTIONS, and each bit positions start-up mode of operation.

Commands	Bit Position	Bit= 0	Bit= 1
Immediate Mode	HEX(01)	OVERSTRIKE MODE	INSERT MODE
Commands and			
Program Lines	HEX(02)	Ins key = Ins char	Ins key toggles mode
LINPUT	HEX(04)	OVERSTRIKE MODE	INSERT MODE
Statements	HEX(08)	Ins key = Ins char	Ins key toggles mode
INPUT	HEX(10)	OVERSTRIKE MODE	INSERT MODE
Statements	HEX(20)	Ins key = Ins char	Ins key toggles mode
RESERVED bits	HEX(40)	Should always be 0	
	HEX(80)	Should always be 0	

NOTE: If the Insert key is not configured to switch between modes, the selected starting mode (either OVERSTRIKE or INSERT) is locked in.

If the Insert key is not configured to switch between modes, there is no visible indicator to show whether the editor is in insert or overstrike mode.

Duplication of the options allows programmers to customize the editor to their personal preferences without affecting the way LINPUT/INPUT statements operate with existing software.

To avoid operator confusion, it is recommended that these values only be changed at startup time, in either the BOOT or PREBOOT programs.

Application software that does the equivalent of an extended LINPUT command should follow the behavior specified by bits 04 and 08 when processing entered keys, the back-space key and the insert key.

For example:

```
0010 DIM O$64<
     : O$=$OPTIONS<
     : STR(O$,44,1)=OR HEX(01)    :;Start Commands in INSERT Mode<
     : STR(O$,44,1)=BOOL4 HEX(02) :;Insert Key is INSERT Character<
     : $OPTIONS=O$                :;(Overstrike Mode not Available)
```

5.4.9 Multi-Statement Program Lines

As has been discussed previously, the readability and editing of multi-statement program lines can be significantly enhanced by the use of return graphics, which are generated by the LINE INSERT edit mode function. In addition to these, several other options are available to enhance and simplify the editing of multi-statement program lines. This section discusses these options.

Suppression of Statement Separators

The display of multi-statement program lines within the RunTime may be made more legible by suppressing the statement separators (":") at the start of multi-statement lines. When suppressed, the spacing of statements does not change, with a blank appearing in place of the statement separators. This is controlled by $OPTIONS byte 45, which contains the following bit values:

HEX(01) Suppress when recalling lines for EDIT

HEX(02) Suppress when displaying text in LIST (except LIST D).

HEX(04) Suppress when displaying text in LIST D.

HEX(08) Suppress when generating source via $SOURCE function.

HEX(10) Display space instead of return graphic when editing lines.

Regardless of the value of this option, a colon statement separator is now always optional after a return graphic.

NOTE: Text may be entered with some statement separators, if desired.

```
:0010 X=1<
    : IF T=Tax_code     :;Special record type?<
         PRINT "Taxes"; : PRINTUSING "##",X;<
      ELSE<
    :     PRINT "Income"<
      END IF
```

When LISTed or recalled for editing, the separators are all displayed or all suppressed, depending on the $OPTIONS setting:

```
:LIST 10

:0010 X=1<
    : IF T=Tax_code     :;Special record type?<
    :    PRINT "Taxes"; : PRINTUSING "##",X;<
    : ELSE<
    :     PRINT "Income"<
    : END IF

:DIM O$64:O$=$OPTIONS:STR(Q$,45,1)= OR HEX(02):$OPTIONS=O$
:LIST 10

:0010 X=1<
      IF T=Tax_code     :;Special record type?<
         PRINT "Taxes"; : PRINTUSING "##",X;<
      ELSE<
          PRINT "Income"<
      END IF
```

NOTE: If the return graphic at the end of a statement is deleted, the separator must be entered to maintain valid syntax.

When editing, statements are always indented at least 2 spaces from the first "editable" character on a line.

When return graphics are suppressed (HEX(10) bit= 1), they still occupy an editable location and can be deleted or inserted, but display as a space.

NOTE: The newest version of the NPL compiler (B2C) permits .SRC files to follow this convention as well, so ASCII versions saved using LIST may use the new convention. Older versions of the compiler require the statement separator at the start of each statement.

Files produced in 2200 atomized format by the compiler (using LSTFORMAT -2200[S]) always have the statement separator inserted where it is implied.

5.4.10 Statement Insertion and Deletion

As discussed in Section 5.4.9, multi-statement program lines can be made more readable and editable with the use of return graphics and the suppression of statement separators.

The LINE INSERT and LINE DELETE Edit Mode special function keys work together with the return graphic to simplify the process of adding, inserting and deleting program statements within a multi-statement program line (not to be confused with inserting and deleting program lines, which is discussed in the next section).

The utility of this is illustrated by the following simple edit session.

Presume we wish to enter a multi-statement program line which contains three statements. The first statement is entered in the usual manner in Immediate Mode as follows:

For example:

```
:1000 FOR I=1 TO 100
```

We have entered the first statement and the cursor resides at the end of the line (RETURN has not been depressed).

Adding Statements

We now wish to add a second program statement to this. Depressing LINE INSERT at this point inserts a return graphic followed by a statement separator (":") and a single space to yield the following:

```
:1000 FOR I=1 TO 100<
    :
```

Automatically, our program line is now syntactically prepared to receive the next program statement. Using the same procedure we can enter the remaining two statements to yield:

```
:1000 FOR I=1 TO 100<
    :     PRINT SGN(I)<
    : NEXT I_
```

Inserting a Statement

Now presume we wish to insert a new statement after the first statement but before the second. To do this, position the cursor to the statement separator between these two statements. Now depressing LINE INSERT yields the following display:

```
:1000 FOR I=1 TO 100<
    : ≤
    :     PRINT SGN(I)<
    : NEXT I
```

The first and second statements are separated by a return graphic, a statement separator is displayed, and the text of the new statement can be entered to yield:

```
:1000 FOR I=1 TO 100<
    :     PRINT I<
    :     PRINT SGN(I)<
    : NEXT I
```

Deleting a Statement

Now presume we wish to delete the 3rd statement in the program line. To do this, position the cursor to the statement separator preceding the third line. Now depressing LINE DELETE deletes all text from the cursor position up to and including the statement separator to yield:

```
:1000 FOR I=1 TO 100<
    :     PRINT I<
    : NEXT I
```

The editing session may be terminated at any point by depressing the RETURN key, at which point the program line is submitted for processing.

NOTE: The WEST 5 and EAST 5 keys operate slightly differently when used on a text line with return-graphics. Specifically, the beginning and ending of physical lines form "sticky spots" for these keys (and these keys only). That is, it takes several depressions of the key to move the cursor past these points. This is intentional as it allows the WEST 5 and EAST 5 keys to be used to quickly move the cursor to the beginning or end of a line.

Forcing Return-Graphics

Programs which make extensive use of multi-statement program lines but do not make use of return-graphics are somewhat more cumbersome to edit. It is still possible to edit a program authored in this fashion with the aid of return graphics.

Return-graphics may be automatically inserted in program lines by use of the $KEEPREMS system variable and with EDIT/RECALL as follows:

In Immediate Mode, assign $KEEPREMS the value of HEX(02) as follows:

```
:$KEEPREMS=HEX(02)
```

Now return-graphics may be automatically inserted into multi-statement program lines by recalling the statement and pressing RETURN without making changes. This causes the line to be compiled with return-graphics.

Now recalling the line displays it in the one statement per line fashion. Be sure to change the value of $KEEPREMS back to the normal value (HEX(01)) when all lines that need return graphics inserted have been modified.

5.4.11 Multi-Screen Text Lines

Due to the multi-statement program line capability of NPL, it is possible that a single text line may exceed the size of the physical screen. Here, the text is still considered to be a single text line, however, special multi-screen handling logic is used by the line editor to allow for this case.

When editing multi-screen program lines, the editor scrolls the screen a single line up or down when the cursor is moved off the bottom or top of the screen. This scrolling only applies to small cursor movements and to typing which extends the line past the end of the screen. Large cursor movements (prev-screen, next-screen, goto-beginning, goto-end) will jump the display in full screen units, 23 lines at a time.

NOTE: Single line scrolling requires the use of a control code (insert line) for remote terminals to scroll the screen down one line. Not all terminals support this capability (in particular, Wang 2236 and IBMPC /R option). These terminals will do a full screen redisplay when scrolling up off the top or down off the bottom of the screen.

When editing multi-screen text lines, the presence of "---PREV---" and/or "---NEXT---" indicators are periodically noticed at the bottom of the screen. These indicators mean that the text line currently displayed has more text on the next page when "---NEXT---" is displayed, and has more text on the previous page when "---PREV---" is displayed. The PREV and NEXT Special Function Keys may be used to "flip" to these pages for viewing or editing.

While entering or editing a multi-screen text line, screen scrolling is largely automatic. That is, any cursor movement which causes the cursor to encounter a page boundary causes the screen to scroll up or down one line at a time.

For example, touching the WEST SF key when the cursor is at the top left hand corner of a page causes an automatic scroll of the screen down one line. After the scroll, the cursor would be located on the top line of the screen at the end of the line.

All other SF keys are fully functional with multi-screen text lines. With the view that text lines of this type are still considered to be a single text line, the actions of the remaining SF keys are easily predicted.

HINT: Widespread use of multi-screen text lines is not recommended as they tend to be cumbersome to edit and maintain.

5.4.12 Use of TAB/SHIFT-TAB

TAB and SHIFT-TAB functions are provided to allow the programmer to add visible structure to the program code.

In addition to the Special Function keys described in previous sections, the TAB and SHIFT-TAB keys may be used to move the cursor horizontally when editing text. Unlike other cursor movement keys, TAB and SHIFT-TAB do not move a fixed number of positions. Rather, movement is based on values contained in the $TAB system variable. $TAB is a 132-byte variable that defines tab stops. Default tab stops are defined every fourth position but these may be changed by the programmer--refer to $TAB in the Statements Guide for further details on defining TAB positions.

Whenever TAB is pressed, the cursor is located at the next higher defined tab stop. If TAB causes the cursor to be located beyond a return graphic character (refer to Section 5.4.9 above for details on return graphic characters), the cursor is moved to the return graphic character and sufficient spaces are automatically inserted to move the cursor and return graphic character to the next TAB position. If TAB would cause the cursor to be moved beyond the end of the program line, the cursor is located at the end of the program line and sufficient spaces are inserted to move the cursor and the end of the program line to the next TAB position. Once the return graphic or end of line has been moved to the desired TAB position, new statements can be entered normally.

Whenever SHIFT-TAB is pressed, the cursor is moved to the left to the next lowest TAB position. SHIFT-TAB never moves the cursor beyond the start of a physical line which follows a return graphic character.

NOTE: TAB never causes spaces to be inserted in the middle of program text. Spaces are inserted only at return graphics or end of program lines. However TAB can be used to insert spaces when used in conjunction with SHIFT-INSERT.

For example, assume that for the statement:

```
0010 A=B: REM Set A equal to B
```

the programmer wishes to move the REM statement to column 40. Further assume that $TAB defines tab positions at columns 4, 8, 12, 16, 20, 24, 28, 32, 36, and 40 (the default values). The REM statement can be moved to column 40 as follows:

1. Recall line 0010.

2. Move the cursor to the colon following "A=B".

3. Press SHIFT/INSERT. This produces a line that appears as:

```
0010 A=B<
   : REM Set A equal to B
```

   with the cursor located under the return graphic.

4. Press TAB 8 times. This moves the cursor and return graphic to column 40.

5. Press DELETE. This deletes the return graphic producing:

```
0010 A=B                          : REM Set A equal to B
```

This technique is useful when it is desirable to establish visible structure in programs that were created before the TAB function was available.

NOTE: TAB does not generate any special characters in the p-code and no adjustment is made to existing program text if different $TAB values are used. Therefore, consistent $TAB values should be used from one editing session to the next to insure that a consistent visible structure is maintained.

The actual TAB and SHIFT-TAB virtual key on a specific keyboard may not be located on the same physical key. Refer to Appendix D for information about the availability of the TAB and SHIFT-TAB keys on the terminal being used.

5.5 Editing Programs

The line editor can be used to edit program text in memory on a line by line basis. Each program line in an NPL program is uniquely identified by its line number. All editing operations use the line numbers to identify the particular line or group of lines that are the object of the edit operation.

5.5.1 Selecting a Module to Edit

This section describes the process of selecting a module for editing as a stand-alone program and while under program control.

Selecting a module for editing as a stand-alone program is easily accomplished by LOADing the program from the appropriate diskimage. Refer to Section 5.3 for details.

Editing a module under program control is a little different since multiple modules may be residing in memory at the same time. To edit a module in memory when other modules are present, the module must first be selected as the current LIST module. This is easily accomplished using the "MODULE" command.

For example:

```
:MODULE "SETCOLOR"
```

The above immediate mode command would set the current LIST module to the program SETCOLOR. Existing modules in memory can be identified with the LISTDT command. Refer to Section 5.3.5 for a discussion on the effects of modules on the current workspace.

5.5.2 Entering a New Program Line

To enter a new program line, simply enter a text line in Immediate Mode which begins with a unique line number followed by one or more program statements. On pressing RETURN, the program line is inserted into program text memory according to its line number position within the current LIST module. Care must be taken that a program line of the same number does not already exist in memory for that module.

5.5.3 Replacing a Program Line

To replace an existing program line in memory with a new line, enter the new program line in Immediate Mode which begins with the line number to be replaced, followed by one or more statements which are to replace the existing program line. Upon depression of the RETURN key, the existing program line is replaced with the new program line.

5.5.4 Changing a Program Line with RECALL

To change an existing program line in memory, enter the line number in Immediate Mode of the program line to be changed without depressing the RETURN key. To recall the program line, either:

- Press the EAST arrow key, or

- Place the SF key pad in EDIT mode (if not already done) by depressing the EDIT key and press the RECALL key.

This causes the line editor to copy the specified program line to the edit buffer, display the program line on the screen. With the second option, the cursor is moved to the end of the text line.

At this point all line editor functions are available to assist in changing the program line. Once all changes have been made, depressing the RETURN key replaces the line in memory with the modified program line as shown on the text line.

NOTE: **During this process, all changes are being made to a copy of the program line in the edit buffer; the original line remains unchanged in memory until the RETURN key is depressed. Therefore, to cancel a change operation, erase the line from the edit buffer with LINE ERASE before touching RETURN. The original line remains in program text memory unchanged.**

5.5.5 Deleting Program Lines

A program line may be deleted from program text memory by entering the program line number in Immediate Mode and depressing the RETURN key without entering any program text. The absence of program text after the line number causes the line editor to interpret the entry as a deletion.

After depressing RETURN, the specified program is deleted from program text memory and the message "LINE DELETED" is displayed.

NOTE: **If the line was deleted in error, immediately depressing the RECALL key without any line number recalls the line to the edit buffer, where it may then be reinserted by depressing RETURN.**

If the specified line number does not exist in the current program text area, the message "NO SUCH LINE" is displayed and no deletion is performed.

A range of program lines may be deleted from the program text area with just one command, the CLEAR P command.

For example:

```
:CLEAR P 1000,1999
```

deletes program lines 1000 through 1999, inclusive, from the program in memory. For further details refer to the Statements Guide, CLEAR.

5.5.6 Concatenating Program Lines

Any program line in memory may be inserted into the current text line. This is done by entering a statement separator (":") at any point in the text line, immediately followed by the program line number to be inserted.. Then, depressing the RECALL SF key (when in EDIT Mode) causes the line editor to replace the specified line number (on the text line) with a copy of the program line text of that line number from program text memory.

This process may be repeated as often as desired allowing for many program lines to be inserted to the current text line.

If the original text on the line was a program line (i.e., begins with a number), then depressing RETURN causes that program line together with all newly concatenated lines to be inserted into program text memory as a single multi-statement program line.

NOTE: Inserting a program line into a text line does not delete the program line from the program text memory. It is merely copied from program text memory. Therefore, if required, the inserted program lines must be deleted with the delete or CLEAR P command.

5.5.7 Moving Program Lines

Program lines in memory may be moved one of two ways:

1. EDIT/RECALL--Once a program line has been RECALLed to the edit buffer (and displayed) using EDIT/RECALL, the line number is editable, as is any other text on the line. If the line number is changed, the program text is saved under the new line number in memory once the RETURN key is depressed.

NOTE: Under this method, the original line is unaffected and must be deleted as a second operation, if required. Further, all references, if any, to that line number by other program lines (e.g., GOTOs) must also be edited to reflect the new line number.

2. RENUMBER--The RENUMBER command can be used to renumber a program line or range of program lines in memory. In addition, a line number increment may be specified which allows for spacing between program lines.

   This method of renumbering program lines has the advantage that the actual program line numbers in memory are changed, and consequently deletion of old line numbers is not required. Further, once the renumbering operation is complete, the command performs an internal line number cross-reference on the entire program and automatically adjusts all references to old line numbers to the new line numbers.

   For example:

   ```
   :RENUMBER 1000-1999 TO 4500 STEP 10
   ```

   Renumbers program lines 1000 to 1999 inclusive starting with new line number 4500 and incrementing line numbers by steps of 10 from there.

For further details refer to the Statements Guide, RENUMBER.

5.5.8 Recalling Long Identifier Names

To help avoid typographical errors when typing in Long Identifier Names (LINs), the "RECALL" key may be used at the end of a line after only partially entering an identifier, to complete the name of the identifier.

For example:

```
:0010 DIM SausageName$27
:0020 PRINT Sau_
```

If RECALL is depressed (or right-arrow) at this point, NPL looks for a unique identifier starting with Sau... If only one identifier exists, NPL completes the name, with the standard case. If more than one identifier starts with the partial name, the system beeps and no change is made to the line.

Variable identifier look-ups may also look for a partial variable name as "ident[?ifier][$][(]" before the cursor, where "ident" must match the start of some previously entered identifier, and if the "?" appears, "ifier" must also match the end of the identifier. If "$" or "(" is specified, the variable name must be the name of a string or array, respectively.

For example, if the following program is entered:

```
0010 DIM ThisLittlePiggyWentToMarket$10
  :  DIM ThisLittlePiggyStayedHome$10
  :  DIM ThisLittlePiggyHadRoastBeef$(2)10
  :  DIM ThisLittlePiggyHadNone
```

Then:

"This?Market"	(-RECALL-) would find ThisLittlePiggyWentToMarket$
"This$"	(-RECALL-) is ambiguous (could be .. Market$ or ..Home$)
"This?None"	(-RECALL-) would find ThisLittlePiggyHadNone
"This$("	(-RECALL-) would find ThisLittlePiggyHadRoastBeef$(
"?Home"	(-RECALL-) would find ThisLittlePiggyStayedHome$

When recalled, String /Array identifiers have a "$" or "(" attached to them, even if not specified.

In addition to the above, programmers using long identifiers may find it useful to set byte 38 of the $OPTIONS to HEX(01), which requires all identifiers to either be common (COM) or to appear in a DIM statement before any other reference in a module. An error occurs at resolve time for the first reference not preceded by a declaration. This can help to avoid programming errors of the form:

```
10 DIM SausageCount
20 SausageCoutn=12 : REM Oops, meant SausageCount
```

In the above instance, an error P55--Undefined variable occurs on line 20 at resolve time, indicating that a variable appeared before its appearance in a DIM/COM statement.

5.5.9 Syntax Errors

After a program line is entered using the line editor, it is immediately checked for correct syntax before it is submitted for storage in program text memory. If a syntax error is detected, it is immediately reported. Specifically, the program line is displayed and a character of the program line is highlighted in the approximate position of the syntax error. Below the line an error indicator is displayed with an error number and descriptive message.

If the program line is a multi-statement program line, then only the first syntax on the error is reported.

For example:

```
:1000 FOR I=1 Tu 1000<
    :     J=A$+SGN(I)<
    :     PRINT J<
    : NEXT IX
             ^
ERR S19: Missing Required Word
```

In this example, program line 1000 was entered but was found to contain 3 syntax errors on the first, third and fourth statements of the multi-statement line. Only the first error is diagnosed. Notice the highlighted character in the statement with the error ("T" in the example) which indicates the line containing the error. The arrow also points to the approximate position of each error.

Once a program line is diagnosed as containing a syntax error, it is stored in program text memory, but is flagged as being inoperable. To correct the program line, it must be recalled in the usual fashion with the line editor and corrected.

NOTE: Both the Interpreter and RunTime Package do not allow execution or even partial execution of a program which contains any program lines with syntax errors. A program must be 100% free of syntax errors before execution proceeds.

Descriptive error messages are contained in files ERRORMSG.HLP and ERRORMSG.IDX. These files may be modified. Please refer to the appropriate NPL Supplements and Section 2.7 for further information on these files.

5.5.10 Other Useful Editing Tools

There are other useful editing tools available using Immediate Mode commands, which, although they do not actually change text, do aid in the process of program text development.

Each of these commands is explained in detail in the Statements Guide, but they are summarized here for reference.

LIST	Lists a program line or range of lines. It is often useful to include LIST as a DEFFN' keyboard text definition. For further details refer to the Statements Guide, DEFFN'.
LIST FIELD	Lists all program lines which contain a specified FIELD or range of FIELDs.

LIST FUNCTION	Produces a FUNCTION cross reference listing.
LIST PROCEDURE	Produces a PROCEDURE cross reference listing.
LIST RECORD	Lists all program lines which contain a specified RECORD or range of RECORDs.
LIST T	Lists all program lines which contain a specified text string.
LIST V	Lists all program lines which contain a specified variable or range of variables.
LIST = name	Lists all program lines which contain a specified Statement Label or rang of Statement Labels.
LIST ’	Produces a DEFFN’ cross-reference listing
LIST #	Produces a line number cross-reference listing.
SPACE	Determines the current maximum size of the edit buffer.
SPACE F	Determines the remaining program text memory available.
SELECT LIST	Temporarily directs output from any LIST command to the printer for hardcopy listings.

5.6 Saving Programs

Once NPL programs have been created or modified using the editor, it is necessary to explicitly SAVE them on disk so that programs or program changes are not lost. This is accomplished through the use of the SAVE command. Programs may be saved to both NPL diskimages and with limitations, to native operating system file systems.

This section discusses saving both newly created programs and already existing programs.

NOTE: This section does not discuss all syntactical variations of the SAVE command. For further details refer to the Statements Guide, SAVE.

5.6.1 Saving New Programs to Diskimages

A new program file is saved to a diskimage with the SAVE command.

For example:

```
:SAVE T/D10,(5) "PROGRAM"          -Saves a program to diskimage
                                    D10 under the program name
                                    "PROGRAM" with 5 extra sectors.
```

SAVE Command--With the SAVE command the diskimage's catalog area is first checked to determine if sufficient space is available to save the program. If so, the specified program name is entered into the catalog index and the program is saved in the catalog area.

NOTE: The program name must not already exist in the catalog index (refer to next section, Saving Existing Programs to Diskimages).

If insufficient space is available in the catalog area for the program an error is issued and the program is not saved. Here, the catalog area must be expanded by the MOVE END statement, or MOVE DISK can be used to reorganize the diskimage to recapture unused space.

If insufficient space is available in the catalog index area for the program name, an error is issued and the program is not saved. In this case a new diskimage with a larger catalog index area must be created. Programs and data from the old diskimage may be copied to the new diskimage using the 2CCOPY utility (refer to Chapter 13).

When saving new programs, it is good practice to reserve extra catalog sectors over and above what is required for the actual program code, to allow for future growth. This is a parameter to the SAVE statement which when specified is added to the actual size of the program when saving. In the SAVE example above, 5 extra sectors were requested.

The LIST DC command can be used to determine the actual amount of free space available for a particular program and also in the entire diskimage catalog area.

For further details refer to the Statements Guide: SAVE, MOVE END, and MOVE DISK.

5.6.2 Saving Existing Programs to Diskimages

Catalog space reserved for existing program files in a diskimage may be reused for newly edited versions of the program (or replaced with entirely new program code and program name).

This is done by using the RESAVE command. The RESAVE command allows the user to save an existing program to disk without having to issue a SCRATCH command on the file first.

For example:

```
:RESAVE T "EXAMPLE"
```

The RESAVE command in this example locates the program called "EXAMPLE" in the catalog index and saves the current contents of memory to it.

For details refer to the Statements Guide, RESAVE.

Insufficient Space in Old Program

When reusing catalog space with the RESAVE command, the condition may arise where the new version of the program may be larger than the available space of the old program to be overwritten in the diskimage.

Here, the RESAVE command automatically attempts to allocate new space at the end of the free catalog area. If space is available, the program is saved at the new location (but with 0 extra sectors). The original space occupied by the old program becomes "dead space" and cannot be used until the diskimage is reorganized using the MOVE DISK command.

If space is not available, a "Catalog Full" error results. In this case, it is necessary to expand the diskimage (using MOVE END) before the program can be saved.

Use of DEFFN’

Scratching and saving programs is a very common practice. As a convenience to the programmer, the DEFFN’ statement can be used to automatically scratch and save program text, using the correct program name with just one or two keystrokes.

For example:

```
0000 DEFFN'30 "X$=";HEX(22);"PROGRAMX";HEX(22);":SCRATCH T X$";HEX(0D)
0001 DEFFN'31 "SAVE T () X$";HEX(0D)
```

Here, program lines 0 and 1 are reserved for the DEFFN' statements. The actual program code would follow. While in Immediate Mode, pressing special function key 30 (SF keys must be in DEFFN Mode--refer to Section 5.4.3 for details) issues a SCRATCH command for "PROGRAMX". Then pressing special function key 31 issues the SAVE command for the same program file.

Alternatively, a single DEFFN' may be used:

```
0000 DEFFN'31 "X$=";HEX(22);"PROGRAMX";HEX(22);":SCRATCH T X$ :SAVE T
() X$";HEX(0D)
```

One other alternative to the above SCRATCH and SAVE operations within the DEFFN' statement would be to use the RESAVE command.

For example:

```
0000 DEFFN'31 "RESAVE T";HEX(22);"PROGRAMX";HEX(220D)
```

The appropriate DEFFN' statements are often included as permanent lines in a program so that this convenience is always available when editing the program.

HINT: For consistency, it is recommended that all programs use the same program line numbers to store the DEFFN' statements and that the same SF Keys be chosen to invoke the DEFFN' statements. The examples above are indicative of the typical choice for these.

For further details on the use of SF keys in DEFFN Mode, refer to Section 5.4.3. For further details refer to the Statements Guide: SAVE, SCRATCH, RESAVE and DEFFN'.

5.6.3 Saving Portions of Programs to Diskimages

In some cases, it is desirable to save a portion of a program in memory to disk. This can be accomplished by use of the line number range parameter in the SAVE command. The line number range parameter is the last parameter on the command line, when used.

The following are examples which save a portion of the program in memory to a new program file in the diskimage:

For examples:

```
SAVE T "PROGRAM1" 400,800
```

stores program lines from 400 through 800 in a new program file named "PROGRAM1".

```
SAVE T "PROGRAM1" 500,
```

stores program lines from 500 through the highest line number in memory in a new program file named "PROGRAM1".

The following example saves a portion of the program in memory to an existing program file in the diskimage:

```
SCRATCHT T "PROGRAM1"
SAVE T () "PROGRAM1" 400,800
```

SCRATCHes program file "PROGRAM1" and overwrites it with lines 400 through 800 of the program in memory.

As an alternative to the above, the RESAVE command may be used to perform the equivalent of the SCRATCH and SAVE operation.

For example:

```
RESAVE T "PROGRAM1" 400,800
```

has the same effect as the above SCRATCH and SAVE statements.

5.6.4 Saving Programs to Native Operating System Files

NPL programs may be saved within the native file system for the limited purpose of the BOOT or PREBOOT program. A BOOT or PREBOOT program is an NPL program which is stored in a native operating system file system and is automatically loaded and executed by the RunTime at startup.

A program file is saved in a native operating system directory by use of the SAVE BOOT command. When executed the specified program is saved using a specific native operating system file specification. If the program already exists within the native file system, it is overwritten.

When saving programs in a native file system, the $SHELL command may be useful for locating and listing the names of BOOT programs contained in the native file system or any other preparation required in advance of executing a SAVE BOOT command.

For further details refer to the Statements Guide, SAVE BOOT and $SHELL. Refer to Section 2.4 of the appropriate NPL Supplement for details on the BOOT program and considerations specific to the operating system being used.

5.6.5 Scramble Protecting Programs

As a means toward copy protection of application systems written in NPL, programs may be SAVEd in a scramble protected format. Programs saved in this format are encrypted on disk, thereby discouraging direct examination of the program. Scramble protected programs may be LOADed and executed by either the interpretive or non-interpretive Run-Time programs, but may not be LISTed or examined using debugging commands.

Programs are scramble protected by use of the optional "!" parameter to the save command.

For example:

```
:SAVE T ! "PROGRAM"
```

SAVEs program "PROGRAM" in scramble protected format.

The compiler can also be used to generate programs in scramble protected format using an OBJFORMAT of "SCRAMBLED".

The programmer should, of course, maintain at least one copy of scramble protected programs in unscrambled format for purposes of program maintenance.

For further details of the scramble parameter refer to the NPL Statements Guide, SAVE.

CHAPTER 6

DEBUGGING CODE

6.1 Overview

The process of "debugging" a program generally requires that the program's operation be inspected to determine the problem and then once determined, corrected by changing the associated source code.

This chapter discusses the methods by which an NPL program may be inspected at execution time with Immediate Mode commands. This includes inspection of the program's logic flow, text, variables and environment. Methods of altering program logic flow (as opposed to program text) during execution are also discussed. This includes modification of program execution order, variable content, and the program's environment.

The NPL debugging facilities described in this chapter are available only in the interpretive RunTime (RTI), not in the non-interpretive RunTime (RTP).

Prerequisites

Program Editing--Once a program bug has been isolated, it is usually corrected by editing the program text. In NPL, program editing is highly integrated with program debugging to allow both to be performed concurrently. The subject of program editing is discussed in Chapter 5, Creating NPL Programs and the NPL Editor, and is a prerequisite to this chapter.

Immediate Mode--An essential part of program debugging is the ability to obtain control of an executing program. Once control is established, the program may then be inspected by the programmer in its various stages of execution. In NPL, control over an executing program is acquired by invoking Immediate Mode in NPL. There are many methods of invoking Immediate Mode. This subject is discussed in Chapter 2 and is essential to an understanding of program debugging in NPL.

Section 6.2 discusses the effect of modules when debugging.

Section 6.3 discusses the inspection of program logic and the Immediate Mode commands available to modify the logic flow of a program during its execution.

Section 6.4 discusses the inspection and modification of program variables.

Section 6.5 discusses the inspection of program text while in Immediate Mode.

Section 6.6 discusses the use of Immediate Mode commands and system variables for inspection of the environment in which the program is running.

6.2 Debugging Modules

Default operation of the RunTime assumes that debugging the Root Module is a more frequent event than debugging other modules at execution time. If another module must be selected, the following guidelines must be followed when debugging programs.

- Debugging with the STEP and TRACE commands is active for all modules.
- Program inspection with LIST commands and RENAME and RENUMBER commands refers to the current List Module only.

- Variable inspection and modification can be done within the scope of the current Executing Module. Refer to Section 4.10 for more detail on the use and types of modules.

If a program has been RUN and an error is detected, before making a change:

1. Ensure that the Run Module is selected as the current List Module by using the MODULE command (with a blank name):
```
:MODULE " "
```
2. Make the necessary program changes to the module and SAVE it.

If a program has been RUN that includes a module (i.e., a "LIBRARY"), and a change is required to the INCLUDEd module, the correct procedure is as follows:

1. Select the currently loaded library as the LIST module by using the MODULE command.

   For example:
```
:MODULE "LIBRARY"
```
2. Make the necessary program changes to the module and SAVE it. To test that the module resolves correctly, a RUN[,STOP] command should be used (assuming that the root module still INCLUDEs "LIBRARY").

NOTE: RUN also resets the current module to the Root (no name) Module, so if there are errors, the current List Module must be reset again.

For programmers familiar with NPL versions prior to Release IV, which do not support multiple modules, the usual inclination is to do a CLEAR P, LOAD the LIBRARY module (without using the MODULE command), fix it (SAVE it) and then try to RUN it to check for syntax errors. This doesn’t work properly because:

- The "LIBRARY" module is still loaded and resolved. All PUBLIC variables will be flagged as duplicates if an attempt to RUN a second copy in the root (no name) MODULE is performed.

- If, after making the change, do a CLEAR P, reload the normal root module and try to RUN it then the changes made to "LIBRARY" are not detected. The old module is still running!

To select the root module as the current List Module, use:

```
:MODULE " "
```

NOTE: We strongly recommend that FUNCTIONs and PROCEDUREs intended to be used in modules first be developed and debugged as local FUNCTIONs/PROCEDUREs. However, interactive debugging of code residing in modules is well supported.

6.3 Inspection and Modification of Program Logic

NPL provides a series of Immediate Mode commands that allow the programmer to inspect the logic flow of a program as it is executing. These commands represent the most powerful debugging capability of NPL since they allow the programmer to view the actual sequence of events that take place "live" during the execution of a program.

Besides inspection of logic flow, other Immediate Mode commands allow the programmer to modify the logic flow of a program during its execution, to test alternate paths of logic (without modifying actual program text).

The following section discusses both the inspection and modification of program logic for debugging purposes .

6.3.1 Logic Inspection Using Stepped Execution

The NPL Interpreter supports execution of programs specially intended for debugging called "Stepped Execution." With Stepped Execution of a program, the Interpreter executes only one statement of the program at a time, displaying each statement before it is executed with full Immediate Mode capabilities enabled between the execution of each statement. This allows for unlimited variable inspection, modification, program listing and other debugging activities to take place concurrent with Stepped Execution.

Stepped Execution is controlled by using two elements that are internal to the Interpreter, named "Step Mode" and "Step Range." Step mode is "global" meaning the status is consistent, despite module status or position. A different Step Range may be specified for each module. Both Step Mode and Step Range are modifiable by the programmer using Immediate Mode commands described later in this section.

Step Mode

Step Mode is an on/off switch. Stepped Execution is eligible to take place only when Step Mode is ON. Step Mode is on or off whatever module status or position. In other words, a module cannot have Step Mode on while another module has Step Mode off.

Step Range

Step Range consists of a starting line number and an ending line number. Stepped Execution is eligible to take place only when the currently executing program is executing a program line that is within the defined Step Range.

Each module is assigned its own STEP line number range. By default, the line number range for STEP in the Root Module encompasses all line numbers while the line number range for STEP in other modules encompasses no line numbers. The effect of this is that it is very easy to step through root code without stepping through code in other modules.

By manipulating the Step Mode, the programmer may begin and stop Stepped Execution of program lines within the Step Range.

By manipulating the Step Range, the programmer can concentrate Stepped Execution on a specific area of code within a program (such as a subroutine or group of subroutines) without stepping through other areas of the program that are already tested. That is, when Step Mode is on, Stepped Execution is enabled automatically whenever program control branches into the range of program line numbers specified by the Step Range for the executing module. When program control branches out of the Step Range, execution of the program automatically resumes (no stepping) until control again passes back into the Step Range.

Activating Stepped Execution

There are three methods by which Stepped Execution may be enabled. These methods may be used both before and during program execution. They are:

- **The STEP instruction**. The STEP instruction manipulates the Step Mode switch. When STEP is executed, Step Mode is turned ON, by that activating Stepped Execution when a line number in the current Step Range is executed.

  STEP may be entered as an Immediate Mode command either before or during program execution. STEP may also be included as a statement in a program.

- **The STEP # instruction**. The STEP # instruction manipulates both Step Mode and Step Range. When STEP # is executed, Step Mode is turned ON, and the Step Range for the current List Module is assigned the line numbers as specified in the instruction. Stepped Execution is started when a line number in the specified Step Range is executed.

  For example:

  ```
  : STEP # 1500,2800
  ```

  When executed, Step Mode is turned ON, and the Step Range for the current List Module is assigned to start with line number 1500 of a program and end with line number 2800 (inclusive).

  STEP # may be entered as an Immediate Mode command either before or during program execution. STEP # may also be included as a statement in a program.

- **From the HELP Display**--Stepped Execution may be started by selecting the STEP menu item on the HELP display. When selected, the HELP processor implicitly executes a STEP # command with a specified line number range of 0-32117(all lines) in the currently executing module.

  This, therefore, sets Step Mode ON and sets the Step Range to 0-32117 (the entire program) in the currently executing module. This immediately begins Stepped Execution.

NOTE: When STEP is selected from the HELP display during program execution, the program may be in the process of executing a keyboard input statement (INPUT, INPUT, or non-polling KEYIN). In these cases, selecting the STEP option in this manner does not invoke Immediate Mode right away. Execution of the keyboard input statement must first be completed by entering the data value in the usual manner. Once completed, Immediate Mode is then activated and Stepped Execution is started.

To start Stepped Execution by entering STEP or STEP # as an Immediate Mode command during program execution, it is necessary to first interrupt the program by invoking Immediate Mode. This is done by a variety of methods, all of which are documented in Chapter 2. Of these, the most common method used is the HALT key. When depressed, program execution is halted and Immediate Mode is invoked.

NOTE: HALT does not disturb the "execution status" of the program, in that stepped execution is still allowed to take place.

To start Stepped Execution in a program using modules, simply HALT the program at whatever point is most convenient, even if not in the Root Module. Then enter the STEP command and press the EXEC key (or enter CONTINUE). If the default Step Range values have not been altered, program execution continues without stepping until the next statement in the Root Module is encountered. If subsequent calls to other modules are encountered while in step mode, the subroutine will execute without stepping, but step mode will be resumed upon return to the Root Module.

To start Stepped Execution to debug a particular module other than the Root Module, follow these steps:

1. Issue the MODULE command to change the current List Module to the module needed to debugged.

2. Issue the STEP # command specifying the range in that module to be used by STEP.

3. Use the MODULE command to change the current List Module back to the Root Module.

4. While in the Root Module, issue the command STEP # 1,0 so that no lines in the Root Module will be STEPped through.

5. Press the EXEC key to begin program execution. This will continue until a statement within the specified range in the particular module is to be executed.

Refer to Section 6.2 for information on how to select a module to debug.

Operating Stepped Execution

Once the STEP processor is started, it displays the next program statement that is executed.

NOTE: The STEP processor always displays a statement before it is executed.

If the statement displayed is part of a multi-statement program line, then it is preceded by one statement separator (":") for each statement that preceded it on the program line (if any), and is followed by another statement separator and ellipsis ":..." if any other statements follow on the same program line. In any case, the program line number is always displayed at the beginning of the line.

For example:

```
1000 :::B$=STR(X$,2):...
   :_    (<--Immediate Mode is invoked)
```

This STEP processor display indicates that "B$=STR(X$,2)" is the next statement to be executed. It was preceded by three other statements on the program line (as indicated by the ":::") and is followed by other statements on the same program line as shown.

NOTE: Return graphics characters are ignored by the STEP processor to preserve screen space while stepping through programs.

The STEP processor always invokes Immediate Mode immediately after each statement is displayed. This allows for full Immediate Mode capability while stepping through program execution. When the Immediate Mode prompt (":") is displayed, the programmer has several options. The programmer may:

- Execute the displayed statement by depressing the EXECUTE key. At this point the statement is executed, the next statement to be executed is displayed and control is again returned to Immediate Mode.
- Avoid execution of the displayed statement by entering certain Immediate Mode commands (as discussed in "Logic Modification" below).
- Enter Immediate Mode commands for any other purpose including variable inspection, variable modification, program listing and other debugging activities.
- Resume normal execution of the program by deactivating Stepped Execution (as documented below).

Any number of Immediate Mode commands may be entered at this point without forfeiting the option to continue Stepped Execution. However, it is important to avoid commands that would disturb the "execution status" of the halted program if it is intended that further Stepped Execution is to be performed. For further details on commands of this type, refer to Section 2.5.3.

If the step processor display is ever scrolled off the screen (due to repeated entry of Immediate Mode commands), it may be redisplayed any time by entering STEP in Immediate Mode. This repeats the information, by indicating the next instruction to be executed.

NOTE: If a keyboard input statement (INPUT, INPUT, non-polling KEYIN) is executed while stepping through a program, the input request must be satisfied by entering the data value in the usual manner before further stepping proceeds.

Programs usually execute slower when Step Mode is ON. This is noticeable when Step Mode is ON and program lines outside the current Step Range are being executed. This is normal and program performance is fully restored when Step Mode is turned OFF.

Temporary Exit from Stepped Execution

While stepping through a program, if executing a program statement that is displayed leaves the Step Range, then depressing EXECUTE resumes normal execution of the program until program control returns to a line number within the Step Range. For example, if the next statement to be executed is: GOSUB 'Display and all the statements which are executed during the DEFFN' Display subroutine are outside the Step Range, pressing EXECUTE will call the 'Display routine, and halt after returning, showing the statement following the GOSUB' as the new next statement.

Besides the Step Range, there are certain other methods by which the programmer may selectively execute segments of program code normally, but then return to Stepped Execution once they are complete. These are the CONTINUE RETURN, CONTINUE NEXT and CONTINUE LOAD instructions.

CONTINUE RETURN

This instruction resumes execution of the program until the currently executing subroutine is completed (i.e., when the RETURN statement is encountered). At that point, Stepped Execution is reactivated. This is useful in avoiding Stepped Execution of lengthy and already tested subroutines.

NOTE: This command can only be executed when actually in a subroutine. Otherwise, a P41 error results, but this does not prevent program continuation.

CONTINUE NEXT

This instruction resumes execution of the program until the current active FOR/NEXT loop has been completed. At that point, Stepped Execution is reactivated. This is useful in avoiding Stepped Execution of iterative FOR/NEXT loops that have already been tested.

NOTE: This command can only be executed when actually in a FOR/NEXT loop. Otherwise a P40 No Corresponding FOR for NEXT Statement error results, but this does not prevent program continuation.

CONTINUE NEXT and CONTINUE RETURN work only with the highest level of the return stack. For example, executing CONTINUE NEXT while in a subroutine called from the FOR/NEXT loop does not work. This is because the subroutine call information is on the top of the stack at this point. Here, exit the subroutine first (with a CONTINUE RETURN) after which CONTINUE NEXT operates as expected.

CONTINUE LOAD

This command resumes normal execution of the program until the next LOAD statement has been executed. At this point, Stepped Execution is reactivated. This is useful for rapidly stepping through a lengthy chain of program overlays but preserving the option to inspect individual overlay steps.

DEFFN’

Stepped Execution is also temporarily deactivated during execution of DEFFN’ subroutines executed as a result of depressing a Special Function key in Immediate Mode. The subroutine is executed as if an implied CONTINUE RETURN statement were executed at the start of the subroutine. Stepped Execution is restored when the subroutine is completed (executing a RETURN instruction).

For further details refer to the Statements Guide: CONTINUE RETURN, CONTINUE NEXT, CONTINUE LOAD and DEFFN’.

Deactivating Stepped Execution

Stepped Execution may be deactivated any time by:

- Entering the CONTINUE command in Immediate Mode. When executed, the CONTINUE command sets Step Mode OFF and resumes normal execution of the program.

NOTE: CONTINUE does not modify the Step Range.

- Selecting the CONTINUE option on the HELP display. This implicitly executes a CONTINUE command. It is used largely as a convenience to the above method.

- Entering STEP OFF in Immediate Mode.

6.3.2 Logic Inspection Using TRACE

The Interpreter supports the execution of programs in a special mode called TRACE Mode. When TRACE Mode is invoked, the trace processor illustrates the logic of an executing program by displaying information on all transfers of control and certain variable modifications when they occur within the program. Information of this kind that is displayed by the TRACE processor is called TRACE output.

Various forms of the TRACE instruction allow for selective tracing of specified variables, program lines and marked subroutines to allow the programmer to concentrate on specific areas of interest while debugging programs.

TRACE Mode can be used with Stepped Execution by starting both TRACE Mode and Step Mode concurrently. By starting both modes, information about the execution of each statement is automatically displayed as each statement is executed. Either mode can be started or deactivated without affecting the other.

This section discusses the TRACE commands in general, and those forms of the trace command which relate to inspection of program logic. This includes the TRACE, TRACE #, TRACE ’, and TRACE OFF commands.

Activating TRACE Mode

Execution of any TRACE statement (with the exception of TRACE OFF) starts TRACE Mode. Once started, TRACE Mode remains in effect until TRACE OFF is executed or until CLEAR or HELP/RESET is executed. All TRACE modes are global in effect. That is to say that TRACE is either on or off, whatever module position or status.

All TRACE statements may be used as either a program statement or as an Immediate Mode command.

The TRACE statements and TRACE OFF may be used as program statements to logically surround areas of program code upon which tracing is required.

For example:

```
1000 TRACE
1010   .
1020   .
1030 (other program code here)
1040   .
1050   .
1060 TRACE OFF
```

When line 1000 is executed, TRACE Mode is activated and TRACE output is generated during execution of lines 1010 through 1050. When 1060 is executed, TRACE Mode is deactivated.

TRACE Mode may be activated by entering a TRACE command in Immediate Mode during program execution. Here, it is necessary to first interrupt the program by invoking Immediate Mode. This is done by a variety of methods, all of which are documented in Section 2.5. Of these, the most common method used is the HALT key. When depressed, program execution is halted and Immediate Mode is invoked. More importantly, HALT does not disturb the "execution status" of the program, and so allows Stepped Execution to take place. Once the TRACE command has been entered, execution may be resumed by depressing EXECUTE or entering the CONTINUE command. The TRACE output is then generated.

TRACE Output

TRACE Mode generates output on two occasions:

1. Whenever a transfer of control takes place within the executing program.

2. Whenever a variable value is modified in any "Trace-Sensitive" statements. Statements considered to be Trace-Sensitive are:

```
LET Assignment statement (alpha and numeric assignment)
FOR/TO
NEXT
GOSUB’ with parameters
LIMITS
READ
ON ERROR GOTO
```

TRACE Output on Transfer of Control

Whenever a transfer of control takes place in a traced program, the TRACE processor displays the message:

```
TRANSFER TO xxxx
```

Where xxxx is the line-number to which control is being transferred. Examples of statements that cause a transfer of control are GOTO, GOSUB, GOSUB’, RETURN, and NEXT.

For example:

```
:1000 IF A=0 THEN 1200           : REM skip next statement
    :    PRINT A                 : REM this is never exec'd
:1200 GOSUB 1500                 : REM call subroutine
:1210 GOSUB '20                  : REM call marked subroutine
:1220 STOP
:1500 REM
    : REM This is a subroutine
    : REM
    : RETURN
:1600 REM
    : REM This is a marked subroutine
    : REM
    : DEFFN'20
    : RETURN
:TRACE                     (<-- set TRACE Mode ON)
:RUN                       (<-- run the program)
TRANSFER TO 1200           (<-- TRACE output caused by line 1000)
TRANSFER TO 1500           (<-- TRACE output caused by line 1200)
TRANSFER TO 1200           (<-- TRACE output caused by line 1500)
TRANSFER TO 1600           (<-- TRACE output caused by line 1210)
TRANSFER TO 1210           (<-- TRACE output caused by line 1600)

STOP
:_
```

NOTE: Only GOTO/GOSUB line-number, NEXT, RETURN (from GOSUB, GOSUB', or PROCEDURE) and GOSUB' statements generate TRACE output for transfer of control. GOTO/GOSUB statement-label, RETURN (value), FUNCTION or PROCEDURE calls, and transfers generated by structured statements (WHILE/WEND, REPEAT/UNTIL, SWITCH/CASE, IF/ELSE/ENDIF) do not generate TRACE output.

Trace Output on Variable Modification

Whenever a variable is modified by a TRACE-sensitive statement, the TRACE processor displays the name of the variable and the value being assigned.

For example:

```
:10 A=500
:TRACE
:RUN
A= 500
:_
```

In the case of alpha-variables, the value is displayed both in ASCII and HEXADECIMAL format, 16 bytes per line up to 6 lines. A maximum of 96 bytes are displayable. If the value being assigned exceeds 96 bytes, an ellipsis ("...") is displayed following the value.

For example:

```
:10 DIM A$128
:20 A$="This will show TRACE output of a string that is being
        assigned to an alpha-variable that is longer than 96 bytes in
        length"
:TRACE
:RUN
A$=       "This will show T"  HEX(5468 6973 2077 696C 6C20 7368 6F77 2054)
STR(17)  "RACE output of a"  HEX(5241 4345 206F 7574 7075 7420 6F66 2061)
STR(33)  " string which is"  HEX(2073 7472 696E 6720 7768 6963 6820 6973)
STR(49)  " being assigned "  HEX(2062 6569 6E67 2061 7373 6967 6E65 6420)
STR(65)  "to an alpha-vari"  HEX(746F 2061 6E20 616C 7068 612D 7661 7269)
STR(81)  "able which is lo"  HEX(6162 6C65 2077 6869 6368 2069 7320 6C6F)
...
:
```

If the receiver is subscripted or is a STR() function, then this is shown in the TRACE output. As an additional aid, the calculated values of subscripts and STR arguments are also displayed. Also, multiple assignment statements are displayed one variable per line

For example:

```
:10 DIM A(7,7),A$(7,7)40
:20 X,Y=6
:30 A(X-1,Y+1)=30
:40 STR(A$(X,Y),X-4,Y+27)="a subscripted and STR assignment"
:TRACE
:RUN
Y= 6
X= 6
A(5,7)= 30
STR(A$(6,6),2,33)=
       "a subscripted an"  HEX(6120 7375 6273 6372 6970 7465 6420 616E)
STR(18)"d STR assignment"  HEX(6420 5354 5220 6173 7369 676E 6D65 6E74)
:
```

If a string value contains undisplayable characters, then each is replaced by a "." in the ASCII portion of the display; the hex portion is unaffected. Undisplayable characters are considered to be from HEX(00) to HEX(0F).

For example:

```
:10 A$=HEX(010203414243)
:TRACE
:RUN
A$=         "...ABC"             HEX(0102 0341 4243)
:_
```

NOTE: Only simple numeric and alpha assignment statements, NEXT statements, and GOSUB' (passing of parameters) generate TRACE output for modification of variables. Other statements, including numeric field assignments, parameter passing by function or procedure call, DATA LOAD, and $PACK/$UNPACK statements do not generate TRACE output for modification of variables.

TRACE output for numeric variables which are modified in other TRACE-sensitive statements are displayed as if modified in a LET statement. However, some minor differences exist. For example, if a NEXT statement is executed which exhausts the loop, then "NEXT END" is displayed.

For example:

```
:10 FOR I=1 TO 3
:20 NEXT I
:TRACE
:RUN
I= 1
I= 2
TRANSFER TO 20
I= 3
TRANSFER TO 20
I= 3
NEXT END
:_
```

Selective TRACE

There are other TRACE instructions which allow the programmer to refine TRACE output to specific areas of interest. These are called the selective TRACE statements. Selective TRACE statements used for tracing program logic are: TRACE # and TRACE ' (refer also to TRACE V, Section 6.4.2).

When using modules, references to specific entities by TRACE V, TRACE #, TRACE ' are not context sensitive. Any entity meeting the specified TRACE range requirement will be displayed. Thus, for example, if the command:

```
:TRACE # 1500,2000
```

has been executed, any module with line numbers within the range encountered during program execution will cause TRACE to invoke immediate mode.

The TRACE # instruction suppresses all TRACE output except that associated with transfer statements to program lines within the specified range of lines specified in the instruction. TRACE output is also enhanced by TRACE # in that the line number of the statement which causes the transfer is also displayed.

For example:

```
TRANSFER FROM 0800 TO 5400
```

TRACE ' suppresses all TRACE output except that associated with transfer of control using GOSUB' to any of the marked subroutines specified in the instruction.

For each of the selective TRACE statements, the system performs an automatic HALT after executing the statement that produced the TRACE output. Specifying the "*" parameter in the instruction shows that the system should not perform the HALT. HALTing due to selective TRACE output is also suppressed using CONTINUE NEXT, CONTINUE RETURN and CONTINUE LOAD, although this output is still printed.

When debugging programs, these selective TRACE statements are particularly useful to rapidly find out when a particular section of a program is "branched into," and where the branch initiated.

For example:

```
:TRACE # 1500,2000
:RUN
```

Assuming an appropriate program is in memory, entering these statements allows the developer to operate the program(s) normally until a statement in the range is executed. However, when any statement in the range 1500 to 2000 of the program is transferred to, TRACE output for the transfer is generated (e.g., TRANSFER FROM 0900 TO 1510), the program is halted and Immediate Mode is invoked. At this point the programmer may further inspect the circumstances of the transfer.

NOTE: TRACE # and TRACE ' generate TRACE output for transfers of control only. If a traced line number range or marked subroutine is encountered as a matter of normal sequential processing of program lines, TRACE output is not generated.

Trace commands are mutually exclusive. Therefore, entering a trace command of any type cancels the effect of the previous trace command entered.

Redirecting TRACE Output

TRACE output is always sent to the console output (CO) device address and as such may be redirected by the SELECT CO instruction. TRACE output can be directed to three types of output devices: the screen (the default), an ASCII file, or a printer. The output address, if not the screen, must be a printer class address (204 or from 210 to 21F) currently setup in the device equivalence table.

For example:

```
SELECT CO 215
```

For further details refer to the Statements Guide, SELECT.

6.3.3 Logic Inspection Using LIST

There are two LIST instructions that are useful when inspecting program logic: the LIST STACK and LIST DT instructions. Both instructions are legal as program statements and as Immediate Mode commands.

LIST STACK

The LIST STACK instruction produces a listing of all currently active FOR/NEXT loops, GOSUBs, FUNCTIONs and PROCEDUREs. Information is listed in reverse stack order (i.e., most recent stack entry is shown last).

For each stack entry, the program statement that caused the entry is displayed. If the stack entry is a FOR/NEXT loop entry, then the current values of the loop counter, loop limit and step value are also displayed. If the stack entry is a FUNCTION or PROCEDURE, the FUNCTION/PROCEDURE identifier name is displayed after the statement.

For example:

```
0010 DIM Total,I,X,Y
0020 X=3:Y=7
0030 FUNCTION 'AddNum(J,K)
    :   STOP #
    :   RETURN (J+K)
    : END FUNCTION
0050 FOR I=X*2 TO 100 STEP 2
    :   GOSUB 70
    : NEXT I
0060 END
0070 GOSUB'Stuff(X,Y)             : REM subroutine
0080 RETURN
0090 DEFFN'Stuff(A,B)             : REM subroutine 'Stuff
    :   Total='AddNum(A,B)
    : RETURN                      : REM STOP here so we can do LIST STACK
```

```
:RUN

STOP 0030
:LIST STACK
0050 FOR I=X*2 TO 100 STEP 2      (<-- output from LIST STACK)
    I=6, TO 100, STEP 2
0050 : GOSUB 70
0070 GOSUB 'Stuff(X,Y)
0090 : Total='AddNum(A,B)
'AddNum
:_
```

The order in which stack information is listed displays the current "nesting" of active FOR/NEXT loops, GOSUBs, and function calls. This is useful when using the CONTINUE NEXT and CONTINUE RETURN instructions. It can also be used to decide when CONTINUE NEXT and CONTINUE RETURN are legal, and if executed, to determine when control is returned to Immediate Mode.

For example, CONTINUE RETURN is only legal when the LIST STACK display shows a GOSUB, GOSUB' or FUNCTION entry. If CONTINUE RETURN is executed at that point, then control is returned to Immediate Mode at the statement following the last GOSUB or GOSUB' entry displayed by LIST STACK (assuming normal completion of the subroutine).

In addition, CONTINUE NEXT is only legal when the last item in the LIST STACK display is a FOR/TO entry. If CONTINUE NEXT is executed at that point, control is returned to Immediate Mode at the statement following the NEXT statement that terminates the loop (assuming normal completion of the loop).

LIST DT

The LIST DT instruction is useful in inspecting the recent program overlay sequence. The program load sequence message shown in the LIST DT display shows the names of up to the last six programs overlaid to produce the current program in memory. If more than six programs have been overlaid, then the first program and the last five programs are displayed only. Only programs loaded by name are displayed. Programs loaded using LOAD DA do not appear. In addition, LIST DT shows any modules that have been INCLUDEd, also which module is currently the active List Module and Run Module.

6.3.4 Logic Modification

It is possible to modify the logic flow of an executing NPL program from Immediate Mode without actually modifying the text of the program. Immediate Mode commands which allow this are:

GOTO
RETURN
RETURN CLEAR
NEXT
RUN [#]

Use of these commands usually requires that the programmer be aware of the status of the RETURN Stack. LIST STACK is useful for this purpose.

GOTO

Entering the GOTO command from Immediate Mode causes program execution to proceed at the specified line number or statement label, as opposed to the next statement internally scheduled by the Interpreter. When GOTO is entered followed by a line number or statement label, the system displays the specified program line, or the statement following the specified statement label. Continuing (or stepping through) the program at this point causes execution to proceed at the indicated statement.

RETURN

Entering RETURN in Immediate Mode during the execution of a subroutine causes the subroutine to RETURN at that point, and not when RETURN would normally be encountered in the program.

This is useful for prematurely terminating the execution of a subroutine, and to continue execution following the subroutine call. As with GOTO, the "new" next statement to be executed is displayed, and continuing or stepping proceeds starting at this statement.

RETURN CLEAR

Entering RETURN CLEAR in Immediate Mode deletes all RETURN Stack entries up to and including the most recent GOSUB entry. FUNCTION and PROCEDURE entries may not be deleted by RETURN CLEAR. That is, the most recent GOSUB call and all subsequent FOR/NEXT loop information are cleared from the stack. The point at which execution resumes (continued or stepped) does not change. Therefore, this is usually followed by entering a GOTO to redirect execution.

This is useful for prematurely terminating the execution of a subroutine when such a statement should have been included in the program, or when a logic modification transfer out of a subroutine has rendered the RETURN information superfluous (i.e., GOSUB or GOSUB’ entry in LIST STACK display) in the stack.

RETURN (value)

Entering RETURN (value) in Immediate Mode during execution of a FUNCTION causes the FUNCTION to prematurely exit with the return value supplied by the expression. Execution continues in the statement that made the function call, and halts before executing any further statements. This may be useful for prematurely terminating the execution of a function.

RETURN ERROR (code)

Entering RETURN ERROR (code) in Immediate Mode during execution of a FUNCTION or PROCEDURE causes the function to prematurely exit, raising the indicated error code to the statement that made the function or procedure call. If the statement that made the function call has error handling logic, program execution halts before executing any further statements, otherwise the statement which contains the function call will be displayed with the error code. This may be useful for prematurely terminating the execution of a FUNCTION or PROCEDURE when it is advisable to continue execution.

NEXT

Entering NEXT followed by the counter variable in Immediate Mode during the execution of a FOR/NEXT loop causes the counter variable to be incremented (or decremented) and execution to continue at the statement following the FOR/TO statement. This is useful for prematurely terminating execution of the current iteration of a FOR/NEXT loop.

NOTE: If the last iteration of a loop is in progress (when NEXT is entered), the NEXT stack entry is removed, but the point at which execution would resume is not changed (a GOTO is usually entered here).

NEXT is also valid for terminating FOR/NEXT loops entered as a multi-statement command in Immediate Mode.

For example:

```
:FOR I=1 TO 10: PRINT A(I) : NEXT I
```

When used in this manner, NEXT does not modify the program logic (assuming the index variable is not used by the program). NEXT only modifies program logic if the corresponding FOR/TO statement was executed as a program statement.

RUN #

Entering the RUN command followed by a line number (and an optional statement number) causes execution to proceed at the specified line number (and statement within line, if specified). In addition, the RETURN Stack is cleared. Therefore, RUN # cannot be used to resume execution within a subroutine or FOR/NEXT loop.

RUN

Entering the RUN command without a line number causes the RETURN Stack to be cleared. All non-common variables are reinitialized and execution resumes at the first program line.

6.4 Inspection/Modification of Program Variables

When a program's execution is halted by any means, program variable values may be inspected or modified using Immediate Mode commands. The Interpreter also provides a means by which variable modification may be automatically traced.

6.4.1 Variable Inspection

The following list summarizes those instructions that are used for the inspection of program variable values (detail on each command is found in the Statements Guide). Each of these instructions may be entered in Immediate Mode to display the contents of the variables of a halted program. All references to variables will refer to variables within scope at the point of execution within the current executing module.

Commands useful for variable inspection include:

- **PRINT**--prints the contents of the specified variables in zoned format.
- **PRINTUSING**--prints the contents of the specified variables according to the specified format image. If image is specified as an image statement, then the program must be resolved.
- **MAT PRINT**--prints the contents of the specified alpha or numeric arrays. One dimensional arrays are displayed with one element per line of the screen. Two dimensional arrays are displayed one row per line.

- **LIST DIM**--displays the contents of all or selected variables which are currently defined (both common and non-common) in the variable stack, in alphabetical order by variable name.

NOTE: Not all variables listed are necessarily referenced by the current program (e.g., common variables from previous programs, or variables defined due to Immediate Mode commands). LIST V generates a list of only those variables that are referenced by the current program.

- **LIST STACK DIM**--same as LIST DIM except variables are displayed in stack order (variables most recently defined by the program appear first).

  The values of variables specified in the commands are retrieved from the variable stack of the halted program if they are currently defined by the program.

NOTE: If a scalar variable is referenced in an Immediate Mode command that was not defined by the program, it is defined in the variable stack when the Immediate Mode command is executed (with default size and value). Although this does not usually affect the integrity of the program's execution, it consumes variable segment space and should be avoided unless intended. Byte 38 of $OPTIONS may be used to suppress the implicit declaration of numeric and string scalars.

The syntax of many of the above instructions supports the evaluation of numeric-expressions. Therefore, since computations are allowed, the programmer can easily manipulate data in a way that renders it more meaningful for debugging purposes.

For example, entering the following command in Immediate Mode:

```
:PRINT A,STR(B$,1,Q),VAL(C$,3)*2^8
```

prints the values of A, the first Q bytes of B$, and the numeric value of C$ times 2 to the eighth power. If the variables A, B$, C$ and Q are already defined (by the resident and executed NPL program), then the current values of these variables are used in the computation for the result.

Variable inspection may be further enhanced by using other statements such as those that perform looping, conditionals, data conversion, etc.

For example:

```
:FOR I=1 TO Q : PRINT VAL(STR(A$(I),20,2)) : NEXT I
```

This command entered in Immediate Mode prints the numeric value of the 21st and 22nd bytes of the first Q elements of A$(). The value of Q is taken from the current variable stack.

NOTE: The value of I is modified by this statement and care should have been taken that the value of I is not depended upon by the program.

When it is found that lengthy display statements are being used repeatedly to inspect variables, it is often useful to include the statement(s) as either a DEFFN' text definition or a DEFFN' subroutine (if the display procedure is more lengthy) in the program, by that allowing that it may be invoked with a single SF Key.

NOTE: This must be done prior to the initial execution of the program since entering program statements during execution disturbs the "execution status" of the program (for further details on the use of DEFFN' text definition, refer to Section 5.4.5).

For example:

```
:0000 DEFFN'0<
: I=L                    : REM set I to top of linked list
:0001 IF I=0 THEN 2      : REM quit when link of 0 found<
:    PRINT I,L1$(I)      : REM print data at node I<
:    I=VAL(L$(I),2)      : REM link to next node<
:    GOTO 1              : REM iterate
:0002 PRINT "END OF LIST"<
: RETURN
```

This example displays the contents of L1$() in the logical order defined by the linked list contained in L$(). The top pointer to the linked list is assumed to be in L. This subroutine may be invoked from Immediate Mode by depressing SF key number 0 (presuming the SF keys are in DEFFN Mode). When the subroutine is completed (i.e., the linked list is displayed) control returns to Immediate Mode.

It is also often useful to include display instructions in program text at strategic points to repeatedly report on the contents of important variables during program execution.

6.4.2 Automatic Variable Inspection with TRACE

As discussed in Section 6.3.2, the TRACE V instructions activate a special mode of program execution called TRACE Mode. When TRACE Mode is active, the TRACE processor displays information on transfer of control, and the name and value of variables whenever a variable is modified in a Trace-Sensitive statement of an executing program.

This section discusses a special form of the TRACE command that allows for selective variable modification "traps."

TRACE V

TRACE V is a derivative of the general TRACE instruction. TRACE V provides a means by which the programmer can refine TRACE output to concentrate on a specific variable or alphabetical range of variables.

The general form of the instruction is:

```
TRACE V [*] [var1][,[var2]]
```

TRACE V may be used in a program statement or entered as an Immediate Mode command. When executed, TRACE V starts TRACE Mode and suppresses all TRACE output except that associated with the modification of the specified variable or alphabetic range of variables.

When a variable in the specified range is modified by a TRACE-sensitive statement in the executing program, the system produces TRACE output for the variable and then performs an automatic HALT, at which point Immediate Mode is invoked (for further details on TRACE output for variables, refer to Section 6.4.2).

Specifying the "*" parameter in the instruction suggests that the system should not perform the HALT. HALTing due to selective TRACE output is also suppressed during CONTINUE NEXT, CONTINUE RETURN and CONTINUE LOAD, although this output is still be printed.

As with other selective TRACE commands, references to specific entities are not context sensitive. Any entity meeting the specified TRACE range requirement will be displayed.

For example:

```
:TRACE V A$
```

If this statement was executed, any TRACE-sensitive assignment to any variable named A$ will be displayed as encountered during execution, even if different actual variables are involved. In other words, a FUNCTION’s local A$ and a module’s static A$ would both fulfill the range, but are completely separate variables.

When debugging programs, the TRACE V statement is particularly useful to rapidly determine at which point a particular variable is being modified by a program or sequence of programs.

For example:

```
:TRACE V Q$
:RUN
```

Assuming an appropriate program is in memory, entering these statements allows the programmer to operate the program normally. However, when any TRACE-sensitive assignment of the program(s) modifies the Q$ variable, TRACE output on Q$ is generated, the program is HALTed and Immediate Mode is invoked. At this point the programmer may further inspect the circumstances of the modification using other Immediate Mode commands.

NOTE: TRACE V generates TRACE output only when a variable is modified in a TRACE-sensitive statement. There are other statements in NPL that may modify a variable but that is not reported in TRACE output (e.g., DATALOAD, $UNPACK, INPUT, etc.).

6.4.3 Variable Modification

Program variable values may be modified any time by halting the program and executing an Immediate Mode command that modifies variable content. Any instruction in the NPL language that modifies variable content is legal as an Immediate Mode command.

This includes assignment statements (LET alpha and numeric), keyboard input statements (INPUT, INPUT, non-polling KEYIN...), data conversion (PACK, UNPACK, CONVERT...), data movement (MAT COPY, MAT MOVE, MAT SORT), diskimage I/O statements (DATALOAD DA, DC, BA), etc.

Whenever a variable is assigned a value in an Immediate Mode command, the value is assigned to the defined variables stack in the variable segment of the halted program. Therefore, when execution of the program is continued, the new value of the variable is used in any subsequently executed program logic.

If a scalar variable is referenced in an Immediate Mode command that is not defined in the current variable stack, then the variable is added to the stack with the new value. $OPTIONS byte 38 may be used to suppress the implicit declaration of numeric and string scalars. Any such default declarations, or any explicit DIM statements entered in Immediate Mode, can only be used to declare static variables in the current Executing Module. Recursive variables may not be declared in Immediate Mode.

For example:

```
:STR(B$,,Q)=ALL("X")
```

Sets the first Q bytes of program variable B$ to "X"s.

```
:READ X,Y
```

Reads the next two data values (as specified by DATA statements in the program) to program variables X and Y. The DATA pointer is also advanced by two elements.

```
:FOR I=1 TO 10: INPUT A(I) : NEXT I
```

Inputs data values from the keyboard into the elements of A().

NOTE: The value of I is also modified.

6.5 Inspection of Program Text

At any time while in Immediate Mode, program text of the HALTed program may be listed without disturbing the execution status of the program.

6.5.1 Inspection Using the LIST Commands

Commands used for inspection of program text included:

- **LIST**-- Lists the program text currently in memory one screen at a time (touching return advances to next screen). Specific segments of a program may be listed by specifying program line number ranges in the command.
- **LIST D**-- Same as LIST except multi-statement program lines are "decompressed" to show one statement per physical screen line, irrespective of the manner in which the line was originally entered. Additionally, LIST D performs an internal cross-reference on program line numbers and displays a "-" prior to the line number on the display if it is referenced by any statement in the program.
- **LIST V**-- Produces a cross-reference listing of all or specified variables referenced in program text. For each variable that is referenced in a program, the cross-reference listing may show program line numbers of the reference, or optionally, the full program line.

- **LIST #**-- Produces a cross-reference listing of references to all or specified program line numbers.
- **LIST '**-- Produces a cross-reference listing of references to all or specified DEFFN' subroutines in the program or in external subroutine libraries.
- **LIST T**-- Finds all occurrences of a specified text string in a program. Either program line numbers, or optionally, the full program line, may be displayed for each occurrence of the text string.
- **LIST (Special Identifier Type)**-- Several LIST statements are available to produce cross-reference listings of special identifier types. Output is similar to LIST V, and the cross-reference is only for the current List Module.

  LIST FUNCTION lists all (or specified) FUNCTION identifier references.

  LIST PROCEDURE lists all (or specified) PROCEDURE identifier references.

  LIST RECORD lists all (or specified) RECORD identifier references.

  LIST FIELD lists all (or specified) FIELD identifier references.

  LIST lists all (or specified) statement label references.

The cross-referencing LIST statements (i.e., LIST V, LIST #, etc.) ignore all program statements that contain syntax errors.

Refer to the Statements Guide for more detailed information on the use of LIST statements.

6.5.2 Redirecting LIST Output

Output displayed from LIST commands is always directed to the device address specified by the LIST device. The current selection for the LIST device can be determined by entering:

```
X$=$SELECT(LIST):PRINT X$
```

in Immediate Mode. The current selection for the LIST device can be modified by the SELECT LIST command.

SELECT LIST may be used to direct LIST output to the screen, a printer or an ASCII file within the native file system.

For example:

```
SELECT LIST 215
```

After executing this command, all LIST output is directed to device address 215 (a printer or an ASCII file). The output address, if not the screen, must be a printer type address (204 or 210 to 21F) currently set up in the device equivalence table. The default address at boot time is address 005 (the screen).

For further details refer to the Statements Guide, SELECT LIST and LIST DT.

6.6 Inspection of the Program Environment

There are many useful Immediate Mode commands and system variables that yield information about the environment in which the program is executing.

These are summarized here and detailed in the NPL Statements Guide.

6.6.1 Commands

The following commands can be used to inspect or modify the NPL programming environment.

- **LIST DT**--Output from this command is particularly useful in inspecting the environment, as it shows how NPL is currently mapped to the native operating system in one display.

  This also displays the status of the Device Table and Device Equivalence Table. This is a display of all currently selected device addresses and opened data files, and the current device address mapping to native operating system files and devices. In addition, the current List Module and Run Module, as well as a list of all INCLUDEd modules are displayed.

- **LIST DC**--Displays the contents of the diskimage at a specified device address. Programs, data files, scratched programs and scratched data files are all listed. The use of wild cards and relational expressions provides for listing files according to a certain name, type, location or date stamp, or combination of that.

- **$SHELL**--Native operating system shell commands may be executed directly from Immediate Mode in NPL. This is useful in locating NPL diskimages, BOOT programs and native environment information. For further details on $SHELL, refer to Chapter 8, in addition to the Statements Guide.

- **SPACE**--The SPACE instruction and its derivatives (SPACEF, SPACEK, and SPACEW) provide information on the available program text area and program variable area.

6.6.2 System Variables

All System Variables may be inspected any time from Immediate Mode or under program control. Many of these system variables may also be modified under the conditions documented in the Statements Guide.

$BOXTABLE	Indicates type of screen "boxes" currently enabled and character segments used to form character boxes.
$KEEPREMS	Contains option values for the handling of REMS, spacing and return-graphics.
$KEYBOARD	Contains the current keyboard translation table.
$MACHINE	Contains basic information about the type of hardware currently hosting NPL and the version of NPL.
$NETID	Contains the current physical (8-byte) node address of the workstation on Novell networks.
$OPTIONS	Contains numerous options in general which control the behavior of the RunTime program under designated situations.
$PROGRAM	Contains information on the program load sequence of all currently loaded overlays.

$PSTAT	Contains general status information about each active NPL process on the system.
$REV	Contains the official 12-digit NPL Revision Code or the RunTime.
$SCREEN	Contains the current screen translation table.
#TERM, #PART, #ID	Contains the unique identification numbers for each user on multi-user systems.

Refer to Chapter 2 of the NPL Statements Guide for more details on these system variables.

6.6.3 Public Functions, Procedures and Variables

Available library functions, procedures, records, marked subroutines (DEFFN's) and variables may be browsed using the LIST PUBLIC statements. All public entries in currently loaded and resolved modules may be displayed by these statements.

The LIST PUBLIC statement displays the declaration and location of the PUBLIC variable types in all currently loaded and resolved PUBLIC sections in the workspace. This requires that all modules be INCLUDEd by a previous RUN of the program, or with an Immediate Mode INCLUDE statement before this statement can be issued. The LIST PUBLIC statement is especially useful to select new public names that would not conflict with currently loaded declarations, as well as a reminder of public references available when programming. Special Identifier Types available for the LIST PUBLIC statement include:

FUNCTION
PROCEDURE
RECORD
FIELD
DEFFN
V

The output of the statement shows the name of the indicated identifiers, preceded by the module name in which the identifier is declared, and the name of any PUBLIC section in which it was declared, if applicable.

All of the above program LIST statements apply only to the program text in the current List Module.

NOTE: All PUBLIC identifiers displayed with LIST PUBLIC are shown no matter whether or not the referenced module is scramble-protected.

CHAPTER 7

DEVICE SUPPORT

7.1 Overview

This chapter discusses the method by which peripheral devices are accessed by NPL programs. Devices are controlled by a device emulation scheme which provides high compatibility with software originally written for the Wang 2200 computer system. As important, it also provides a common device controlling scheme across varied hardware and native operating system environments which NPL supports. As a result, porting NPL application software from one environment to another requires minimal or no changes to application code. This chapter deals primarily with this subject and is essential to an understanding of NPL.

NOTE: Subtle differences do exist for specific hardware and operating environments. These differences are documented in a series of operating system and environment-specific NPL Supplements which should be used in conjunction to this documentation.

Section 7.2 provides a general overview to the NPL device addressing scheme and to the method of equating logical NPL device addresses with physical native operating system files and devices.

Section 7.3 discusses storage devices.

Section 7.4 discusses screen handling.

Section 7.5 discusses keyboard support.

Section 7.6 discusses mouse support.

Section 7.7 discusses terminal/monitor support.

Section 7.8 discusses printers.

Section 7.9 discusses serial devices.

7.2 Device Emulation

As discussed in Chapter 1, NPL uses a hardware independent method of accessing peripheral devices. This scheme allows developers to reference external peripheral devices in a generic fashion which in turn allows highly portable application to be developed. The following sections discuss NPL device addressing in detail.

- Section 7.2.1 discusses NPL device addresses.
- NPL devices are *logical* devices, not *physical* devices. Each NPL *logical* device is mapped to a *physical* native operating system peripheral device or filename. Section 7.2.2 discusses how NPL *logical* device addresses are established using the Device Equivalence Table (DET).
- In addition to device addresses, NPL provides developers with a means of establishing default addresses for subsequent use by their NPL routines. This is accomplished by the Internal Device TABLE (DT) and is discussed in Section 7.2.3.
- NPL also allows developers to gain exclusive access to selected devices. This is essential in multi-user environments. This is discussed in Section 7.2.4.

7.2.1 Device Addressing

Each NPL peripheral device is accessed by a unique three character device address. Each character of the device address must be an ASCII representation of a hexadecimal digit (numbers 0-9 or letters A-F). Each class of peripheral has a certain range of acceptable device addresses. These are discussed in more detail in the sections below dealing with the classes of peripherals. In addition, for some classes of devices, notably print devices, the actual address is specified by the second and third character of the device address. The first character is used to control the format of output to the device. This topic is discussed in greater detail in sections below dealing with specific peripherals.

NOTE: This implementation of device addresses is fully compatible with the Wang 2200.

7.2.2 The Device Equivalency Table (DET)

For NPL I/O routines to function properly, the I/O operation must be directed to the correct native operating system file or device. The Device Equivalence Table (DET) is used to establish an equivalence between the NPL *logical* device addresses and the equivalent *physical* files or devices maintained within the native operating system for the particular hardware.

For each NPL device address used by a given application, the DET contains an entry with that particular address, and the equivalent native operating system file or device name. For example, the following 3 character device addresses could be equated to the corresponding native devices or files.

D10	PLATTER1.BS2
D20	PROGRAMS.BS2
215	LPT1
216	COM2
204	WORK.TXT

During program execution, whenever the RunTime Program encounters a *logical* NPL device address in program syntax, the DET is inspected to determine the equivalent *physical* native operating system device or file name that it references. The I/O operation is then directed to that file or device. If it is the first access to the specified NPL device address since the device equivalence was established, a logical open is performed for the native operating system file or device before the I/O operation is performed. This mapping operation is entirely transparent to the NPL program.

The $DEVICE Statement

The Device Equivalence Table is setup at execution time through the use of the $DEVICE statement. This is usually accomplished either the startup BOOT or PREBOOT programs before any application code is executed. Refer to Section 2.4 and 5.3 for further information on the startup BOOT and PREBOOT programs.

At any point during execution of NPL programs, the DET may be modified and inspected, again through the use of the $DEVICE statement.

The general format the $DEVICE statement is:

General Form:

Form 1:

```
$DEVICE({device-address})=alpha-expression
        {file-number    }
        {alpha-variable }
```

Form 2:

```
alpha-receiver=$DEVICE({device-address})
                       {file-number   }
                       {alpha-variable}
```

Where:

```
alpha-variable   =contains a valid NPL device-address or file-num-
                  ber (with no preceding slash).

alpha-expression =an alpha-expression which evaluates to a native
                  operating system file-specification or device-
                  specification followed by one or more optional
                  clauses. The total resultant value of the alpha-
                  expression should not exceed 50 characters in
                  length. One blank space separates the file or de-
                  vice-name and each optional clause.

clause           =[1.2={N,Y}           ]
                  [1.4={N,Y}           ]
                  [2.8={N,Y]           ]
                  [360={N,Y}           ]
                  [720={N,Y}           ]
                  [ALF={N,Y}           ]
                  [ERR={N,Y}           ]
                  [EXT={N,Y}           ]
                  [LCL={N,Y}           ]
                  [PES=numeric-constant]
                 [SES=numeric-constant]
                 [TMO={N,Y}           ]
                 [XLA={N,Y}           ]
```

NOTE: "File numbers" are established in the Internal Device Table. Refer to Section 7.2.3 below for details on the Internal Device Table.

The $DEVICE statement can also be used to set certain I/O options for accessing the specified native operating system file or device. Refer to the Statements Guide, $DEVICE, for details.

NOTE: Native operating environment file and device naming conventions are extremely implementation specific. The examples shown below and throughout this guide reference IBM MS-DOS file naming conventions. Refer to the appropriate NPL Supplement for details on naming conventions for the machine being used.

For example:

```
$DEVICE(/D11)="PLATTER1.BS2"
```

equates the NPL disk address D11 with the native operating system file PLATTER1.BS2. Any subsequent disk I/O directed to address D11 is directed to file PLATTER1.BS2.

```
$DEVICE(/215)="LPT1"
```

equates the NPL printer address 215 with the native operating system device LPT1. Any subsequent printer output directed to address 215 is directed to device LPT1.

This method of device handling ensures that I/O statements in existing Wang 2200 Basic-2 programs require no modification. It also ensures that device addressing methods used internally by NPL applications are 100% portable to any NPL supported operating system. The job of "software conversion" for I/O is reduced to adapting the $DEVICE statements in a startup program, to the appropriate native operating system file and device naming conventions.

Default Device Equivalence Table

When the NPL RunTime program is initially executed, the Device Equivalence Table automatically contains a series of default entries. Device addresses D10, D11, D12, 215, and 204 are defined. Refer to Chapter 5 of the appropriate NPL Supplement for specifics on which native operating system files or devices are assigned to these default devices.

Maximum Entries Allowed in the Device Equivalence Table

The default maximum number of entries addressable by the NPL Device Table at startup is 16. This default value can be overridden by the RunTime's /D startup option (refer to Section 2.4). The /D option allows the developer to specify the maximum number of Device Equivalence Table entries required by an application, in the range of 16 to 255.

The maximum number of DET entries in effect is stored in byte 16 of $MACHINE. If this maximum number of entries is exceeded, a non-recoverable error (A02) is generated. In addition, byte 17 of $OPTIONS contains the current number of DET entries defined (non-blank addresses). Refer to the Statements Manual, $MACHINE, for details.

Inspecting the Device Equivalence Table

Existing entries in the DET can be examined by use of $DEVICE on the right side of a LET statement.

For example:

```
X$=$DEVICE(/D11)
```

places the currently defined device equivalence for address D11 in the variable X$.

This allows programs to retain prior values of the DET before modifying the DET so that the original values can be restored at a later point.

NOTE: If a specified device address is not defined in the DET, the receiver variable is set to blank.

The DET may also be examined in physical order without knowing specific addresses defined by use of the $DET(x) statement. $DET(x) returns the device-address defined for the specified (x) physical entry in DET. Refer to the Statements Guide, $DET, for details.

Modifying the DET

The DET is "keyed" by device address. To modify an existing entry for a particular address, simply execute a $DEVICE statement to establish the new equivalence.

For example:

```
10 $DEVICE (/D11)="PLATTER1.BS2"
.
.
.
8000 $DEVICE (/D11)="DATA2.BS2"
```

NOTE: Execution of a $DEVICE statement which references an existing entry (i.e., an address already defined) causes a logical CLOSE of the original native operating system file or device equated with the NPL address. The new native operating system file or device is opened when the next I/O instruction to the device address is executed.

Clearing Entries in the Device Equivalence Table

To remove an entry from the Device Equivalence Table, or reuse a "slot" in the table with an NPL address not currently defined, the device equivalence entry must be cleared before it can be available for new devices. The format of the statement for clearing device entries is:

```
$DEVICE(/xxx)=" " or blank alpha-variable
```

where xxx is the address to be cleared

NOTE: The default devices which have been setup by the system must be cleared before any of those particular "slots" would become available for use.

7.2.3 The Internal Device Table (DT)

In many cases, NPL I/O statements do not explicitly specify an NPL device address. Instead, reference to devices is based on the program modifiable Internal Device Table (DT). The Internal Device Table contains a series of mnemonic device specifications, for different default peripheral devices, which are equated to NPL addresses by use of the SELECT statement. In addition to the mnemonic device specifications, the Internal Device Table contains up to 256 file number slots for use with disk device addresses which are also equated to NPL addresses by the SELECT statement.

Use of the DT for certain disk I/O operations is optional, but use of other I/O operations requires that valid entries be present in the DT. That is, some disk I/O statements allow the disk device address to be specified explicitly in the statement itself. Other disk I/O statements and most other I/O operations use the implicit device addresses specified in the DT. This topic is covered in more detail in the sections dealing with specific devices below.

NOTE: The Internal Device Table contains other information which affects program operation, but is not directly related to I/O. These parameters are established by the SELECT statement. For further details refer to the NPL Statements Guide, SELECT.

Default Devices

The DT maintains a series of default device addresses for use in I/O operations where a specific address is not specified. This feature enables an application to set up default device addresses in one program and access these defaults in subsequent programs. Thus, the logic for assigning device addresses can be centralized and therefore easy to modify should the need arise. The modifiable default devices maintained by the internal device table are:

DISK — Used for the input and output of all disk related operations directed to the primary (default) disk device.

NOTE: Up to 255 additional disk devices can be established by use of the disk file slots (refer to "File Number Slots" below in this section). The default DISK address is /D11.

CO (Console Output) — Used for TRACE output when debugging. The default CO address is /005--the terminal screen.

PRINT — Used for the output of PRINT and PRINTUSING statements executed during program operation.

NOTE: Immediate Mode PRINT statements are always directed to the terminal screen (address /005) regardless of the default PRINT selection. The default PRINT address is /005--the terminal screen.

LIST — Used for the output of LIST commands. The default LIST address is /005--the terminal screen.

LOG — Used for output of keyboard logging. The default LOG address is /000--the null device.

The DT also maintains default line width values for the PRINT, CO, LIST LINE, and LIST devices. The default value for line width is 80.

NOTE: Other entries for default devices are maintained for purposes of compatibility with the Wang 2200. These other default devices may be modified by an NPL program but, in general, do not affect operation of the NPL program. For further details refer to the Statements Guide, SELECT.

File Number Slots

In addition to the default devices described above, the DT maintains between 16 and 256 file number slots. The number of file number slots is dynamic and is controlled by the SELECT DISK/FILE NUMBER statement. Refer to the Statements Guide, SELECT DISK/FILE NUMBER, for further information on establishing the number of file number slots.

These slots are referenced by file number: #0 to #255. Each file slot can maintain one device address. Once a disk device address is assigned to a file number slot, subsequent I/O operations directed to that file number are directed to the corresponding device address. The same device address may be assigned to multiple file number slots.

NOTE: The default DISK parameter is file slot #0. In addition to maintaining disk device addresses, the file number slots can be used for a class of disk operations referred to as CATALOG operations. This topic is discussed in further detail in Section 7.3.8 below.

Modifying the Internal Device Table

Both default device addresses and file number slots within the DT may be modified explicitly by use of the SELECT statement or implicitly by use of the RESET function, CLEAR instructions or LOAD RUN instruction.

Use of the SELECT Statement

Addresses (and line width for PRINT, LIST, and CO) for DT entries may be specified by use of the SELECT statement.

For example:

```
10 SELECT PRINT /215(132)
```

Causes subsequent PRINT output (under program control) to be directed to address 215 with a line width of 132 characters.

```
20 PRINT "The amount due is ";A
```

Prints the literal string "The amount due is " followed by the contents of variable A to the native operating system device or file equated to address 215.

```
10 SELECT DISK /D35
```

Causes subsequent disk I/O where an address or file number is not specified to be directed to address D35.

```
20 DATA LOAD BAT (100) X$()
```

Loads the contents of sector 100 of the native operating system device or file equated (using the DET) with address D35 into variable X$().

```
10 SELECT #2 /D20
```

Causes subsequent disk I/O directed to file #2 to be directed to address D20.

```
20 DATA LOAD BAT #2, (84) X$()
```

Loads the contents of sector 84 of the native operating system device or file equated (using the DET) with address D20 into variable X$().

Other Functions and Statements

Use of the RESET function effects the DT as follows:

- The CO entry is set to device /005. The line width for the CO entry is set to 80 columns.
- The RESET function can be accessed from the HELP processor when using the interpretive RunTime (RTI). For further details refer to the Statements Guide, RESET. For further details on the HELP processor, refer to Section 11.7 of the Programmer's Guide.
- Use of the CLEAR statement with no parameters, or a LOAD RUN statement, affects the DT as follows:

1. PRINT and LIST entries are set to the current device address and line width of CO.
2. File slot #0 is cleared except for the DISK device address.
3. File slots greater than #1 - #255 are cleared entirely.

NOTE: The effects are the same whether the statements are used under program control or in Immediate Mode.

Examining the DT

The DT may be listed to the screen or printer by use of the LIST DT statement or command. For further details refer to the Statements Guide, LIST DT.

Individual entries within the DT may be examined under program control by the $SELECT function.

For example:

```
10 X$=SELECT DISK
10 X$=SELECT #4
10 X$=SELECT LIST
```

For further details on $SELECT, refer to the Statements Guide, $SELECT.

Module Status

The $PROGRAM and "Program Load Sequence" information displayed in LIST DT applies to program overlays loaded in current RUN module only and does not show additional modules loaded from INCLUDE statements.

NOTE: The executing MODULE is not displayed unless the program is currently HALTed and could be CONTINUEd or STEPped.

Following the Program Load sequence, LIST DT displays all modules (except the root module) which are currently Included in the workspace, and the current state of the each module. Refer to the Statements Guide, LIST DT and $PROGRAM for details.

Wang 2200 Compatibility Note

The use of the DT is almost fully compatible with the Wang 2200 Basic-2 language. For further details on SELECT, and full details on NPL variance on the implementation of the DT, refer to the Statements Guide, SELECT. In addition, the LIST DT command has been enhanced under NPL. For further details refer to the Statements Guide, LIST DT.

7.2.4 Exclusive Access

NPL provides a method of obtaining exclusive access on a dynamic basis to specified devices. This is an essential component of multi-user environments. Typically exclusive access would be used to:

- Reserve a printer or other output device for the duration of a report.
- Temporarily restrict other users from accessing a certain disk device during a critical update.

Exclusive access to specified devices is obtained by the $OPEN statement.

The $OPEN Statement

Use of the $OPEN statement "hogs" a particular device for the user issuing the request, so that all subsequent read or write operations to that device are reserved for that user. This access to the device remains "hogged" by the user until the device is released by one of the following events:

- A $CLOSE statement is executed for that device or for all devices.
- The HELP processor is invoked. For further details on the HELP processor, refer to Chapter 11.
- The RunTime program is exited.
- An END, CLEAR (with no parameters), or LOAD RUN statement is executed.
- A $DEVICE statement for the OPENed device is executed.

NOTE: A disk device is always accessed exclusively for the duration of any NPL statement. The developer must only be concerned about explicitly obtaining exclusive access (using $OPEN) when multiple NPL statements must be executed against a specific disk device with no intervening access by other users.

However, print class devices are not necessarily accessed exclusively for the duration of a single NPL statement. Although this is not usually a problem (most programs would use multiple PRINT statements and therefore would contain the required $OPEN statements), this can cause a problem with stand-alone statements such as LIST, which may generate multiple pages of output.
For further details refer to the Statements Guide: $CLOSE, $OPEN, END, CLEAR, LOAD RUN, and $DEVICE.

Implied Exclusive Access

All of the exclusive access statements in the above discussion, explicitly "hog" a specific device and in turn, require it to be explicitly released using the methods discussed.

In addition to these explicit commands, a number of NPL disk I/O commands will perform implied $OPEN/$CLOSE operations. In multi-user environments these implied "hogs" are a necessity, especially for write operations. For less critical read operations, programmer's may choose to suppress the implied $OPEN/$CLOSE operations in certain environments by use of byte 39 of $OPTIONS.. Suppression of these implied operations can provide a performance increase in disk I/O. Refer to the NPL Statements Guide, $OPTIONS, for details of these features.

7.3 Storage Devices

Disk I/O for NPL Applications can be implemented in three ways. The first, and most common way provides for a high degree of compatibility between the NPL environment and the Wang 2200/CS environment. As important is that it provides a common access method across the various hardware/operating system platforms which are supported by NPL. This allows the porting between NPL platforms to be done with little or no change to the application code.

The second method is an extension to the disk I/O routines supported in NPL. This extension is the support for diskimage files larger than 16MB. Because use of this extended capability has implications that affect the entire discussion of disk I/O, it is discussed separately in Section 7.3.10. Programmers who do not intend to use extended diskimage files may disregard Section 7.3.10. Programmers who do intend to use extended diskimages should study this section carefully.

NOTE: Sections 7.3.1 through 7.3.9 contain information that is common to both extended and non-extended diskimages as well as information that is specific to use of non-extended diskimages.

The third method of performing disk I/O for NPL applications is to use the Niakwa Data Manager (NDM). NDM allows developers to create, save and access application data files in native ISAM file formats such as Novell's Btrieve or Informix's C-ISAM. Data files maintained in native ISAM formats provide the programmer with several powerful benefits. First and foremost, file maintenance is virtually eliminated since the native ISAM routines are self maintaining and files are expanded dynamically. Second, performance of disk I/O to data files is dramatically improved by taking advantage of the highly optimized, modern ISAM's. And finally, end user data files become accessible to any third party product that can access the native ISAM files supported by NDM. Thus, allowing developers to provide their end-users with additional services by introducing them to new productivity tools that were once inaccessible to them in the NPL environment.

NOTE: The first two access methods described above are discussed in detail below. For information regarding the Niakwa Data Manager, refer to the NDM Programmer's Guide, or contact Niakwa.

7.3.1 NPL Diskimages

The fundamental way in which information, both programs and data, is stored on disk for use with NPL is the diskimage.

An NPL diskimage is a *logical* disk device which may be equated with a *physical* native operating system file or device by use of the Device Equivalence Table.

7.3.2 General Features of Diskimages

The following are features of all NPL diskimages.

Sector Size

All access to NPL diskimages uses a logical sector size of 256 bytes. This is unchanged regardless of the physical sector size of the native operating system disk being used. NPL internally handles any translation required.

NOTE: Some diskimage operations are multi-sector operations. That is, a number of 256 byte sectors may be read or written with a single instruction. However, it is not possible to read or write to a diskimage in less than 256 byte increments.

All references to sector numbers within a diskimage are totally unrelated to the physical sector numbers of the equivalent native operating system file or device. Thus, the NPL program is not concerned with the absolute location of NPL diskimages within the native operating system file structure.

Maximum Size

Unless extended diskimages are in use, the maximum size for any NPL diskimage is 65535 sectors (16MB). Smaller sizes may be specified. Refer to Section 7.3.10 for details on extended diskimages.

Maximum Number of Diskimages

Each diskimage requires an entry in the Device Equivalences Table to establish the native operating system file to be used. The maximum number of diskimages that may be accessed at one time is therefore limited by the defined size of the DET (refer to Section 7.2.2 for details on the DET) less the number of DET entries used for print class devices. Additional restrictions on the number of diskimages that can be accessed at one time may be imposed by the open file limits of the host operating system. Refer to Chapter 2 of the appropriate NPL Supplement for details.

NOTE: The above restrictions can be avoided when necessary by dynamic assignment of DET entries. The DET is established on a per user basis.

File Storage

A single diskimage may contain up to 4079 individual files (programs or data). Each diskimage has an internal index which maintains file status information. By combining large numbers of files into a single native operating system "diskimage" file, several benefits are gained:

Entire applications consisting of hundreds of programs and data files can be easily backed up as single native operating system "diskimage" file.

Excessive disk space due to native file system minimum allocation units is avoided.

Compatibility

All disk I/O operations are compatible across all machines on which NPL operates. In addition, compatibility with the Wang 2200 Basic-2 language is maintained.

NOTE: Compatibility with the Wang 2200 Basic-2 language is not maintained for statements which address areas of a diskimage beyond sector 65535. Refer to Section 7.3.10 for details on extended (non-standard) diskimages.

Compatibility with Native Operating System File Structures

The use of NPL diskimages on any machine does not interfere in any way with other data stored on a shared disk. All NPL disk operations are contained within the specified native operating system files or raw devices (using the DET).

7.3.3 Diskimage Addressing

As stated above in Section 7.2.1, every diskimage must be identified by a unique 3-character device address. This device address in turn is equated to a native operating system file or "raw" device by use of the DET (refer to Section 7.2.2).

Valid diskimage addresses are defined as:

> Dxx - where x represents the ASCII equivalent of a single hexadecimal character (the numbers 0-9 or letters A-F), and D represents a DISK class device.

Some typical diskimage addresses are:

D10
D21
D35
D42

In addition, NPL supports use of the following addresses for compatibility with Wang 2200 applications.

Bx0
3x0

where x represents the ASCII equivalent of a single hexadecimal character (the numbers 0-9 or letters A-F).

NOTE: Use of the "B" or "3" series addresses introduces the possibility of conflict with the "D" series addresses because address Bx0 is the equivalent of address Dx0, and address 3x0 is the equivalent of address Dx1. Use the "D" series of addresses for new development.

7.3.4 Native Operating System Files as Diskimage Files

The following characteristics apply to the use of native operating system files as NPL diskimages ("diskimage files"):

- To the native operating system, the diskimage is a normal file. It can be located anywhere on any physical device within the native operating system file structure. It can be of any name that conforms to the native operating system file naming conventions. It can be accessed by any native operating system utilities, including native backup/restore utilities.
- The internal organization of a standard diskimage file is 100% compatible with Wang Basic-2 disk platters. Refer to Section 7.3.6 below for further details on the internal organization of diskimage files. Refer to Section 7.3.10 for details on compatibility when using non-standard (extended) diskimage files.
- Diskimage file size is dynamic. That is, diskimage files may be created at any size (up to the maximum size), and then expanded or contracted under program control.

NOTE: On some operating systems, file expansion and contraction is not a well supported feature, or may have some limitations which apply (refer to the appropriate NPL Supplement for details).

Creating Diskimage Files

Most disk I/O statements require that the diskimage file exist in advance of a statements execution. A diskimage file can be created by the special NPL statement SCRATCH DISK (for details refer to the Statements Guide, SCRATCH DISK). The SCRATCH DISK statement allows the programmer to specify the number of index sectors and total size for a specified diskimage.

NOTE: The device address must be established in the DET table before a SCRATCH DISK statement is used.

For example:

```
10 $DEVICE (/D21) = "PLATTER3.BS2"
20 SCRATCH DISK T/D21, LS=10, END=99
```

creates a file called PLATTER3.BS2 within the native operating system file system. The size of the file is 100 sectors (at 256 bytes per sector: 25,600 bytes). The size is determined by the END parameter, which indicates the number of sectors to be allocated starting from zero (END=99 therefore becomes 100 total sectors). The LS parameter establishes 10 INDEX sectors (refer to Section 7.3.6 below).

When the SCRATCH DISK statement is executed, the physical space on the native operating system disk is allocated at that time. If there is insufficient physical space available to accommodate the specified size of the diskimage file, an I98 - Illegal Sector Address error is generated and only a portion of the diskimage file is created.

WARNING--If the diskimage file already exists, SCRATCH DISK deletes the old file and create a new file.

Changing the Size of a Diskimage File

Standard diskimage files may be expanded, up to the maximum size, by use of the MOVE END statement.

For example, using the diskimage file created in the example above:

```
10 MOVE END T/D21,=999
```

expands the diskimage file PLATTER3.BS2 from 100 sectors (as created by SCRATCH DISK) to 1000 sectors (256,000 bytes).

NOTE: Any existing data on the diskimage is not affected by the MOVE END statement.

When the MOVE END statement is used to increase the size of a diskimage file, physical space on the native operating system disk is allocated at that time. If there is insufficient physical space available to accommodate the increased size of the diskimage file, an I98 - Illegal Sector Address error is generated and the diskimage file may be only partially expanded.

MOVE END can also be used to decrease the size of a diskimage file. However MOVE END cannot decrease the size of a diskimage file below the number of sectors actually used to store cataloged data (this number is the CURRENT END--the topic of cataloged data is discussed in detail in Section 7.3.6 below).

The use of MOVE END to decrease the size of a diskimage file is not supported on all hardware versions of NPL. Refer to the appropriate NPL Supplement for details.

There are several statements which may implicitly expand the size of a diskimage file. Any statement which writes to a sector beyond the physical end of a diskimage file causes the diskimage file to be expanded to the size required to accommodate the write, assuming physical space is available.

HINT: Any new space added to the diskimage file would be treated as NON-CATALOGED space (refer to Section 7.3.6 below for details on the diskimage CATALOG). Therefore, do not use this technique. Statements which may have this effect are:

DATA SAVE BA
DATA SAVE BM
DATA SAVE DA
COPY

Deleting Diskimage Files

Since a diskimage file is a native operating system file, it may be deleted by direct use of the native operating system file delete function.

Diskimage files may also be deleted under NPL program control using the $FORMAT DISK statement.

Wang 2200 Compatibility Notes

Diskimage files as described above are fully equivalent to the Wang 2200 disk platter. Statements for creating the catalog and index, changing the size of the catalog, and removing the entire catalog are fully supported. The concepts described above ensure nearly 100% compatibility with Wang Basic-2 with the following exception:

Incompatibilities arise in the handling of "non-cataloged" space on a diskimage. Although supported by NPL, the use of non-cataloged space could have negative implications for an NPL application. This is due to the dynamic size of diskimage files. On the Wang 2200, a disk platter is always a fixed size, and a program can assume that space for non-cataloged operations is available regardless of the stated catalog size. This is no longer true with diskimage files. The size of a diskimage file is the CATALOG size (unless extended by direct writes as stated above). Therefore, an attempt to read data beyond the catalog end may fail. Also, attempts to write data beyond the catalog end may fail if there is insufficient physical space to extend the diskimage file.

HINT: Do not use non-cataloged operations in NPL.

7.3.5 Native Operating System "RAW" Devices as Diskimages

In addition to defining NPL diskimages as native operating system "diskimage files", NPL allows direct accessing of native operating system "raw" physical devices.

NOTE: This method is extremely hardware dependent. Refer to the appropriate NPL Supplement for details on the availability and support of access to "raw" devices.

Purpose of Using "Raw" Devices

The primary reason for support of "raw" devices is to allow for transfer of data and programs from one machine to another where the native operating system's file formats are incompatible. A special subset of the "raw" device are Wang 2200 compatible diskettes. The Wang 2200 compatible diskettes provide a method of transferring programs and data between a Wang 2200 (with a 5-1/4" diskette drive) and machines using NPL (where Wang 2200 compatible diskettes are supported). Refer to the appropriate NPL Supplement and Appendix G of the Programmer's Guide for details on the support of "raw" diskettes.

Characteristics of "Raw" Devices

A "raw" device is a physical disk which does not contain a native file system. Typically, these are diskette devices. When a device is accessed as a "raw" device, NPL uses absolute addressing on the device as opposed to addressing the device as a native operating system file. The NPL catalog format is placed on the device starting at sector zero.

Use of "raw" devices for diskimages has the following implications:

- "Raw" devices cannot usually be accessed directly by native operating system utilities or programs.
- The size of the diskimage is limited by the physical size of the "raw" device, usually much less than 16MB. Refer to Chapter 5 of the appropriate NPL Supplement and Appendix G of the Programmer's Guide for details on supported diskette devices.
- The $FORMAT DISK statement actually invokes the native operating system format routines to physically format the specified device. The physical format may vary on different machines.

NOTE: $FORMAT DISK may not be supported for all "raw" diskette types on all operating systems. Refer to the appropriate NPL Supplement for details.

- The SCRATCH DISK statement does not "create" the "raw" device. Rather SCRATCH DISK constructs the internal structure of the NPL diskimage on the "raw" device. In addition, MOVE END does not allocate or de-allocate space. It simply modifies the existing internal structure to the new parameters. Refer to Section 7.3.6 below for details on the internal structure of NPL diskimages.

- All other NPL disk I/O statements directed to diskimages defined as "raw" devices function identically to statements directed to a diskimage defined as a native operating system file, except where otherwise noted in the appropriate NPL Supplement.

7.3.6 Internal Structure of Diskimage Files

Every NPL diskimage may contain up to 4079 logical files. These may be either program or data files. Diskimages have a specialized internal format which is used to maintain information about these files within the disk. This format consists of an INDEX area, a CATALOG area, and an optional NON-CATALOGED area.

Catalog Index and Catalog Area

The Index contains the name and sector location of every cataloged program and data file on the diskimage. The Catalog Area is where each cataloged program file and data file is physically stored. The size of the Catalog Index and Catalog Area is established through the use of the SCRATCH DISK statement with the following format:

General Form:

```
SCRATCH DISK [’]T [file-number,  ][LS=value1,] END=value2
                  [disk-address, ]
                  [<address-var>,]
```

Where:

```
’      =    specifies the alternate hashing method is
            to be used.

value1 =    a numeric- expression which represents
            the required size of the Catalog Index
            whose value is from 1 to 255.  The de-
            fault value if not specified is 24.

value2 =    a numeric- expression which represents
            the highest sector address in the Catalog
            area.  The expression must be less than
            or equal to highest sector address avail-
            able on disk.
```

The Index is always stored at the beginning of each diskimage, starting at sector zero. Each sector of the Index contains 16 file entry slots. The first sector of the Index uses the first file entry to store the hash method, index size, current end of the Catalog Area, and the upper limit (END) of the Catalog Area. All other file entry slots are used to maintain information about individual files. Files listed in the Index are referred to as CATA-LOGED files.

The first file entry in sector 0 of the Index is structured as follows:

Byte,	Description
Byte 1	00 = Old hashing method. 01 = New hashing method.
Byte 2	The number of index sectors from 1 to 255. HEX(00) indicates disk is not scratched.
Bytes 3,4	Next available sector for storing program or data files (byte 3 is high,, byte 4 is low). This is the CURRENT END parameter and is used whenever a new cataloged file is created on the disk. It also can be used to determine how much of the catalog area is actually used for the storage of data files. This parameter is established initially by SCRATCH DISK and is incremented whenever a new file is added to the catalog.
Bytes 5,6	First sector beyond the cataloged area (byte 5 is high,, byte 6 is low). This is the END CATALOG parameter and is used in conjunction with the CURRENT END to determine how much space is available within the catalog area for storing additional files (END CATALOG - CURRENT END = space available). This parameter is established initially by SCRATCH DISK and may be modified by MOVE END.
Bytes 7,8	Used for extended diskimages (refer to Section 7.3.10 for details on their use).
Bytes 9-16	Unused and should contain all HEX(00).

NOTE: Old hashing/new hashing refers to the method by which file entries are stored within the Index. A hashing method is always used to determine the index sector to use for storage of any given file entry. The same hashing method is used to locate the file when requested. If any given Index sector is full when a file is added, the next available sequential Index sector is used. The "old" versus "new" designation refers to the hashing algorithm used. In cases where many files are stored within a disk, the "new" hash may be somewhat more efficient. The hash type is specified as part of the SCRATCH DISK statement and this byte is set at the time that the SCRATCH DISK is executed. For more details refer to the Statements Guide, SCRATCH DISK).

All other file entries in the catalog are structured as follows:

Byte	Description
Byte 1	00 = No entry. 10 = File entry. 11 = Scratched file entry. 21 = Entry no longer used.
Byte 2	File type: 00 = Data. 80 = Program.
Bytes 3,4	First file sector (byte 3 is high
Bytes 5,6	Last file sector (byte 5 is high
Bytes 7,8	Used for extended diskimages (refer to Section 7.3.10 for details).
Bytes 9-16	Contains an eight byte file name.

Listing the Index

The Index may be examined by the LIST DCT command. This command lists the number of Index Sectors, Catalog Current End, and End Catalog parameters. LIST DCT may also be used to list the status and location of selected cataloged files. In addition to LISTDCT, the Index may also be examined the LIMITS INDEX and READ DCT statements.

LIMITS INDEX returns the number of index sectors, current end, end catalog, and hash type for the specified diskimage to numeric variables.

READ DCT returns filenames matching a specified a wildcard pattern to a specified alpha-variable one at a time.

Refer to the Statements Guide, LISTDCT, LIMITS INDEX and READ DCT for details.

Non-Cataloged Area

Any space on a diskimage beyond the End Catalog is considered non-cataloged space. This space cannot be used to store catalog files. Only direct access to this space is supported. Non-cataloged space on a diskimage file can only be created by use of a direct access statement to write to a sector number beyond the End Catalog (refer to Section 7.3.4 above).

HINT: This technique is supported for compatibility with Wang 2200 applications which use this technique, but there are serious restrictions to its functionality. Therefore, do not use this technique for new development in NPL. Refer to Section 7.3.4 for further details on the restrictions in using non-cataloged space under NPL.

7.3.7 Cataloged Files

A cataloged file is a file that is maintained within the catalog area of a diskimage with an associated file entry in the Catalog Index. Catalog files have the following characteristics:

- Files may be either program files or data files. Characteristics that depend on the file type are discussed in the corresponding sections below. Characteristics described in this section refer to all files.

- File names may consist of any NPL characters (HEX(00-FF)) up to a maximum of eight characters. Imbedded and trailing spaces are considered part of the file name. File names are case sensitive.

HINT: Filenames should be constructed from characters enterable from the keyboard and displayable on the terminal screen. Duplicate filenames within the same diskimage are not allowed.

- File size is restricted only by the size limits of the diskimage (16MB for standard diskimages). Files may not span multiple diskimages.

- File size is fixed. Once a file has been created, it cannot be expanded or contracted by any NPL statement.

NOTE: Some NPL utility programs (not supplied by Niakwa) which directly manipulate the Catalog Index and Area do have the ability to expand or contract files.

- Physical space for a file is allocated at the time the file is created. If there is insufficient space for the specified file size within the catalog area, an error results and the file is not created. Files may be flagged as no longer valid by use of the SCRATCH statement, but this does not automatically release the space allocated for use by other files. However, there are some methods of reusing the space allocated for SCRATCHed files (refer to Creating Files, below). Space allocated for files not reused can only be freed by use of the MOVE statement to MOVE all catalog files to a new diskimage, or by use of a utility program.

NOTE: The Niakwa General Backup and Restore utilities can be used to free space allocated to scratched files. Refer to Chapter 13 for details.

- The last sector of every file is a trailer sector which contains status information about the file. This sector is reserved for the maintenance of system information and should not be modified directly by an NPL program. This sector contains:

Byte 1	Control Byte - HEX(A0)
Bytes 2-3	Number of sectors used,, including the trailer (byte 2 is high,, byte 3 is low).
Bytes 4-22	Date/Time stamp - in the format yyyy/mm/dd hh:mm:ss.
Bytes 23-25	Used for extended diskimages. Refer to Section 7.3.10 for details.
Bytes 26	File status (as contained in byte 1 of the index entry for the file).
Bytes 27	File type (as contained in byte 2 of the index entry for the file).
Bytes 28-35	File name (as contained in bytes 9-16 of the index entry for the file).
Remainder	All other bytes are undefined and may contain random data.

File status, type, and name information are maintained by all NPL statements which can create files or modify this information. In addition, the compiler (B2C) creates this information when creating programs. This information is not used by any NPL statement except for LISTDCT which checks it for consistency with the Index entry and displays a "?" if it does not match. Rather, this information is maintained strictly as a convenience for various applications or utilities which may wish to examine it.

- Current information about the status of individual files may be obtained by use of the LIMITS statement. LIMITS returns the starting sector, ending sector, sectors used, and file status (program or data, active or scratched) for the specified file. For further details refer to the Statements Guide, LIMITS.

Wang 2200 Compatibility Notes

The above "standard" diskimage characteristics are completely compatible with the Wang 2200 method of storing files with the following exceptions:

Date/Time Stamp
File Status
File Type
File Name information

These features are specific to NPL. However, since only the trailer sector is used for this, there should be no impact on the portability of data files between a Wang 2200 and an NPL environment.

In the event it is desired to suppress this information, Byte 40 of $OPTIONS may be used to specify whether or not bytes 4-35 of the file trailer sector are used by NPL to store the date/time stamp, file status, file type and filename. The default value of HEX(00) indicates that NPL should store this information in the file trailer sector (as is done by previous revisions). A value of HEX(01) indicates that the NPL should not store this information. Bytes 1-3 of the file trailer are updated by NPL regardless of the value of byte 40 of $OPTIONS. If extended diskimages are in use and the size of a file exceeds 64K, then bytes 4-35 of the trailer sector are updated by NPL regardless of the value of byte 40 of $OPTIONS. Refer to the Statements Guide, $OPTIONS for details. Refer to Section 7.3.10 for details on extended diskimages.

Creating Files

Program files may be created by use of the SAVE DC statement. The SAVE DC statement saves the program text currently in memory to a diskimage file of the specified name. If a new file is to be created, the size and name of the file may be specified. If space from an existing file is to be used, the old and new file names may be specified. For further details refer to the Statements Guide, SAVE DC. Also Refer to Section 5.6, Saving Programs, for additional discussion on saving program text to disk.

Data files are created by the DATA SAVE DC OPEN statement. If a new file is being created, the name and size of the file must be specified. If a scratched file is to be reused, the old name and new name must be specified. For further details, refer to the Statements Guide, DATA SAVE DC OPEN.

Maintaining Files

File status (scratched or not scratched), file type (program or data), and file name are dynamic values which may be modified by certain NPL statements:

SCRATCH	Sets the file status for specified files to "scratched".
UNSCRATCH	Sets the file status for specified files to "not scratched".

SET DATA	Sets the file type for specified files to "data".
SET PROGRAM	Sets the file type for specified files to "program".
RENAME	Changes the name of a specified file.

In addition, certain forms of MOVE, SAVE, and RESAVE may change file status or file name.

Accessing Files

Cataloged program files can be loaded into memory by use of the LOAD DC or LOAD RUN command. For further details refer to the Statements Guide, LOAD DC and LOAD RUN. Also refer to Section 5.3 for further information on LOADing programs.

Cataloged data files may be accessed by either the catalog method or by direct access. Refer to Sections 7.3.8 and 7.3.9 for details.

Internal Format of Program Files

The internal format of program files is detailed in the discussion of the $SOURCE and $OBJECT statements. For further details refer to the Statements Guide, $SOURCE and $OBJECT.

NOTE: With the introduction of Long Identifier Names in NPL Release IV, some changes may need to be made to programs using the $SOURCE and $OBJECT statements. Niakwa has simplified this process by providing a two libraries, "SOURCEIO" and "OBJECTIO", to assist developers in adapting existing routines to handle Long Identifier Names. Refer to Chapter 3 of the Statements Guide for details.

Internal Format of Data Files

The internal format of data files, except for the trailer sector as discussed above, is under control of the programmer. However, NPL does provide a method of accessing data files in a structured fashion with the Catalog Method (discussed in Section 7.3.8). Alternatively, the developer may choose to use direct access methods where the internal structure of the data file is totally under the control of the programmer. Refer to Section 7.3.9 for details.

7.3.8 Catalog Access Methods

The above discussion has introduced the concept of the catalog access method. Techniques for creating and maintaining logical files within a diskimage Catalog were discussed in Section 7.3.7 above. In this section, the catalog access method is further explored.

The catalog access method provides a series of statements which can be used to access data files in a structured fashion.

The advantages of catalog access to data files are:

- Data can be referenced by name as opposed to sector location.
- Data is stored in a logical format where each record consists of one or more physical sectors.
- There is a capability to skip forward and backward over data records within the file.

The disadvantages are:

- Direct access techniques should not be used on conjunction with catalog access techniques.
- Only sequential access to logical records is supported.
- Applications which require random access must use direct access techniques and impose their own random access method.
- Automatic blocking of logical records within a sector is not supported. If blocking is desired,, the application program must control it.

HINT: Use of catalog access methods is discouraged for new development. The Niakwa Data Manager routines provide a more efficient and reliable alternative to NPL's internal catalog access methods and should be considered for all new development.

Summary of Statements Used for Catalog Access to Data Files

A series of NPL statements are provided to support catalog access to data files. These statements are discussed in greater detail in the Statements Guide.

Creating Data Files

DATASAVE DC OPEN	Used to reserve space within the Catalog Area for any cataloged files, with appropriate system information stored in the Catalog Index.

Opening Existing Data Files

DATALOAD DC OPEN	Used to open any data file which has previously been cataloged on the disk.

Reading and Writing Data

DATALOAD DC	Used to read logical data records from a cataloged diskimage file, and assign to variables the values read from each read operation.
DATASAVE DC	Used to write to diskimage one logical data record containing specific data.

Setting the Current Record Pointer

DSKIP	Used to skip over a specific number of logical records or sectors in a cataloged diskimage file.
DBACKSPACE	Used to backspace over a specific number of logical records or sectors in a cataloged diskimage file.

Setting the End-of-File Mark

DATASAVE DC END	Used for writing a data trailer record within a cataloged diskimage file, indicating the logical end of the file.

Testing for End-of-File

IF END THEN	Used to test for the logical end of a cataloged data file, and branch to a specified line number within the program.

Closing a Data File

DATASAVE DC CLOSE Used to close a specific cataloged data file or all data files on diskimage currently open.

The Logical Record

As stated above, catalog access methods allow data within a file to be accessed by logical records. A logical record consists of a group of data values. Any number of data values may be included in a logical record. Values may consist of both numeric and alpha items. Values may be scalar variables, constants, literals, or arrays when written.

NOTE: Each element of an array is considered as a separate value. The value list for reading data may only contain scalar or array variables. No single value may be greater than 124 bytes in length (refer to Diskimage Utilization below).

A value list is specified for each write (DATA SAVE DC) or read (DATA LOAD DC) of the logical record. Value lists may be different for different records within the file. When values are read by a DATA LOAD DC statement, the value list MUST correspond to the order of data values on the diskimage for that logical record. That is, where a numeric variable is specified, numeric data must be present, and when an alpha-variable is specified, alpha data must be present.

Diskimage Utilization for Logical Records

For efficient use of diskimage space, some understanding of the internal organization of logical records is required. Each logical record occupies one or more physical sectors on the disk. Each sector contains as many data values as can be contained in the sector, plus control information.

NOTE: Data values do not span sectors.

The control information consists of:

- Three bytes per sector of sector control information. The first two bytes of every sector plus the byte following the last data value are reserved for system use.
- Each data value is preceded by one byte of control information. The length of data values when stored on diskimage is (not including the value control byte):
- Alpha values: the length of the value (trailing spaces included).
- Numeric values: eight bytes.

The maximum size for any data value is 124 bytes.

Careful planning of logical records is required to ensure efficient use of the disk.

For example, an array defined as:

```
DIM X$(8)64
```

requires 3 sectors when saved to disk using a DATA SAVE DC statement. However, an array defined as:

```
DIM X$(8)62
```

requires only 2 sectors on disk, but contains only sixteen fewer bytes of data.

In many cases, where the data list consists of multiple arrays or variables with different dimensions, simply changing the order of the value list may result in saving one or more sectors per record.

More efficient use of the diskimage may also be obtained by packing multiple data elements into a single variable or array before saving. This is particularly true for numeric values since a numeric value can often be packed into a much smaller alpha string. For further details refer to the Statements Guide, $PACK and $UNPACK.

Logical End-of-File

A logical end-of-file sector can be saved by the DATA SAVE DC END statement. Use of DATA SAVE DC END is not required, but is a good programming technique to use, so that subsequent sequential reads of the file can test for the logical end of file after every DATA LOAD DC, by use of the IF END statement.

NOTE: If a DATA SAVE DC statement is written to the sector containing the logical end-of-file flag, the end-of-file flag is overwritten. However, a new end-of-file flag can be set by a subsequent DATA SAVE DC END. The logical end-of-file flag is also useful in applications where data may be added to existing files. If an end-of-file flag is present, a DSKIP END statement may be used to locate the current record pointer to the logical end of file, so that new data does not overwrite existing valid data.

The logical end-of-file sector should not be confused with the trailer sector. The trailer sector denotes the physical end-of-file. The logical end-of-file sector must precede the trailer sector.

Use of the Internal Device Table

The catalog access method uses the diskimage file slots of the Internal Device Table (DT) to maintain information about individual data files being accessed (other aspects of the DT are discussed in Section 7.2 above). The file slots of the DT are used to maintain information about cataloged files currently being processed. Up to 256 cataloged files can be processed at one time. These files can be on any defined diskimage. Multiple files on the same diskimage may be open for processing.

Before using the DT for cataloged file processing, two steps must be performed:

1. A diskimage device address must be assigned to the diskimage file slot.

   For example:

   ```
   SELECT #1/D35
   ```

   assigns address D35 to file slot #1.

NOTE: Any address assigned to a file slot must be properly configured in the Device Equivalence Table.

2. The specific data file must be opened by a DATASAVE DC OPEN (for a new file) or DATALOAD DC OPEN (for an existing file) statement.

   For example:

   ```
   DATALOAD DC OPEN T#1,"ARMASTER"
   ```

   opens the existing file ARMASTER and establishes the diskimage file slot entry.

Assuming that the file ARMASTER occupies sectors 1034 to 2048 on diskimage D35, the results of the above two steps would generate the following entry in the DT:

File, Address, File-name, Start, Current, End
1, T/D35, "ARMASTER", 1034, 1034, 2048

File (# 1) is the number of the file slot used.

Address (T/D35) is the device address assigned to this slot by the SELECT statement.

File name ("ARMASTER") is the name of the file specified in the DATALOAD DC OPEN statement.

Start (1034) is the first sector number of file ARMASTER.

Current (1034) is a pointer to the sector number to be accessed by the next operation. This value is maintained automatically by the catalog statements described above. Current is initialized to be the same as START when a file is first opened by DATA SAVE DC OPEN or DATA LOAD DC OPEN. Subsequent operations update the CURRENT pointer based on the results of the operation.

For example, if a DATA SAVE DC statement is executed with a value list that causes a logical record three sectors long to be written, CURRENT is incremented by three so that the next operation starts at the sector number following the last sector used in the DATA SAVE DC operation.

End (2048) is the physical end of file ARMASTER. Execution of any statements which would cause access to sector numbers equal to or greater than END results in an error.

7.3.9 Direct Access

In addition to the catalog access methods described above, NPL supports a series of instructions which provide direct low level access to specified diskimage sectors. Direct access techniques are often used in applications where the catalog method is insufficient for the requirements of the application. Applications which make use of direct access techniques are likely to incorporate some form of internal mechanism for accessing logical records within an otherwise unstructured file.

Summary of Statements Used for Direct Access to Data Files

A series of NPL statements are provided to support direct access to data files. These statements are discussed in greater detail in the Statements Guide.

DATASAVE DA	Used to save logical data records on the diskimage at a specified sector.
DATALOAD DA	Used to read one or more logical records from a data file on disk, beginning at a specified absolute sector address.
DATASAVE BA	Used to save one sector of unformatted data on the diskimage at a specified sector location.
DATALOAD BA	Used to load one sector of unformatted data from the disk, from a specified sector location.
DATASAVE BM	Used to save one or more sectors of unformatted data on the diskimage at a specified sector location.
DATALOAD BM	Used to load one or more sectors of unformatted data from the diskimage, from a specified sector location.
COPY	Used to copy a range of sectors from one diskimage to another.

NOTE: These statements are often used with cataloged data files. However, they are not typically used in conjunction with the catalog access technique described in Section 7.3.8 above (except for the statement DATA SAVE DC OPEN, which is required to establish an Index entry and Catalog space for the file).

For example:

```
10 LIMITS T/D32,"FILE1",A,B,C
20 FOR X=A TO B: DATA LOAD BAT/D32,(X)I$(): NEXT X
```

This program reads all sectors, one at a time, from the cataloged file named FILE1. The LIMITS statement is used to access the Catalog Index and find the starting (A) and ending (B) sector locations. Then, as long as access is directed to this range of sectors, the program can be sure that it is working within the correct logical file.

NOTE: This example assumes that the file FILE1 has previously been established as a cataloged file by a DATA SAVE DC OPEN statement.

7.3.10 Extended Diskimages

The following details the use of NPL extended diskimages.

Implementation

In a number of places, the original Wang Basic-2 language assumed that valid sector numbers could always be stored as a 2-byte binary number. As the capacity of available disk units increased past this limit, this limitation required that such larger units be partitioned into "logical" platters whose size did not exceed the 16MB limit.

Access to diskimages greater than 16MB is accomplished by supporting the use of three-byte sector addressing. The functional effect of many NPL disk I/O statements is affected by this addressing scheme, and applications which are using this method for the first time are most likely to require modifications.

Because this technique can cause difficulty for existing application software, the ability to create and reference diskimage files greater than 16MB requires that the application specifically request this capability on a diskimage-by-diskimage basis. This is accomplished by use of the "EXT=Y" clause in the $DEVICE statement. For example:

```
$DEVICE(/D11)="PLATTER1.BS2"
```

would allow one to address a maximum of 16MB, however;

```
$DEVICE(/D11)="PLATTER1.BS2 EXT=Y"
```

allows access to sectors beyond 16MB.

NOTE: As a safety measure designed to prevent use of extended diskimages by software which may not be suitable for use with them, the RunTime does not allow access to diskimage files larger than 16MB unless this capability is enabled by use of this special clause in the $DEVICE statement for the particular diskimage file.

The default value for the EXT clause is N. Therefore if not specifically requested, diskimage files are treated as non-extended.

WARNING--When changing the size of a diskimage file from < 16MB to > 16MB, it is important that all users on the system exit the RunTime and reenter. Otherwise it is possible to have one terminal accessing a file as extended while another terminal has opened the file as non-extended. This results in difficulties because of the different access methods used.

Maximum Size

Extended diskimages allow the addressing of sectors up to 16777214 (i.e., (2^24-2) sectors of 256 bytes; a total of 4096MB). This is the largest number X for which X+1 can be stored as an unsigned binary number in 3 bytes. A smaller limitation may be imposed by the operating system due to internal constraints or available disk storage types.

Compatibility

Use of diskimages with more than 65534 sectors is not downward compatible with revisions of the RunTime prior to 2.01 (or with the Wang 2200), and may not be compatible with software which directly manipulates the data in the index area of diskimage. Refer to the discussion on Internal Structure of diskimages below for details.

Older disk images are upwardly compatible provided that the bytes previously unused in the index area are zero, which should be the case unless software has been altering the index and using these bytes for some other reason.

NOTE: **The new diskimages are directly upwardly compatible with the old diskimages, and also downward compatible provided that the size does not exceed 65534 sectors.**

The NPL compiler does not require the EXT= Y clause when referencing an extended diskimage. B2C automatically determines the correct size of the diskimage and properly accesses extended diskimages.

Implications for Application Software

The use of three-byte disk addresses has many implications for existing application software. In this section, the most typical areas that the developer should consider before implementing this feature are identified. We strongly urge that all elements of an application be thoroughly tested with expanded diskimages prior to end-user installation. In cases where third-party software is to be used, it is essential that the software author verify that the product is suitable for use with extended diskimages.

Three-Byte Addresses

By introducing diskimages which exceed the 16MB limitation, a problem arises since applications may assume that valid sector numbers and sizes never exceed the older 65534 value, and consequently store the value in tables or records as a 2-byte binary number. No general solution to the problem exists, and programmers who wish to use larger disk images must ensure that all fields which store sector numbers (or relative sector numbers), file sizes or record counts are expanded to accommodate the larger values.

A particular case of this problem occurs in the index area of NPL diskimages, where the start and end sectors of each file are stored as 2-byte binary numbers. Also, the "Current End" and "End of Catalog" values are stored in this way, as is the "used" value in the last sector of each file on the diskimage. Developers wishing to use the larger disk images who use routines which directly inspect or modify the index area must modify these routines. For information on how these fields have been expanded, refer to the changes to the "Direct Access to the Index" section below.

Direct Access to the Index

Many NPL programs attempt to perform direct access to the index area of the catalog. All such programs must be modified to accommodate the use of bytes 7 & 8 for storage of the high-order byte of sector addresses if these programs are to be used with extended diskimages. Programs which contain logic of this nature are typically utility programs designed to perform backup or diskimage reorganization, but are not necessarily limited to these functions.

WARNING--Attempts to use existing versions of such programs on extended diskimages result in damage to the diskimage file!

The Niakwa utilities supplied with the Development Packages for Revisions 2.01 or greater have been modified to address extended diskimages properly. If larger diskimages are to be used, these versions must be used or damage to the diskimages may result.

Use of Bytes 7 & 8 for Other Purposes

Some applications use bytes 7 & 8 of the index entries for their own purposes. Attempting to use a diskimage file where bytes 7 & 8 are used for other purposes as an extended diskimage file results in the RunTime program attempting to use these values as the high-order byte of sector addresses. This likely results in serious damage to data contained in the diskimage.

Internal Structure of Diskimages

Sector 0	Standard Diskimages	**Extended Diskimages**
Byte 1	Hashing method	Hashing method
Byte 2	# of index sectors	# of index sectors
Byte 3,, 4	# sectors used	Low order # of sectors used
Byte 5,, 6	# sectors cataloged	Low order # of sectors cataloged
Byte 7	Not used	High order # of sectors used
Byte 8	Not used	High order # of sectors cataloged
Byte 9-16	Not used	Not used

Other Catalog Elements	Standard Diskimages	**Extended Diskimages**
Byte 1	File status	File status
Byte 2	File type	File type
Byte 3, 4	File start sector	Low order file start sector
Byte 5, 6	File end sector	Low order file end sector
Byte 7	Not used	High order file start sector
Byte 8	Not used	High order file end sector
Byte 9-16	file name	file name

File trailer	Standard Diskimages	**Extended Diskimages**
Byte 1	End of file mark	End of file mark
Byte 2,, 3	# of sectors used	# of sectors used if < 65535. Otherwise contains 0.
Byte 4-22	Date stamp	Date stamp
Byte 23-25	Not used	Number of used sectors if > 65535. NOTE: bytes 2,,3 must be zero.
Bytes 26	File status	File status
Bytes 27	File type	File type
Bytes 28-35	File name	File name
Byte 36-256	Not used	Not used

An additional case which may affect application programmers is the form of DATA LOAD and DATA SAVE, which use alpha-variables as the sector number to access or as the return-variable. In the case of the sector address, only the binary value of the first two bytes is used to determine the address, whether or not the diskimage file accessed is extended. As for the return-variable, to avoid possible misinterpretation of this result, if such a statement is used and the sector number exceeds 65535, an error P51 (Variable or value too short) occurs. Developers wishing to use the larger diskimages who use this form of DATA LOAD or DATA SAVE are advised to use numeric-expressions for sector-address and return-variable parameters instead. The numeric equivalent of a sector-address may be obtained through use of the VAL(,3) statement, and the binary equivalent of the return-variable can then be obtained using the BIN(,3) function. Alternatively, conversion between alpha-variables containing sector addresses and numeric variables can be accomplished by use of the extended $PACK/$UNPACK statements. Refer to the Statements Guide, $PACK, $UNPACK.

NPL Statements Affected

The effect of extended diskimage files on various NPL disk I/O statements follows. This discussion is broken down into two parts: the effect on statements directed towards an extended diskimage (EXT=Y), and the effect on statements directed towards a non-extended diskimage.

For Extended Diskimages:

COPY	Accepts larger values for start, end, and destination parameters.
DATA LOAD BA	Accepts larger values for the sector location if a numeric expression is used. Returns larger values in the specified return-variable if a numeric-variable is used.

NOTE: If an alpha-variable is used for the sector location or return-variable specification, only the first two bytes are used. Refer to the discussion of internal diskimage structure in this section for further details on the use of alpha-variables to store sector addresses with extended diskimage files.

DATA LOAD DC	Operation of this statement is transparent with extended diskimage files. Internal operation of the statement handles sector numbers larger than 65535 with no programming considerations.

DATA LOAD DC OPEN	Initializes the Internal Device Table with correct sector locations for start and end of file. Bytes 7 & 8 of index entry for specified file are used.
DATA LOAD BM	Same as DATA LOAD BA.
DATA LOAD DA	Same as DATA LOAD BA.
DATA SAVE BA	Same as DATA LOAD BA.
DATA SAVE BM	Same as DATA LOAD BA.
DATA SAVE DA	Same as DATA LOAD BA.
DATA SAVE DC	Same as DATA LOAD DC.
DATA SAVE DC END	Same as DATA LOAD DC.
DATA SAVE DC OPEN	Allows larger value for the space parameter. Otherwise, like DATA LOAD DC OPEN.
DBACKSPACE	Accepts larger values for number of records or sectors.
DSKIP	Same as DBACKSPACE.
LIMITS	Returns appropriate values. Bytes 7 & 8 of index entry for specified file are used.
LISTDT	Displays expanded sector number fields.
LISTDC	Displays expanded sector number fields.
MOVE (form 1)	Creates an extended diskimage only if the output diskimage file is defined as extended. Generates a P48 - Illegal Device Specification if output diskimage is not defined as extended and input diskimage is larger than 16MB.
MOVE (form 2)	Allows larger value for space parameter. Uses bytes 7 & 8 of index entry for specified file on both input and output diskimages.

MOVE END	Allows larger value for END parameter. Bytes 7 & 8 of first entry of sector zero used.
SAVE	Allows larger value for space parameter. Uses bytes 7 & 8 of file entry and the first entry of sector zero.
SAVE DA	Same as DATA LOAD BA.
SCRATCH DISK	Creates diskimage files larger than 16MB. Allows larger value for END parameter. Bytes 7 & 8 of first entry of sector zero used.
VERIFY	Allows larger value for START and END parameters. Larger value may be returned in specified numeric-receiver.

For Non-Extended Diskimages:

With minor exceptions, operation of all NPL disk I/O statements directed against non-extended diskimages is unchanged from prior releases. The exceptions are:

COPY	No range check is performed on the destination parameter. If the destination parameter is greater than 16MB, NPL attempts to perform the write operation. This has varying results on different operating systems. This should present no problem for applications unless they have problems which would cause an invalid destination parameter to be used.
DATA SAVE BA	No range check is performed on the specified sector address. Results of this are as discussed in COPY above.
DATA SAVE BA	Same as DATA SAVE BA.
DATA SAVE DA	Same as DATA SAVE BA.
SAVE DA	Same as DATA SAVE BA.

NOTE: If extended diskimages are not specified, NPL statements do not attempt to use bytes 7 & 8 of the index entries for sector addresses. In addition, specifying an END value above 65535 for SCRATCH DISK or MOVE END results in a P34 - Illegal Value error as in prior versions of the RunTime.

7.3.11 Native Operating System Files

In addition to NPL's internal catalog access methods, NPL provides developers the ability to access almost any external native operating system file. This section discusses two potential methods of accessing native operating system files. The first method describes the handling of ASCII text files using NPL. The second discusses the Niakwa Data Manager (NDM) and it's inherent benefits over the standard NPL catalog access methods.

ASCII File Access

The task of creating, accessing and manipulating ASCII text files is easily accomplished under NPL. NPL has the ability to handle I/O from two type of devices: Disk Class and Print Class devices.

Print Class Devices

The following is an example of creating a delimited text file named PRTTEXT.TXT on an MS-DOS based system using the print class device /215.

```
20  PRINT HEX(03)
30  $DEVICE(/215)="PRTTEXT.DAT"
40  $GIO/215,(HEX(8700)<
    :REM**REWINDS FILE OR REM OUT TO APPEND TO FILE
50  DIM X$256,D$512
60  A$="DELIMITED"<
    :B$="ASCII"<
    :C$="RECORD"
70  D$=HEX(22) & A$ & HEX(222C22) & B$ & HEX(222C22) & C$ & HEX(22)
80  SELECT PRINT 215
90  FOR I = 1 TO 10
100 PRINT D$
110 NEXT I
120 SELECT PRINT 005
130 $SHELL "TYPE PRTTEXT.DAT"
140 STOP
```

This is probably the most common method of text file creation. Text files created in this fashion contain a carriage return line feed character at the end of each record. These characters represent end of record information when importing text files into other third party database, spreadsheet or word processing products.

The following example is an input routine which will read in the PRTTEXT.DAT file created in the above routine.

```
10  DIM A$80,B$49
20  PRINT HEX(03)
25  LINPUT "ENTER FILE TO BE READ IN",B$
30  $DEVICE(/215)=B$
40  REM ***REWIND THE FILE***<
    :$GIO/215,HEX(8700))
50  REM ***GET SOME BYTES***<
    :$GIO/215,(HEX(C620),G$)A$
60  REM ***GET COUNT OF BYTES READ***<
    :IF G=VAL(STR(G$,9),2)
70  REM ***CHECK END OF FILE***<
    :IF G=0 THEN 100
80  REM ***PRINT BYTES TO THE SCREEN***<
    :$GIO/005,(HEX(A000))A$,G
85  KEYIN P$
90  GOTO 50
100 STOP
```

This routine uses the $GIO (C620) microcommand sequence to read the file in using the print class device /215. Since $GIO does not insert and line handling characters, each record read in is output nicely to the screen with it's HEX(0D0A) characters enforcing carriage return line feeds where appropriate.

Refer to the Statements Guide, $GIO, for details on the full functionality of this statement.

Disk Class Devices

The following is an example of creating a delimited text file named DISKTEXT.DAT on an MS-DOS based system using the disk class device /D40.

```
20  PRINT HEX(03)
30  $DEVICE(/D40)="DISKTEXT.DAT"
40  SCRATCH DISK T/D40,LS=1,END=10<
    :REM***CREATE FILE***
50  DIM X$256,D$512
60  A$="DELIMITED"<
    :B$="ASCII"<
    :C$="RECORD"
70  D$=HEX(22) & A$ & HEX(222C22) & B$ & EX(222C22) & C$ & HEX(22)
80  SELECT DISK/D40
90  FOR I = 1 TO 5
100 DATASAVE BM T((I-1*2))D$
110 NEXT I
120 SELECT PRINT 005
130 $SHELL "TYPE PRTTEXT.DAT"
140 STOP
```

This routine outputs each record using direct sector addressing. This is accomplished using the DATA SAVE BM statement in line 100. Each record is stored in it's own 256 byte sector, unlike the print class example above, which omitted all trailing spaces. In addition, carriage return and line feed characters are omitted from the end of each record.

The following example is an input routine to read in the DISKTEXT.DAT file created in the above program.

```
10  PRINT HEX(03)
20  DIM X$256,X1$256,A$49
30  LINPUT "ENTER NAME OF ASCII TEXT FILE",-A$
40  $DEVICE(/D30)=A$
50  X=0 : E=0
60  X$=" "
70  DATA LOAD BA T/D30,(X)X$<
    :ERROR DO<
    :E=ERR<
    :IF E<> 98 THEN STOP "error"
    :ENDDO
80  PRINT HEX(03)
    :PRINT AT(1,24);"SECTOR";X;"OF FILE";A$
    :PRINT
    :LIST DIM *X$
90  REM X$ CONTAINS NEXT 256 BYTES OF DATA - DO WITH  IT AS YOU PLEASE
100 X=X+1
110 KEYIN L$
120 IF E=0 THEN GOTO 60:rem MORE DATA LEFT
180 REM DONE
190 REM *UNDER VMS DISK CLASS DEVICES CREATE 256 BYTE
         FIXED LENGTH SEQUENTIAL FILES
200 REM *FILES CREATED BY THE VMS TEXT EDITOR, OR NPL
         PRINT CLASS DEVICES ARE CONSIDERED VARIABLE
210 REM *LENGTH SEQUENTIAL FILES AND MAY ONLY BE READ
         INTO NPL VIA  $GIO STATEMENTS
```

The DATA LOAD BA direct sector address statement on line 30 is used to read each sector of the specified file and displays a formatted sector dump of each record in the file, allowing the developer to view the sectors for embedded line feed and carriage return characters.

NOTE: The ERROR statement in line 70 traps for the native end-of-file error. If you intend to access native files under DEC VMS, please note the comments in lines 190 and 220.

Refer to the Statements Guide, DATA LOAD BA and DATA SAVE BA for details on these and other direct sector addressing statements.

The Niakwa Data Manager (NDM)

The Niakwa Data Manager (NDM) is library of external routines, designed to give developers an alternative to storing application data files in NPL diskimages using either NPL's catalog access methods or a variety of internally developed, less efficient ISAM's written directly in NPL.

NDM allows NPL developers to create and maintain application data files in a variety of industry standard, highly efficient, native ISAM products, while maintaining the portability of those data files across various NPL supported platforms.

The immediate benefits of NDM are:

- Increased performance of data access
- Decreased file maintenance (no "File Full" errors).
- Increased product offerings. Files stored in industry standard native ISAM's are accessible to all other third party products that support the native ISAM.

For more information regarding the features of NDM, refer to the NDM Programmer's Guide, or contact Niakwa.

7.4 Screen Handling

This section discusses the methods by which output to the display device screen can be generated by NPL programs.

NOTE: **The actual implementation of many of the features discussed in this section is highly dependent upon the capabilities of the native operating environment and the display device used. Refer to the appropriate NPL Supplement for details on the display devices supported ad the specific compatibility issues involved for different implementations of NPL.**

7.4.1 General Features

As with other I/O operations, the NPL method of screen output is designed to be compatible across all machines on which NPL operates, as well as to maintain compatibility with the Wang 2200. The following characteristics apply to screen output:

- NPL can address 80 columns (numbered 0 through 79) and 24 rows (numbered 0 through 23) on the terminal screen.

NOTE: On devices which support 132 column mode, NPL allows programs to make use of this feature. Refer to Section 7.4.12 for details.

- All output to the screen is in character format as opposed to block format. That is, any character can be printed at any location on the screen without affecting other characters on the screen (except for the possibility of scrolling, as described below).

- If row and column are not explicitly specified, output is displayed at the current cursor location. The cursor location is always set to the screen position following the last character printed, unless explicitly set elsewhere.

- If any output would cause printing past the last available column (80 or 132), the output "wraps" to the next line.

NOTE: Use of the 205 address, or an output width greater than last available column, causes wrapping to the same line (refer to Section 7.4.2 below for details).

- If any output would cause printing past line 23, the screen display is scrolled up one line (line 23 is cleared and the new output is printed on line 23).

NOTE: Printing a line feed on line 23 causes the screen to scroll one line without affecting cursor position.

A line feed is automatically issued with a carriage return when using the /005 address Refer to Section 7.4.2 below.

7.4.2 Addressing the Screen

As discussed in Section 7.2, all devices attached to the system are assigned a three-digit code, the first digit of which refers to the device type and classification. This code identifies which class of I/O device is being used, enabling the operating system to generate the appropriate I/O procedures.

The following is a table which indicates the device types normally used for character printing and display operations to the screen:

Type 005 Outputs a line feed character (HEX(0A)) after each carriage return character (HEX(0D)) is output.

Type 205 Does not generate an extra character after each carriage return character (HEX(OD)) is output.

Type 405 Outputs a line feed character (HEX(OA)) after each carriage return character, but suppresses the automatic carriage return character (HEX(0D)) normally issued on PRINT statements that do not end in a semicolon.

For addressing the screen, the device address used must always be:

x05

with x being either device type 0, 2 or 4, as explained above. The address (x05) cannot be changed by the application program.

The terminal screen is set up as the default address 005. This address can not be modified by the application program other than to change the device type as indicated above. It is not necessary to put this address in the device table using the $DEVICE statement.

7.4.3 The NPL Screen Character Set

NPL employs its own version of the character set for screen displays, which may be different from the native operating system running on the particular hardware version being used.

The NPL Normal Character Set contains the following:

HEX(00) to HEX(0F) NPL control codes.

HEX(10) to HEX(1F) International characters.

HEX(20) to HEX(7F) ASCII codes (except for the special codes 5F, 60, 7B-7F).

HEX(80) to HEX(8F) Special graphics characters.

HEX(90) to HEX(FF)   Contains the underlined versions of the codes HEX(10) to HEX(7F).

NPL also offers an Alternate Character Set, which differs from the Normal Character Set in just the characters HEX(90) to HEX(FF):

HEX(90) to HEX(9F)   Special graphics extensions.

HEX(A0) to HEX(AF)   Character box pieces.

HEX(B0) to HEX(BF)   Reserved.

HEX(C0) to HEX(FF)   Pixel graphics. When these characters are displayed, they are represented by different graphic combinations of a character space, which is divided into sixths of a character.

NOTE: The implementation of the NPL screen character set is highly dependent upon the capabilities of the display devices used. Refer to Appendix D for details. Refer to Section 1.5 for the ideal representation of the NPL Character Set.

To display the standard NPL character set actually present on any given machine, enter and run the following program:

```
10 DIM X$(16)16
20 SELECT PRINT 005, LIST 005
30 PRINT HEX(0202000E):      REM ENSURE STANDARD CHARACTER SET
40 FOR X=1 TO 256: STR(X$(),X,1)=BIN(X-1): NEXT X
50 LIST DIM * X$(
```

Now replace line 30 with the following:

```
PRINT HEX(0202020E)
```

and press RETURN. Running the program again with this change, displays the NPL Alternate Character Set.

Look carefully at the characters represented by the codes HEX(90) to HEX(FF), and notice the "pixel" graphic display of those characters in the Alternate Character Set. Refer to Section 7.4.9 below for details on "pixel" graphics.

Screen Output Character Translation

Except for the range of characters between HEX(20) and HEX(7F), there is little standardization of character sets across terminals. Therefore, if characters of the NPL character set were sent directly to screens, the resulting characters displayed would be quite different from one machine to another. NPL provides a mechanism for displaying the NPL character set consistently on every supported terminal.

Depending on the nature of the native operating system and the display device used, either modifiable fonts or character translation is used to resolve differences between various devices.

Modifiable Fonts--On some systems which support downloadable fonts, NPL provides an alternate font table which contains the NPL character set. This font table is downloaded to the video controller by the RunTime program at execution time. The font table can be modified by the EDFONT utility in the NPL Compiler utilities. Refer to Chapter 13 for details.

Character Translation--On some systems, a dynamic character translation technique is used. For machines which require this, character set compatibility is approximated by the use of a simple look-up table which translates characters into hex codes which generate an equivalent, or at least similar, display character.

The RunTime program contains a built-in default translation table which may be examined or modified by use of the $SCREEN statement. Refer the Statements Guide, $SCREEN, for details.

In addition, EDSCREEN, a screen translation utility is provided which allows a table file to be generated to disk. This table file is loaded by the RunTime program at execution time. For further details on the Compiler utilities, refer to Chapter 13 of the Programmer's Guide. For further details on the hardware specific implementation of screen character translation, refer to the appropriate NPL Supplement.

7.4.4 Internal Access

Some operating systems support two methods of accessing the display device. These are:

Direct video mapping	Characters are sent directly to the screen controller.
"Generic" screen handling	Standard native operating system characters are sent to the screen by using standard native operating system calls.

Direct video mapping is much faster than generic native operating system screen handling. However, to perform direct video mapping, the system software generating the screen output must recognize the type of screen that is on the system and must know the actual memory location to map to.

There may be minor differences in the results of screen output between direct video mode and generic mode. These differences are highly hardware dependent and can be referenced in the appropriate NPL Supplement for the particular hardware being used.

7.4.5 Screen Control Codes and Attributes

Through the use of various system reserved hexadecimal codes, the features of cursor movement and the terminal alarm may be controlled. In addition, various attributes for the screen display may be specified by the use of a hexadecimal sequence of codes. In both cases, the necessary codes can be sent to the screen by use of a PRINT statement when the default PRINT device in the Internal Device Table is set to address /005.

Use of these codes is operating environment and display device dependent.

For example:

```
10 SELECT PRINT 005
20 PRINT HEX(03)
```

sends the hexadecimal control code (03) to the screen.

The following table lists the supported control codes:

HEX(01)	Moves the cursor to the home position (top left of the screen).
HEX(02)	Indicates the start of a multi-character control sequence. Multi-character sequences always end in HEX(0E) or HEX(0F).
HEX(02050F)	Enables cursor blinking.
HEX(03)	Clears the screen and homes the cursor.
HEX(05)	Turns the cursor on.
HEX(06)	Turns the cursor off.
HEX(07)	Emits the terminal's audio alarm (a beep).
HEX(08)	Moves the cursor to the left one space (non-destructive backspace).
HEX(09)	Moves the cursor to the right one space (non-destructive space).
HEX(0A)	Moves the cursor down one line (line feed).
HEX(0C)	Moves the cursor up one line.
HEX(0D)	Moves the cursor to the beginning of the current line (carriage return).
HEX(0E)	Select enhanced video mode.
HEX(0F)	Select non-enhanced video mode.
HEX(0202020E)	Select alternate character font.
HEX(0202000E)	Select normal character font.
HEX(020D0C030E)	Terminal reset function. Clears screen

Video Attributes

For the ability to highlight different groups of information on the display device, a series of attributes is available to be selected for any character displayable on the screen. These attributes do not occupy a character position on the screen and do not affect the cursor location. Therefore, fields with different attributes can be displayed with no intervening spaces. These display attributes consist of:

Bright	Characters displayed in high intensity.
Blinking	Characters displayed in blinking fashion.
Reverse Video	Characters displayed as dark on a white background. On color displays, background and foreground colors are reversed.
Underline	Characters displayed with an underline.

To select a display attribute to be used, the following command is sent to the terminal:

HEX(02040E) or HEX(02040F)

where:

0204 The control code sequence that indicates to the display device that special character display attributes are to be selected.

The HEX codes, aabb, specifying the display attributes to be selected as follows:

aa =	**Description**
00	For normal intensity, no blinking
02	For bright, no blinking
04	For normal intensity, blinking
0B	For bright, blinking

bb =	Description
00	For normal video, no underlining
02	For reverse video
04	For underlining
0B	For reverse video, underlining

0E or 0F represents the termination character which causes the display attributes selected by aa and bb to be turned on (0E) or turned off (0F).

NOTE: Once an attribute has been selected (by the 0204aabb sequence) it can be turned on or off with the single character HEX(0E) or (0F). Once on, attributes remain on until turned off with the HEX(0F) or until another attribute is turned on. If an attribute is turned on using HEX(0E) instead of the control sequence, a HEX(0D) also turns off the attribute.

The actual effect of these control sequences is extremely hardware dependent. In all hardware versions, NPL attempts to provide as close an approximation as possible to the specifications above. Refer to the appropriate NPL Supplement for details on hardware specific features and considerations.

7.4.6 Graphics

Under certain environments, NPL has the capability of running in "Graphics Mode" as opposed to "Text Mode". This capability allows for the support of "true box" graphics (discussed below) through the use of downloadable screen fonts. The availability of graphics mode is extremely hardware dependent. Refer to the appropriate NPL Supplement for a complete discussion of Graphics Mode.

7.4.7 Color Support

Applications can be enhanced by use of color. NPL supports the use of color on specific operating systems. This section discusses language requirements necessary to use color in an application, as well as the colors available and the selection of colors using $OPTIONS or dynamically. Color support under NPL is operating environment and display device dependent. The hardware/operating system requirements are outlined in Chapter 6 of the appropriate NPL Supplement, as well as details on availability of color support.

NPL supports the following colors:

Value	Color
0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	WHITE

Color use can be controlled by using $OPTIONS bytes 22 and 23 with three functions available:

1. Color support may be enabled by setting byte 22 of $OPTIONS to a non-zero value.

2. Default background, foreground, perimeter and underline colors may be set by using specific values in bytes 22 and 23 (described below).

3. Color support may be disabled and the screen may be returned to the "non-color" defaults (white letters on black background with no perimeter and underline controlled by byte 1 of $OPTIONS) by setting byte 22 of $OPTIONS to HEX(00).

NOTE: Only values 0-7 are supported in bytes 22 and 23 of $OPTIONS. All other values should be avoided and will provide inconsistent results.

For any function, the values in bytes 22 and 23 of $OPTIONS are used only when a terminal reset sequence (HEX(020D0C030E) is issued. In other words, placing values into these bytes has no effect until the terminal reset sequence is issued.

For example:

```
10 DIM X$64
20 X$=$OPTIONS                    :REM Always preserve other values
30 STR(X$,22,1)=HEX(13)           :REM Set blue background, cyan foreground
40 $OPTIONS=X$                    :REM Place new values in $OPTIONS
50 PRINT HEX(020D0C030E)          :REM Put color selections into effect
60 PRINT "THIS IS A TEST"         :REM Text is printed as cyan on blue
70 REM ** Dynamic color selection is now enabled
80 STR(X$,22,1)=HEX(00)           :REM Value to turn off color
90 $OPTIONS=X$                    :REM Place value in $OPTIONS
100 PRINT HEX(020D0C030E):REM Screen now reverts to white on black
110 REM ** Dynamic color selection is now disabled
```

When used to specify specific color combinations, bytes 22 and 23 of $OPTIONS may be set to the following values:

Byte 22 = HEX(bf)	This byte provides power-on default colors for background and foreground, where b and f are values in the range of 0-7. Default value is HEX(00) (old color conventions).
Byte 23 = HEX(pu)	This byte provides power-on default colors for perimeter and underlined text. Where p and u are values in the range of 0-7. Refer to the section below for further information on underline color replacement.

NOTE: The values specified in the above examples take effect only when a power-on reset sequence HEX(020D0C030E) is printed to the screen. If byte 22 is set to HEX(00), all new color options are disabled and the RunTime continues to operate as it did in previous releases.

Dynamic Selection of Colors Using HEX Sequences

As an alternative to modifying color values in bytes 22 and 23 of $OPTIONS, colors may be selected dynamically by using the following HEX sequence:

```
HEX(02000605 0b 0f 0u 0p 0i 0F)
```

Where:

b, f, u, p, and i are hex digits with the following meanings:

b is the color to use in the background for subsequent characters.

f is the color to use in the foreground for subsequent characters.

u is the color to use for subsequent underlined text. Refer "Underline Color Replacement" below for further information.

p is the required perimeter color.

i is set to 0 if the perimeter color is dim, 1 if it is bright.

For example:

```
10 PRINT HEX(02000605040703020F)
```

The above sequence would set background to red, foreground to white, perimeter to cyan, and underline text to green.

```
10 PRINT HEX(020006050B030F)
```

The above sequence would leave the background unchanged (0B), set foreground to cyan (03), and leave the perimeter and underline text unchanged.

NOTE: The specified values of b, f, u, p and i must be in the range of 0-7 (colors corresponding to the IBM color codes) or may be "B" to indicate no change is required from a previous sequence.

Dynamic selection of colors is activated only if byte 22 of $OPTIONS was set to a non-zero value when the most recent reset sequence (HEX(020D0C030E)) was issued. This can be easily done in the boot or preboot program. Refer to the sample program provided below for an example.

WARNING--Use of a color value of 01 for any of the optional parameters in the dynamic color control sequence causes a Wang 2x36 terminal to hang.

For example, the sequence

HEX (02000605010304020F)

causes the Wang 2x36 to hang. Programmers who wish to use Dynamic Color Control Sequences and need the application to execute properly on a Wang 2x36 terminal either under NPL or Wang Basic-2, must avoid using color values or include conditional logic so that the color sequence is not sent to a Wang 2x36 terminal.

NOTE: Under NPL, as long as Byte 22 of $OPTIONS is equal to HEX(00), dynamic color sequences are not sent to the terminal, effectively avoiding this problem.

On some display adapters, line 25 is used for the perimeter color. This line 25 is otherwise reserved by the RunTime for optional CAPS LOCK and NUM LOCK indicator displays. Please refer to the appropriate NPL Supplement for further details.

A HEX sequence may be terminated at any time by a HEX(0E) or HEX(0F) sequence, in which case the remaining color options are unaffected. New colors become effective immediately and remain in effect until a new sequence of this type is printed or until a power-on reset sequence is printed.

NOTE: The HEX(02...0F) sequences have the added advantage that if they are printed using actual 2200 terminals (on a 2200 system), or using an older RunTime version, they are ignored (except for the problem with the HEX(01) sequence as noted above). Thus, color can be safely added to programs without fear of encountering compatibility problems. However, unless color equipment is mandatory, it is the developer's responsibility to ensure that the programs are still usable on a display which has no color capability.

Underline Color Replacement

Byte one of $OPTIONS (the high-order nibble) is used as the background color for underline. The "u" value from byte 23 of $OPTIONS or a dynamically issued HEX sequence is used as the foreground color if specified.

If the color options are disabled (byte 22 of $OPTIONS = HEX(00)), byte one of $OPTIONS is still used as the foreground and background underline replacement colors. Refer to the Statements Guide, $OPTIONS, for more detail concerning byte 1 of $OPTIONS.

Selecting "BRIGHT" Attribute

In addition to the eight colors that are available, foreground colors may be duplicated using the "BRIGHT" attribute, providing for a selection of 16 colors.

NOTE: This requires special hardware considerations. Refer to Chapter 6 of the appropriate NPL Supplement for details. This is achieved by sending the standard hex sequence for selecting the "BRIGHT" attribute. Refer to Section 7.4.5 for details.

Sample Color Program

The following is a short program designed to present an example of the concepts discussed in this section.

```
10 DIM X$64
 : X$=$OPTIONS
 : STR(X$,3,1)=HEX(01) :REM Cut down on "noise"
 : STR(X$,22,1)=HEX(47)  :REM White letters on Red
 : STR(X$,23,1)=HEX(12) :REM Blue perimeter Green underline
 : $OPTIONS=X$:REM Enable new color sequences
 : N$=HEX(02000605070004070F):REM Dynamic restore sequence
20 PRINT HEX(020D0C030E);:REM Power-on reset sequence
 : PRINT "The available background colors are:"
 : PRINT HEX(02000605000F);"0 - BLACK   " :
 : PRINT HEX(02000605010F);"1 - BLUE    "
 : PRINT HEX(02000605020F);"2 - GREEN   "
 : PRINT HEX(02000605030F);"3 - CYAN    "
 : PRINT HEX(02000605040F);"4 - RED     "
 : PRINT HEX(02000605050F);"5 - MAGENTA "
 : PRINT HEX(02000605060F);"6 - BROWN   "
 : PRINT HEX(02000605070F);"7 - WHITE   " :REM "invisible"
 : PRINT N$ :REM Back to normal
30 PRINT "The available foreground colors are:"
 : PRINT HEX(020006050B000F);"0 - BLACK   ",HEX(0E);"GRAY         "
 : PRINT HEX(020006050B010F);"1 - BLUE    ",HEX(0E);"LIGHT BLUE   "
 : PRINT HEX(020006050B020F);"2 - GREEN   ",HEX(0E);"LIGHT GREEN  "
 : PRINT HEX(020006050B030F);"3 - CYAN    ",HEX(0E);"LIGHT CYAN   "
 : PRINT HEX(020006050B040F);"4 - RED     ",HEX(0E);"PINK         "
 : PRINT HEX(020006050B050F);"5 - MAGENTA ",HEX(0E);"LAVENDER     "
 : PRINT HEX(020006050B060F);"6 - BROWN   ",HEX(0E);"YELLOW       "
 : PRINT HEX(020006050B070F);"7 - WHITE   ",HEX(0E);"BRIGHT WHITE "
 : PRINT N$            :REM Back to white letters
```

7.4.8 Box Graphics

Screen displays can be enhanced by the use of boxes to enclose specified portions of the screen. These boxes can be produced by the PRINT BOX statement. Two types of boxes are supported: "true" boxes and "character" boxes. The primary distinction between these two types of boxes is the horizontal attribute of the box. In true boxes, the horizontal attribute is displayed between character rows. In character boxes, the horizontal attribute occupies one character row.

True Boxes

The implementation of true box graphics requires a monitor that can support dual text and graphics mode. If such a monitor is present and is supported by the RunTime program, then true box graphics are displayed. In all other cases, if the monitor is supported but does not have dual text and graphics capabilities, or the monitor is simply not supported (generic screen handling used), true box graphics are not available.

Application programs may examine byte 4 of the $MACHINE system variable to determine if true box graphics are available on the machine currently being used. A value of "G" indicates that true box graphics are available. A value of " " indicates that they are not. For further details refer to the Statements Guide, $MACHINE.

Character Boxes

Box graphics created from the standard character set are supported. This provides the capability of printing boxes on screens where true boxes are not available. This feature does have some limitations. The primary limitation is that horizontal lines must occupy a full character row. This means that applications which wish to use these character boxes must allocate a blank row above and below the text to be boxed. Because of this limitation, the use of character boxes is optional (controlled by the $BOXTABLE system variable). $BOXTABLE is also used to define the characters used for the components of character boxes.

NOTE: If character boxes are selected, they are printed in place of true boxes even on systems which have true box capability. As noted above, the availability of true boxes can be tested by accessing the $MACHINE system variable. Thus, character boxes can be selected conditionally.

Character boxes are generally more "portable" than true boxes. For further details refer to the Statements Guide, $BOXTABLE. Also, refer to Appendix D of the Programmer's Guide for details on which terminals support true or character box graphics.

7.4.9 Pixel Graphics

As explained in Section 7.4.3 above, NPL offers an Alternate Character Set, which displays the alternate characters in the range HEX(90-FF), including the characters from HEX(C0) to HEX(FF) in a "pixel" graphic mode.

These 64 characters are represented by different graphic combinations of a character space, which is divided into sixths of a character. Each of the character spaces is sectioned off into 3 rows by 2 columns, and are filled in by the proper HEX code specification. As adjacent graphic characters begin to touch, continuous lines of light and dark areas are displayed, making special displays such as bar graphs and histograms possible.

Refer to the program example in Section 7.4.3 above to display the pixel graphics on the terminal screen.

Because screen handling of graphics is extremely hardware dependent, refer to Appendix D for details on terminals which support use of "pixel" graphics or Chapter 6 of the appropriate NPL Supplement.

7.4.10 Plotting Capabilities

For machines which have graphic capabilities, use of Niakwa's Plotter Driver software allows NPL programs to perform plot-like functions to screens with graphic capabilities. Characters may be sent to the plotter driver through standard NPL PRINT or $GIO statements. Refer to the Niakwa Scientific and Communications Drivers Package for further details on plotting capabilities.

NOTE: Because plotting capabilities are extremely hardware dependent, refer to the appropriate NPL Supplement for details on this availability.

7.4.11 Statements Used to Print to the Screen

The following NPL statements are available for use in directing printed output to the display devices:

PRINT	Used for printing the values of specified variables or expressions to the currently selected output device.
PRINT AT	Used to position the cursor to any location on the screen for beginning printed output.
PRINT BOX	Used for producing a tabulated format of printed output.
PRINT TAB	Used for producing a tabulated format of printed output.
PRINTUSING	Used to create a formatted print output of numeric and alpha-numeric data in a user defined format.
MAT PRINT	Used for printing specified array values, row by row, to the currently selected output device.
$GIO	Used for various functions difficult to achieve using other statements.
PRINT SCREEN	Used to print a specified portion of the screen from a specified variable. Intended to be used with INPUT SCREEN to temporarily save then redisplay a portion of the screen.

For further details refer to the NPL Statements Guide.

7.4.12 132-Column Support

NPL permits application programs to make use of the 132-column mode on NPL supported display devices that have this capability. Refer to Appendix D for details on which specific terminals support this feature. This feature enables application programs to present wide reports (132 columns) to a display device for review before (or in place of) printing them to a print device. This option allows for dynamic switching between the 80-column mode, which is more legible and usually sufficient for most displays, and the 132-column mode which may be useful for reports but is less legible.

Byte 17 of the $OPTIONS system variable has been allocated as a flag to indicate whether dynamic selection of the 80 column or 132 column mode is to be performed by the RunTime program. If set to HEX(00), the RunTime does not change modes and the screen remains with the last selected width. If set to HEX(01), each time the screen is cleared (by use of the HEX(03) control code, or HEX(020D0C030E) sequence), the currently selected display width (from last SELECT PRINT, SELECT LIST or SELECT CO, whichever is appropriate) is examined. If it has a value of 132, the 132-column display is used. If the value is not 132, the 80-column display is used.

Byte 8 of the $MACHINE system variable contains a one-byte binary value indicating the current display width. This can be inspected by programs that need to know the currently selected width, to properly format the output. It may also be used by routines to change the screen width temporarily and restore the width to the original value when finished. It should also be examined by programs which are attempting to change the screen width after they have cleared the screen, to verify that the change has taken place.

For example:

```
0010 DIM X$64,X0$1
   :M$=$MACHINE
   :W0=VAL(STR(M$,8))   :REM GET CURRENT SCREEN WIDTH
   :X$=$OPTIONS
   : X0$=STR(X$,17,1)     :REM SAVE DYNAMIC ENABLE STATE
   : STR(X$,17,1)=BIN(1) :REM ENABLE DYNAMIC SCREEN SIZE
   : $OPTIONS=X$
   : SELECT PRINT 005(132)
   : PRINT HEX(03);
   : M$=$MACHINE
   : W=VAL(STR(M$,8))     :REM GET ACTUAL SCREEN WIDTH
   : IF W=132 THEN PRINT "WIDE SCREEN IN USE"
   : ELSE PRINT "WIDE SCREEN NOT AVAILABLE"
   : GOSUB 2000           :REM PRINT REPORT TO SCREEN WIDTH=W
   : SELECT PRINT 005(W0):REM SET BACK TO PREVIOUS WIDTH
   : PRINT HEX(03);
   : X$=$OPTIONS
   : STR(X$,17,1)=X0$     :REM RESTORE DYNAMIC ENABLE STATE
   : $OPTIONS=X$
```

NOTE: Changes to the $OPTIONS system variable can also be made using the NPL Edit Options Utility (EDOPTION). Refer to Chapter 13 for details on this utility.

7.4.13 Uses and Limitations of INPUT SCREEN/PRINT SCREEN

Two extremely useful NPL screen handling commands are INPUT SCREEN and PRINT SCREEN. These commands allow for easy implementation of "pop-up" type features to be added to existing code or the restoration of the screen after a $SHELL command is used.

INPUT SCREEN only recognizes information which has been displayed by NPL print class output directed to a device address x05. Use of any other function, such as external calls which update screen or native operating system messages, may result in incorrect data being returned by INPUT SCREEN.

Refer to the NPL Statements Guide, INPUT SCREEN or PRINT SCREEN for details on the uses and limitations of these commands.

7.5 Keyboard Support

NPL assumes the existence of an "ideal" keyboard which contains all of the keys required for the full use of the language. This keyboard is referred to as the NPL Virtual Keyboard (refer to Section 1.5). Each of the keys on the Virtual Keyboard has a label, and when depressed, generates a one byte value to the NPL program. For each of these keys, the label and code generated is constant, irrespective of the hardware or native operating system which hosts NPL. All references to keyboard keys in this guide refer to the Virtual Key label (usually in capital letters, e.g., EXECUTE, RETURN, EDIT, etc.).

In reality, the physical keyboard of the system which hosts NPL may have very different standards for key labels and the codes from that of the Virtual Keyboard. These differences are resolved by the $KEYBOARD system variable. This is a translation table which maps the physical keyboard onto the NPL Virtual Keys. The RunTime include a pre-determined $KEYBOARD table which contains suitable keyboard mapping for the specific hardware and operating system as labeled on the package. This table may be used as is, or may be modified by the developer to suit individual preferences.

This technique ensures application program compatibility across all hardware and operating environments without requiring program changes or special program logic to handle varying keyboards.

7.5.1 The Keyboard Device Address

The keyboard device address is special, in that it is a constant on all NPL platforms and is set to /001 by the RunTime at initialization. This cannot be changed by the application program. As such, it is not necessary to put this in the Device Equivalence Table.

7.5.2 Virtual Key Names and Codes

The Virtual Keyboard contains two types of keys: "Standard Keys" and "Special Function Keys". Both types of keys generate a one-byte code when depressed. Furthermore, some Standard Keys generate the same one-byte code as some Special Function Keys, however, their "special" designation allows that they be processed differently by both NPL itself and NPL programs.

Generally, Special Function Keys (SF Keys) are used by NPL for editing operations and function calls. Application programs which use the KEYIN statement may also use Special Function Keys for other customized purposes (discussed later in this section). The following table lists the Virtual Key names and codes that they generate, as required by NPL:

NOTE: The physical key names on specific keyboards may vary from these. Consult the appropriate NPL Supplement to determine the exact location of these keys on the keyboard being used.

NPL Virtual Key Code(HEX)	NPL Virtual **Key Label**
	Standard Keys are:
08	BACKSPACE
0D	RETURN
82	EXEC (or EXECUTE)
84	Continue
A1	LOAD
E5	LINE ERASE
	plus all standard ASCII key labels and codes.
	Special Function Keys are:
00 ... ’0F	SF Key number 0 to 15
’10 ... ’1F	SF Key number 16 to 31
’42	PREV
’43	NEXT
’45	SOUTH
’46	NORTH
’48	ERASE
’49	DELETE

’4A	INSERT
’4C	EAST
’4D	WEST
’4F	RECALL
’50	SHIFT CANCEL
’52	SHIFT PREV
’53	SHIFT NEXT
’55	SHIFT SOUTH
’56	SHIFT NORTH
’59	SHIFT DELETE
’5A	SHIFT INSERT
’5C	SHIFT EAST
’5D	SHIFT WEST
’5F	D TAB
’7C	GL
’7D	SHIFT GL
’7E	TAB or FN
’7F	SHIFT TAB
’A0	UNDERLINE
’F0	CANCEL or EDIT

Use of Special Function Keys by NPL

The NPL line editor (Section 5.4) makes use of the Special Function Keys. Specifically, Special Function Keys numbered 0 to 31 (codes HEX(00) to HEX(1F)) perform system pre-defined text editing functions. These functions are documented in Section 5.4.4, Special Function Keys.

Where possible, these keys are mapped to a corresponding physical key bank of generically numbered "function keys" available on many physical keyboards.

The balance of the Special Function Keys (codes HEX(32) and up, as shown) are mapped to the remainder of the physical keyboard in a manner where the Virtual Key label matches or nearly matches the Physical Key label, if possible. In many cases, the line editor additionally employs these SF Keys as an alternative to performing the same edit functions as SF Keys numbered 0 to 31.

Details of this mapping are documented in the appropriate NPL Supplement.

The HALT Key

A special Virtual Key is provided which HALTs program execution if pressed when using the interpretive RunTime (RTI). Refer to Section 2.6.3 for details on the functions of the HALT key. Refer to Appendix D and/or Chapter 6 of the appropriate NPL Supplement for the keyboard sequence used to generate the HALT key.

NOTE: **The HALT key may be suppressed by use of byte 13 of the $OPTIONS system variable. For further details refer to the Statements Guide, $OPTIONS.**

The Underline Key

If a Special Function HEX(A0) is defined in the Keyboard Translation Table, the system allows entry of underlined characters as follows (during INPUT or LINPUT):

- The text which is to be underlined is keyed in.
- The arrow keys are used to reposition to the start of the text.
- The defined SF Key is pressed once for each character requiring underlining. After each depression, a single character is underlined and the cursor advances one place.

For programs using a KEYIN operation, the key defined as special function HEX(A0) is reported as special function HEX(A0), with no other side effects.

NOTE: **The default character translation tables do not define a special function HEX(A0) in some implementations of NPL, therefore this functionality is not available unless a replacement Keyboard Translation Table is defined.**

The standard underline key (shift of the "-" key) currently generates a HEX(5F), which is an underline character in the standard ASCII character set, but a back-arrow character in the standard Wang 2200 character set. With the underline character being used in constants and as part of Long Identifier Names (LINs), the generation of a "true" underline character may be more familiar to many programmers. The functionality of this key can be easily changed to display a true underline character by using $SCREEN (the PREBOOT program is the best place to define this change), however, keep in mind that to be able to specify certain filenames for some batch compile (B2C) operations requires the availability of the "back-arrow" type of underline key. For an example of implementing this logic, refer to the preboot program in Section 2.4.

Creating native operating system files with underlined names is not always permitted.

7.5.3 Keyboard Translation

As stated above, the key codes generated by the native operating environment typically do not correspond to codes expected by NPL programs. These differences are resolved by the use of a simple lookup table which translates keys received from the keyboard to HEX codes expected by NPL programs. The contents of this table vary from one machine to the next. However, in all cases, standard built-in defaults for keyboard remapping are present. Refer to Appendix D and/or the appropriate NPL Supplement for details on the display device being used--These should prove adequate for most applications.

Should modifications be required, they can be made to the Keyboard Translation Table by one of two methods:

1. The preferred method is to use the NPL utility, "EDKEYBOA", which is provided with the Development Package. This utility allows the programmer to create or modify a disk file containing keyboard equivalences which are to be loaded when the RunTime program is executed. NPL always looks for a keyboard file using a filename based upon the type of terminal that is attempting to execute the RunTime program. The assumed name and location of that keyboard file is detailed in Chapter 6 of the appropriate NPL Supplement.

NOTE: For some terminal types, the RunTime uses a built-in default keyboard translation values, should the keyboard file not be located. However, for many terminals, no built-in defaults are present and keyboard access may fail entirely unless the proper keyboard translation file is present.

2. Modification of the keyboard translation table may also be performed dynamically by the use of the NPL $KEYBOARD statement. For additional details, refer to the Statements Guide, $KEYBOARD.

For further details on the EDKEYBOA utility, refer to Chapter 13 of the Programmer’s Guide.

7.5.4 Statements Used for Keyboard Input

The following statements can be used for accepting input from the keyboard under program control. Refer to the Statements Guide for complete details on the use of these statements:

INPUT	Used for accepting multi-character numeric or alpha input from the keyboard into the specified variable.
KEYIN	Used for accepting input from the keyboard one character at a time. With the KEYIN statement, application programs can accept both Standard Keys and Special Function Keys as input. Provisions of the KEYIN statement allow that these keys are distinguished and therefore may be processed differently.
LINPUT	Used for accepting multi-character alpha input. With LINPUT, the NPL line editor is active and thereby provides editing features during entry.
MAT INPUT	Used for accepting multi-character input of values into alpha and numeric arrays.

NOTE: NPL Release IV now supports and "insert mode" for operation for LINPUT and INPUT, in addition to the default "overstrike mode" of operation. Refer to the Statements Guide, $OPTIONS, Byte 44, for details.

7.5.5 Extended Keyboard Entry

NPL supports a large part of the 2200 character set as "normal" characters during line entry (LINPUT, INPUT operations) and command entry. The main purpose of this is to support the use of non-English character sets which may require more than the standard range (HEX(10) through HEX(7F)). This also provides a mechanism for migration to different, extended character sets (such as WISCII, IBM extended or DEC international) which may be more suitable to the host environment.

NOTE: Use of such extensions means that data which uses characters outside the HEX(10)-HEX(7F) range as "normal" characters requires translation when moving between different operating systems.

The range of extended character input is controlled by two bytes of the $OPTIONS system variable. These two bytes are:

Byte 18	Sets low value of extended range	(default=HEX(00))
Byte 19	Sets high value of extended range	(default=HEX(00))

NOTE: With the default values, no extension of the character set takes place. Values of HEX(80) and HEX(FF) provide maximum extension of the character set (a total of 240 "normal" characters).

For example:

```
0010 DIM X$64<
   : X$=$OPTIONS<
   : STR(X$,18,2)=HEX(80FF)<
   : $OPTIONS=X$
```

These changes to the $OPTIONS system variable can also be made using the NPL Edit Options utility (EDOPTION). Refer to Chapter 13 for details on the operation of the NPL Edit Options utility.

Users are advised to beware of the following potential problems which may occur when the range of "normal" characters is extended:

- The RUN/EXEC key, assumed to be a key with a "normal" value of HEX(82), may become unavailable if HEX(82) is included in the supplementary range defined by $OPTIONS bytes 18 and 19. To permit continued use of the EXECUTE function (required for example when using the statement stepping mechanism under the interpreter), the line entry routine now also accepts "special" key HEX(82) as EXECUTE.

- The ERASE key, normally assumed to be a key with a "normal" value of HEX(E5) may become unavailable if HEX(E5) is included in the supplementary range defined by $OPTIONS bytes 18 and 19. To permit continued use of the ERASE key mechanism under the interpretive RunTime, the line entry routine now also accepts "special" key HEX(E5) as ERASE.

- NPL programming "atoms" which are included in the supplementary range defined by $OPTIONS bytes 18 and 19 become unavailable. Refer to Appendix E for details on use of NPL "atoms".

- The characters in the extended range replace the "alternate" font characters in the same range. Thus, use of an extended character set may make the special graphics in the range HEX(80)-HEX(9F), the character box graphics in the range HEX(A0)-HEX(AF), or the pixel graphics characters in the range HEX(C0)-HEX(FF) unavailable.

- While using an extended character set, the technique of setting the HEX(80) bit in each character of a string to underline it becomes invalid when the resulting character is in the auxiliary range defined by bytes 18 and 19. It is still possible to display underlined characters by use of an appropriate enhanced display sequence (HEX(0204...0E).

- Usually, correct display of the character set requires definition of a proper screen translation table and correct entry of characters requires definition of a proper keyboard translation table. These tables may be different for different types of terminals. Carefully document the "ideal" display of the extended character set of choice. Refer to Chapter 13 for details on the usage of the EDSCREEN and EDKEYBOA utilities.

7.6 Mouse Support

This section details standard NPL support of the mouse interface within supported environments.

Mouse events are reported to the NPL application by generation of special function key codes as follows:

’F1 - left button pressed
’F2 - left button released
’F3 - left button pressed (within double click time)
’F4 - right button pressed
’F5 - right button released
’F6 - right button pressed (within double click time)
’F7 - dragged north (up)
’F8 - dragged south (down)
’F9 - dragged east (right)
’FA - dragged west (left)

The values reported for mouse events are modifiable. The method used to modify these values is operating system dependent.

When a key is read by the NPL KEYIN statement, bytes 23 and 24 of $MACHINE contain the Y (row) and X (column) address of the current location of the mouse when the event occurred, if it is "on screen". When the cursor is outside the screen area (or if there is no mouse) these bytes contain high values (HEX(FF)).

The mouse cursor appearance is operating environment dependent, refer to Chapter 5 of the appropriate NPL Supplement for more information.

Byte 29 of $MACHINE indicates whether keyboard mouse events are supported.

The values are:

HEX(00)	mouse not available
HEX(01)	mouse available

This value is operating environment dependent, refer to Chapter 5 of the appropriate NPL Supplement.

Support of a mouse interface is hardware specific and not supported in all NPL environments. Refer to Chapter 5 of the appropriate NPL Supplement for operating environment specific details regarding mouse support.

7.7 Terminal/Monitor Support

This section concerns itself with the terminals which are supported under the various supported NPL platforms. Inclusion of a terminal in this section does not guarantee that it will operate under all supported NPL environments. Refer to the specific NPL Supplement for more details on supported terminals on each operating system.

7.7.1 Determination of Type

On all versions of NPL, the terminal kind is determined by the RunTime at startup. Once a terminal kind is established, the RunTime initializes various internal NPL system variables. Among them, the $SCREEN and $KEYBOARD variables are loaded with the translation values stored in the KEYBOARD.xxx and SCREEN.xxx files for the corresponding terminal kind.

The method NPL uses to determine what terminal kind is attached at startup varies by operating environment. For example, under UNIX, NPL interrogates a user defined environment variable called BASIC2C_TERM, while under another environment such as SuperDOS, NPL will determines terminal kind from an ASCII text file containing a table of terminal kinds. In all cases, developers must establish these values at the native operating system level. Refer to Chapter 6 of the appropriate NPL Supplement for details on terminal type determination.

7.7.2 NPL Directly Supported Terminals/Monitors

The following table lists the terminals that are currently supported by the NPL RunTime. This table also contains the terminal 'kind' name which should be established at the native operating system level using the methods described above. Refer to Appendix D for a complete description of terminal characteristics for all terminals listed below, with the exception of the IBM console.

Terminal Name	Terminal "kind"
ALTOS III	altos
ALTOS V	al5
Bull HDS I	wy50
Bull HDS III	vt100
DEC VT100	vt100
DEC VT200	vt200
IBM 3151	i3151
NCR 4970	vt200
Spectrix SPX 701	spx
Wang 2110a	w2110a
Wang 2x36 DE/DW	w2236
Wyse 50	wy50
Wyse 60, 150, 160	wy60
Wyse 370	wy370

NOTE: The support of the IBM Console is operating-system dependent and is discussed in Chapter 6 of the appropriate NPL Supplement.

The following terminals are supported by the various versions of NPL RunTime.

NOTE: Not all terminals are supported by all versions of the NPL RunTime. Each terminal has a value assigned to it in byte 9 of the $MACHINE variable. This value is consistent across all operating systems that support the terminal. For details on which terminals are supported under a specific operating environment, refer to Chapter 6 of the appropriate NPL Supplement.

TERMINAL	$MACHINE BYTE 9 VALUE
Altos III	HEX(02)
Altos V	HEX(08)
Bull HDS I	HEX(06)
Bull HDS III	HEX(03)
DEC VT100	HEX(03)
DEC VT200	HEX(03)
IBM 3151	HEX(0D)
NCR 4970	HEX(03)

TERMINAL	$MACHINE BYTE 9 VALUE
Spectrix SPX 701	HEX(0C)
Wang 2110a	HEX(01)
Wang 2x36 DE/DW	HEX(07)
Wyse 50	HEX(06)
Wyse 60,150, 160	HEX(05)
Wyse 370	HEX(0B)

7.7.3 Local Printer Support

Local printers may be accessed under NPL by using a special printer device designation "LCL=Y", where supported. The device naming convention for redirecting output to a local printer will vary from operating system to operating system.

For example, under SuperDOS:

```
$DEVICE(/204)=">0 LCL=Y"
```

While under UNIX:

```
$DEVICE(/204)="/dev/tty LCL=Y"
```

Both of the above examples will direct all print output sent to the NPL print address /204 to the a terminal's local printer port. Refer to Chapter 6 of the appropriate NPL Supplement for the proper device naming conventions to use for local printing. Both serial and parallel local printers are supported. Refer to Appendix D for specific implementation notes for each terminal type.

NOTE: Local printers can be accessed only by the terminal to which they are attached.

7.7.4 Support of Native Operating System and Utilities

Each operating system supported by NPL supports specific terminal(s). Not all supported NPL terminals are supported on all operating systems. Refer to Chapter 6 of the appropriate NPL Supplement for details on specific terminals supported by NPL on that operating system and for details on which terminals are well suited for use with native operating system functions or applications.

7.8 Printers

I/O to printer class devices is handled differently than I/O to other NPL supported devices. This section discusses the methods by which output to printer class devices is accomplished.

7.8.1 Valid NPL Print Addresses

The following table indicates the device classes normally used for character printing and displaying operations. They are stored in the first character of the device address.

Type 0	Outputs a line feed character (HEX(0A)) after each carriage return character (HEX(0D)) is output.
Type 2	Outputs no extra characters after each carriage return character (HEX(0D)) is output.
Type 4	Outputs a line feed character (HEX(0A)) at the end of each line, but suppresses the automatic carriage return character (HEX(0D)) normally issued.

The second and third digits in the device address refer to the actual device address recognized by NPL as valid print class devices. Valid printer device addresses supported by NPL are in the range:

x00, x04, x05, and x10 through x1F

The code x05 is reserved for the terminal. Therefore, print output directed to address 005, 205 or 405 is displayed on the display device screen. The code 00 is reserved for a "null" device. Output directed to the 000, 200, and 400 addresses is not printed.

7.8.2 Equivalent Native Operating System Devices or Files

NPL print output may be directed to native operating system physical devices (usually printers) or to native operating system files (for spooled output). These files or devices are equated to device addresses through the use of the $DEVICE statement as described in Section 7.2.2 above. Refer to the appropriate NPL Supplement for specific naming conventions used for printer devices or files.

7.8.3 Specifying the Output Printer Device

The device address and line width for printer output under program control is always taken from the default PRINT entry in the Internal Device Table (DT). To assign a different device address requires that the particular device be selected for PRINT output. This is accomplished by using the SELECT statement.

For example:

```
SELECT PRINT 215(132)
```

assigns the address 215 to the default PRINT device in the Internal Device Table (refer to Section 7.2.3 above). All subsequent PRINT output from program mode is then directed to the native operating system file or device equivalent to address 215. Specifying the numeric parameter inside the parentheses establishes the column width of the printed output.

NOTE: The SELECT PRINT statement affects PRINT output generated in program mode only. As a debugging convenience, immediate mode PRINT command output is directed to the terminal screen regardless of the selected print address.

At system startup, the default address 005, which is the terminal screen, is selected for all print operations.

Refer to Section 7.2.3 for further details about the Internal Device Table. For further details refer to the Statements Guide, SELECT.

7.8.4 Available Print Statements

The following statements are available for use in directing printed output to a specified designated output device:

PRINT	Used for printing the values of specified variables or expressions to the currently selected output device.
PRINTUSING	Used to create a formatted print output of numeric and alphanumeric data in a user-defined format.
MAT PRINT	Used for printing specified array values, row-by-row, to the currently selected output device.

$GIO	Used for various functions difficult to achieve using other statements.

For further details refer to the NPL Statements Guide.

7.8.5 Control Sequences

Many printers have a variety of special capabilities such as program selectable pitch, character fonts, etc., which can be accessed through the use of specialized control codes. NPL allows complete control over generation of printer control codes to be performed by the NPL program. That is, any characters (HEX(00)-(FF)) can be printed by the program, and these characters are passed through to the native operating system device or file unmodified by NPL.

NOTE: There may be exceptions to this on certain environments (refer to the appropriate NPL Supplement for details).

NPL does provide an alternative to program generation of printer control codes. This is the PRINT CONTROL function of the HELP processor. Refer to PRINT CONTROL, in the appropriate NPL Supplement for details.

7.8.6 Use of Native Operating System Files

Print output can be directed to native operating system files instead of actual printers by use of the DET to equate the NPL device address with a native operating system filename. This technique has several applications:

- Print output can be spooled to disk and physically printed at a later time, possibly in the background if the native operating environment supports this. This makes the generation of the report much quicker in most cases.

NOTE: Some operating environments supported by NPL support print spooling mechanisms that can be addressed by the DET. In these cases, this is a preferred method for spooling output since the NPL program does not have to be concerned with naming of the spool file. Refer to the appropriate NPL Supplement for more details on printing to files.

- Directing print output to disk can provide a convenient method of passing data to other non-NPL software operating on the native machine. Many such applications have provisions for accessing data stored in an ASCII format, which is the result of spooling print output.

NOTE: For successful transfer to other applications, some minor modifications may be required to report programs. Typically, these modifications involve the removal of headings, totals, and other extraneous information so that the actual data is spooled in a consistent format.

7.8.7 Printer Translation

There are times when the code being generated by the RunTime is not producing the required character for the printer. This is mainly seen when trying to print non-English characters. These differences are resolved by the use of the Printer Translation Table option. The Printer Translation Table option permits the automatic translation of characters as they are sent to a printer (or spool file) using a simple lookup table. This simplifies application programs which deal with foreign language character sets and extended character sets when the type of printer to be used is known.

The Printer Translation Table option is implemented by the use of the 256-byte system variable $PRINTER.

An options clause in the $DEVICE specification has also been implemented to specify that $PRINTER translation should take place. The syntax of the $DEVICE options clause is:

```
$DEVICE(/xxx)="device-name XLA= Y"
```

where "xxx" is the NPL printer address
"device-name" is the specified printer device

If no "XLA= Y" clause is specified, no translation takes place.

Each byte of the $PRINTER system variable contains a native operating environment code which represents the corresponding NPL character set entry.

For example, each time an application sends a HEX(60) to a printer with the XLA= Y option, the RunTime substitutes byte 97 (= VAL(HEX(60))+ 1) of the $PRINTER system variable.

Only one $PRINTER translation table may be defined. If an application must support multiple printers with different character set requirements, the $PRINTER variable must be set under program control to a value which is correct for that printer.

For example:

```
1000 DIM X$256
   : X$=$PRINTER
   : STR(X$,VAL(HEX(10))+1,1)=HEX(83) :REM a circumflex on IBM PC
   : STR(X$,VAL(HEX(11))+1,1)=HEX(88) :REM e circumflex on IBM PC
   : STR(X$,VAL(HEX(12))+1,1)=HEX(8C) :REM i circumflex on IBM PC
   : STR(X$,VAL(HEX(13))+1,1)=HEX(93) :REM o circumflex on IBM PC
   : STR(X$,VAL(HEX(14))+1,1)=HEX(96) :REM u circumflex on IBM PC
   : STR(X$,VAL(HEX(15))+1,1)=HEX(84) :REM a umlaut on IBM PC
   : STR(X$,VAL(HEX(16))+1,1)=HEX(89) :REM e umlaut on IBM PC
   : STR(X$,VAL(HEX(17))+1,1)=HEX(8B) :REM i umlaut on IBM PC
   : STR(X$,VAL(HEX(18))+1,1)=HEX(94) :REM o umlaut on IBM PC
   : STR(X$,VAL(HEX(19))+1,1)=HEX(81) :REM u umlaut on IBM PC
   : STR(X$,VAL(HEX(1A))+1,1)=HEX(85) :REM a grave on IBM PC
   : STR(X$,VAL(HEX(1B))+1,1)=HEX(8A) :REM e grave on IBM PC
   : STR(X$,VAL(HEX(1C))+1,1)=HEX(97) :REM u grave on IBM PC
   : STR(X$,VAL(HEX(1D))+1,1)=HEX(8E) :REM A umlaut on IBM PC
   : STR(X$,VAL(HEX(1E))+1,1)=HEX(99) :REM O umlaut on IBM PC
   : STR(X$,VAL(HEX(1F))+1,1)=HEX(9A) :REM U umlaut on IBM PC
   : $PRINTER=X$
   : $DEVICE(/215)=">P1 XLA=Y"
```

NOTE: Some devices, especially serial printers, may have difficulty in receiving characters outside the standard ASCII range (HEX(20) to HEX(7F)) unless the correct number of bits (8) and parity (none) are set.

This type of translation does not allow for the possibility that some characters available on a printer can only be accessed by the use of a control code sequence. Such characters can only be accessed if provision has been make by the application software to send the control sequence.

Programmers who wish to send binary sequences (e.g., graphics) should avoid the use of the XLA= Y option, since the RunTime translates all bytes sent to the printer under the assumption that they are characters (not graphic bit patterns or control sequences).

The default value for the $PRINTER system variable depends on the version of the Run-Time used (the Wang version assumes printer understands WISCII character set, VAX version assumes DEC character set, etc.), but is usually suitable for the production of spool files based on the standard character set of the system.

Modifications to the $PRINTER system variable can be made using the $PRINTER statement as shown above or by use of the "EDPRINTC" utility supplied with the NPL Development Package. Refer to Chapter 13 of the Programmer's Guide for details on using this utility. For further details refer to the Statements Guide, $PRINTER.

7.8.8 Local Printer

Many of the terminals supported by the RunTime can support a local printer attached to them. This is a particular advantage when using the terminal on a remote basis. Local printers can be accessed from an NPL application by use of the LCL=Y clause in the $DEVICE statement for the printer address to be used.

Refer to Appendix D for details on local printer support for particular terminals or 7.7.3 above.

7.8.9 Special End-of-Line Handling

The RunTime program contains special logic that attempts to make end-of-line characters correspond to those expected by standard printers on the operating system in use.

For example, under MS-DOS, a HEX(0A) is inserted in print class output after each HEX(0D) printed.

This special logic is enabled or disabled by use of the ALF clause on the $DEVICE statement for the print address in use. Specification of ALF= Y enables special end-of-line handling, specification of ALF= N disables special end-of-line handling. The default of ALF= Y is assumed if no ALF clause is specified.

For example, under MS-DOS,

```
$DEVICE(/215)="LPT1 ALF=N"
```

assigns address 215 to the parallel printer with special end-of-line handling turned off.

NOTE: ALF= N would typically be required when using printers which are configured for operation on a Wang 2200/CS.

End-of-line handling can also be dynamically enabled/disabled by the operator using the Print Control function of the Help Processor.

The exact effect of the end-of-line handling varies among operating environments. Refer to the appropriate NPL Supplement for details on end-of-line handling with respect to the operating system being used.

NOTE: More than one clause may be specified on a particular device.

For example, the device equivalency for a local printer that must use printer translation and wants to turn the automatic line feed off, would appear as follows under SuperDOS:

```
$DEVICE(/204)=">P0 LCL=Y XLA=Y ALF=N"
```

7.9 Serial Devices

NPL can be used to address other native operating system devices. However, such access is limited to simple read or write operations unless a specialized device driver is used. Any native operating system device can be accessed by the NPL program by using the $DEVICE statement to establish a DET entry for that device with an associated NPL device address. The device should usually be defined as a PRINT type device (address 004,010-01F).

The RunTime contains limited ability to read and write raw data from a serial port using the C620 $GIO microcommand. This ability is highly operating environment dependent. For example, serial ports on the IBM and compatibles under MS-DOS can be accessed through the special device designation "COMx" in the $DEVICE statement, where "x" is the physical "COM" port number.

7.9.1 Directing Output to Other Devices

The simplest method of directing output to a device is to PRINT to it.

For example:

```
10 $DEVICE (/219) = "TTYPORT"
20 SELECT PRINT 219
30 PRINT ...
```

causes all the output from all subsequent PRINT statements (in program mode) to be directed to the native operating system device "TTYPORT".

NOTE: This method provides no mechanism for detecting that the device addressed is ready to accept data or has been properly configured.

NPL PRINT statements may be used to direct output to the serial port. In addition, $GIO output microcommands, both single character and multi-character, may be used.

NOTE: Output to a serial port is handled the same as output to a printer. That is, automatic line feed insertion (unless it is overridden by the ALF option of $DEVICE) and character translation (if specified by the XLA option of $DEVICE) are in effect.

7.9.2 Accepting Input

The only method of accepting input from such devices is by use of the $GIO statement. KEYIN cannot be directed to devices other than the keyboard.

For example:

```
10 DIM L$10, A$80:REM $GIO CONTROL, BUFFER<
       : $DEVICE (/01C)="TTYPORT TMO=Y":REM READ DATA FROM TTYPORT
20 $GIO /01C(HEX(C620),G$)A$():REM LOOK FOR DATA ON PORT<
       : L=VAL(STR(L$,9),2):REM GET # OF BYTES RECEIVED<
       : IF L=0 THEN $BREAK:REM CHECK NO DATA AVAILABLE<
       : IF L0 THEN GOSUB xxx:REM PROCESS DATA<
       : GOTO 20
```

reads bytes from the native operating system device "TTYPORT".

Use of the "TMO" clause allows operations directed to the "TTYPORT" to return to program control without waiting for an end-of-record indicator to be present. If this clause was not specified, then execution of the $GIO statement at line 20 causes the program to wait for input to be received, which could "hang" the program if data is not present. Refer to the Statements Guide, $DEVICE for further details on the TMO clause. The ability to read data from serial devices may have several restrictions depending on the operating environment in use. Refer to Chapter 5 or the appropriate NPL Supplement for further details.

Input from the Serial Port

The following discussion refers to functionality when the TMO= Y option has been specified in the $DEVICE statement, as explained above.

Some versions of the RunTime contains limited support for accepting input from a serial port. This support for input from the serial port is intended to provide a simple mechanism for utilizing serial input devices where the interface requirements are very simple and straightforward. Applications which require the more sophisticated features of the Wang 2227b Buffered Asynchronous Communications Controller board should refer to the Niakwa 2227 emulation driver provided in the Niakwa Scientific and Communication Drivers package, where available.

The $GIO microcommand C620 now accesses binary data in the buffer for the port specified and returns as many bytes of data as available; zero bytes if no data is available.

For example, under MS-DOS

```
10 DIM L$10,A$80                    :REM GIO control, buffer<
: $DEVICE(/01C)="COM1 TMO=Y"        :REM read from "COM" port 1
20 $GIO/01C(C620,L$)A$              :REM look for data on port<
: L=VAL(STR(L$,9),2)                :REM get # of bytes received<
: IF L=0 THEN $BREAK                :REM check no data available<
: IF L>0 THEN GOSUB xxx             :REM process data<
: GOTO 20
```

In this example, the first L bytes of A$ are the data returned from the port.

This method of accessing data as input on the serial port is very primitive and has several restrictions that the programmer must consider:

- Communications parameters for the port must be established externally to NPL.
- If the buffer overflows, incoming characters are lost. There is no flow control for incoming data.
- There is no error detection.
- There is no method for checking signals on the line (e.g., device not attached or not ready).
- Communication parameters are set by the MS-DOS "MODE" command.
- None of the $GIO microcommands supported by the Niakwa 2227 emulation driver is supported using this technique. Program modifications are required to support input from the serial device using this method.

HINT: For many applications, use of the 2227 emulation driver (where available) remains the recommended choice for asynchronous communications on the IBM PC.

7.9.3 Telecommunications Support

Through its Scientific and Communications Drivers Package, Niakwa offers a more complete software capability for handling asynchronous communications. The 2227 Emulation Driver performs the physical I/O to the serial board required to support asynchronous communications from an NPL program.

This driver performs buffering of input and output to a standard communications port (where available). Both input and output are interrupt driven, allowing the NPL application program to process buffered data without risk of data loss at normal speeds.

A translation table for received data may be defined, including end-of-record character detection.

A translation table for transmitted data may be defined. Protocols with 5 or 6 data bits may request automatic shift character insertion for transmission, and automatic shift character removal on reception.

The current status of the "controller" may be obtained, which indicates the number of characters received at the port, the number of characters waiting for transmission, and the status of DCD (modem ready) and DSR (terminal ready) status lines.

Refer to the NPL Scientific and Communications Drivers Package for more details on supported devices and NPL statements used.

NOTE: The SCD package is not available for all operating environments of NPL.

CHAPTER 8

ERROR HANDLING

8.1 Overview

To aid in the debugging of programs, NPL has a variety of errors that can potentially arise during program editing or execution. When an error occurs, the line number and statement at fault is displayed, along with an appropriate error code is displayed, followed by a short description of the error. NPL offers the ability to handle these errors under program control with several NPL statements. This avoids the possibility of inadvertent halting of an application, and allowing the program to recover itself from the error.

Section 8.2 discusses the different types and classes of NPL errors.

Section 8.3 describes the difference between recoverable and nonrecoverable errors.

Section 8.4 describes the difference between resolution and execution errors.

Section 8.5 discusses ways to handle error trapping under program control with the use of NPL error statements.

8.2 Error Classes

There are nine different error code classes, each categorized by the first digit of the error code. The second and third digits identify the specific error condition that occurred. Errors codes of 100 or greater are available only in NPL, and consequently, are not supported on the Wang 2200. The following table outlines the nine classes of errors:

Error Code Range	Error Class
A00-A09	Miscellaneous Errors
S10-S29	Syntax Errors
P30-P59	Program Errors
C60-C69	Computation Errors
X70-X79	Execution Errors
D80-D89	Disk Errors
I90-I99	I/O Errors
100-199	Reserved
200-499	Extended NPL Errors
500-799	External Errors
800-999	Reserved

8.2.1 RunTime Program Errors and Warnings

When an error occurs, the terminal beeps, the line and statement at fault are displayed, and the letters "ERR" followed by the 3-digit error code and description, appear on the screen. For multi-line statements, only the faulty statement is displayed, preceded by one of more colons. Each colon indicates how many statements into the line the error occurred at (i.e., ":::" is equivalent to three statements). Refer to Appendix B for a complete listing of the individual error codes.

For example:

```
1000 ::: X=1/Y

ERR C62: Division by Zero.
```

indicates the statement X= 1/Y is in error, and this is the fourth statement on line 1000.

8.3 Recoverable Versus Non-Recoverable Errors

Some NPL errors are recoverable, while others are non-recoverable. When handling errors under program control, recoverable errors can be trapped, allowing execution of the application to continue. Non-recoverable errors are usually the result of some condition that does not allow program execution and will immediately halt the program with the appropriate system error message (these types of errors are usually detected at resolution time). Recoverable errors can only occur once the program or module is fully resolved, and is executing. Some errors which occur at execution time are considered serious enough that they certainly indicate a programming error, for which it could be hazardous to continue execution.

1. An example of a non-recoverable error which occurs at resolution time:

```
0010 DIM RecA$100, RecB$200<
   : DIM RecA$200
```

Here, a variable (RecA$) is declared twice with conflicting attributes (the string length is different). This clearly indicates a programming error, and the program will stop at resolution time with an error P59 (Illegal Redimension).

2. An example of a recoverable error which occurs at execution time:

```
0010 REM Get the next record<
   : DATA LOAD DC #1,B$<
   : ERROR DO<
   :   REM an error occurred on the I/O operation<
   :   E=ERR  :REM get the error code<
   :   IF E=88                :REM check for invalid data type<
   :      'ReportInvalidDataType<
   :   ELSE<
   :      PRINT "Unexpected Error";E<
   :      STOP #<
   :   ENDIF<
   : ENDO
```

Here, the program is taking precautions against invalid or corrupted data in a data file. Any recoverable error codes which occur when executing the DATA LOAD statement will cause the statements in the ERROR DO:...:ENDDO block to be executed, otherwise these are skipped. The numeric error code value can be inspected to ensure that the error is one which the program can reliably recover from.

3. An example of an unrecoverable error which occurs at execution time:

```
0001 DIM LIMIT=1000
0010 FUNCTION 'Check(K)<
   : DIM T<
   : T=K<
   : WHILE T < LIMIT<
   :    T+=1<
   :    IF 'Check(T) > K<
   :            RETURN (T)<
   :    END IF<
   : WEND<
   : RETURN (-1)<
   : END FUNCTION
0100 ;Mainline<
   : PRINT 'Check (12)
```

Here, a condition involving excessive use of recursion can cause an unrecoverable error to occur after the "Check function has called itself a large number of times. In this case, the logic of the program causes the 'Check function to always call itself to a depth of LIMIT calls before returning the value -1. Since each level of recursion uses some stack, if the LIMIT value is large enough, the available stack will be exhausted, and this is a condition which cannot be remedied by the programmer. An unrecoverable A01 will occur in this case.

The following table lists all error-code ranges and indicates which ones are recoverable:

Error Code Range	Recoverable or Not recoverable
A00-P36	Not recoverable
P37	Recoverable
P38-P47	Not recoverable
P48	Recoverable
P49-P59	Not recoverable
C60-I99	Recoverable
100-199	Reserved
200-299	Not recoverable
300-499	Recoverable
500-599	Not recoverable
600-799	Recoverable
800-899	Reserved

NOTE: There are exceptions: I/O errors or other errors that may occur when loading a module in an INCLUDE statement may not be recoverable.

Refer to Section 8.5 of this chapter for more information on the handling of recoverable errors under program control.

8.4 Resolution Versus Execution Errors

Errors can occur in two different instances: during program resolution, or during program execution. Resolution is the time between the loading of a program and the time at which it can begin execution. During this time, the RunTime must resolve all variable references, dimensions, and any line or subroutine references made as a call or branch. If the RunTime finds any inconsistency with the program code such that it cannot execute, a resolution error is displayed. Most resolution errors are non-recoverable. Examples of resolution errors are miscellaneous errors A00-A09, syntax errors S10-S29, most program errors P32-P59 (exceptions are P37 and P48,) and most extended NPL errors 200-299 (exceptions are 200 and 233.)

For example:

```
0010 PRINT"Program Started..."
0020 GOTO 100
```

Here, a P36 error (Undefined Line-Number) occurs before the program starts (this occurs at resolution time) since the reference to line 100 is to a line number that does not exist.

Once the resolve pass is made successfully without error, program execution begins. During execution, any errors that occur are considered execution errors. These errors are not detectable at resolve time, because the error is the result of a value or external condition that is not apparent at the start of the program. Most execution errors are recoverable. Examples of execution errors are computation errors C60-C69, execution errors X70-X79, disk errors D80-D89, and I/O errors I90-I99.

For example:

```
0010 PRINT"Program Started..."
0020 A=1
   : B=0
   :  C=A/B
```

Here, a C62 error (divide by zero) occurs in line 20 during execution. This means that line 10 was executed, but execution halted when the error was encountered on line 20.

The concept of resolution-time versus execution-time becomes more ambiguous when using INCLUDEd modules. Errors can occur while NPL is evaluating functions that are located in a fully resolved INCLUDEd library module.

For example:

```
0010 ;RUN Module<
    :INCLUDE T"LIBRARY"<
    :Size = 'LibFunctionToFindArraySize<
    :DIM Array(Size)
----------------------------------------------
0010 ;LIBRARY Module<
    :FUNCTION 'LibFunctionToFindArraySize/PUBLIC<
    :RETURN(1/0)<
    :END FUNCTION
```

Here, an error C62 (divide by zero) occurs during the resolution of the RUN module. The LIBRARY module is resolved, but the RUN module is not. Even though the error occurs at resolve time for the RUN module, it occurs at execution time in the LIBRARY module, which is fully resolved, and, therefore, is considered recoverable within the LIBRARY module.

To trap for this condition, several lines have been added to the LIBRARY module in the above example:

```
0010 ;RUN Module<
    :INCLUDE T"LIBRARY"<
    :Size = 'LibFunctionToFindArraySize<
    :DIM Array(Size)
----------------------------------------------
0010 ;LIBRARY Module
    :FUNCTION'LibFunctionToFindArraySize/PUBLIC<
    :RETURN(1/0)<
    :ERROR DO<
    :RETURN (1000)<
    :END DO<
    :END FUNCTION
```

In the scope of programming good error handling routines, it is best to concentrate on whether or not the errors are recoverable or not recoverable (refer to Section 8.3.). Although most recoverable errors occur at execution time, and most non-recoverable errors occur at resolution time, it is possible with multiple modules to have some recoverable errors occur at resolution time, as demonstrated above.

8.5 Handling Under Program Control

NPL provides several statements to aid in error handling and error detection. The perfect program has no syntactical errors, but some errors are always possible due to hardware failure, etc. The ability to handle these errors during program execution improves the integrity of the application and provides a "cleaner" interface to the end-user.

8.5.1 ERROR DO

The ERROR DO statement provides program control of recoverable errors. Only the following error codes are recoverable:

P37, P48, C60-I99, 300-304, 600-799

Refer to Section 8.3 for more discussion on recoverable versus non-recoverable errors.

When a recoverable error is detected in a simple statement that is immediately followed by an ERROR DO statement, the standard system error response is suppressed and execution continues with the statements within the ERROR DO/END DO statements.

When using the DO/END DO group format of ERROR, any statements within the DO/END DO group are executed only if an error occurs. If an error does not occur, program execution resumes with the first statement following the END DO statement.

For example:

```
0010 Q=T/W
   : ERROR DO<
   :   Q=0<
   : ENDDO<
   : T=T+1
```

Here, the statement Q = 0 is executed only if an error has not occurred on the statement Q = T/W, but the statement T = T + 1 is always executed.

NOTE: Although not recommended, a simpler form of the ERROR statement can be used without the DO clause. When using the statement format of ERROR, any simple statements on a program line following an ERROR statement are only executed if an ERROR occurs. If a statement is followed by ERROR and the statement executes without an error, program execution continues with the first statement on the next line. Structured statements are permitted on the statement form of ERROR, but their use is not encouraged since breaking such a structured statement into multiple lines would mean that no error on the simple statement preceding ERROR could branch into the middle of a structured statement.

Math errors that have been suppressed using the SELECT ERROR statement do not generate errors (default values are returned; refer to SELECT ERROR for details) and, therefore, are not detected by the ERROR statement. In these cases, programs can detect the occurrence of suppressed errors by use of the ERR function (discussed in the following section).

For more details on ERROR and ERROR DO, refer to the Statements Guide.

8.5.2 ERR

The ERR statement returns a unique two or three-digit error code of the most recent error condition.

Whenever a RunTime error is detected, ERR is set to the appropriate error code. Any reference to the ERR function resets the value to zero. ERR is also reset to zero whenever a RUN or CLEAR command is executed. Refer to Appendix B of the Programmer’s Guide for a table of error codes.

NOTE: ERR is typically used in conjunction with the ERROR statement.

For example:

```
0010 DATALOAD BAT (X)X$()<
    : ERROR DO<
    :   E=ERR<
    :   PRINT "ERROR";E;"OCCURRED"<
    : END DO


0010 LIMITS T"FILE1",A,B,C,D<
    : ERROR DO<
    :   GOSUB 100<
    : ENDDO
0020 REM NO ERROR - NORMAL PROCESSING
.
.
.
0100 REM ERROR ROUTINE<
    : X=ERR<
    : IF X=48 THEN 110<
    : IF X=93 THEN 120<
    : RETURN
0110 REM Error P48 would run this<
    : STOP
0120 REM Error I93 would run this<
    : STOP
```

ERR may also be inspected to determine if an overflow or mathematical error has occurred, and the error is currently suppressed by the SELECT ERROR setting. For example, assuming a SELECT ERROR > 69 has been executed, errors between 60 to 69 cannot be detected by the ERROR statement.

For example:

```
0010 X=ERR : REM CLEAR ERROR<
    :SELECT ERROR>69<
    :Q=T/W<
    :IF ERR>0 THEN 100
0020 PRINT "Executes only if error has not occurred"
.
.
.
0100 E=ERR
    : PRINT "WARNING - Error ";E;" has occurred!"
    : STOP
```

Here, assuming that error 62 has been suppressed by SELECT ERROR, the statement in line 20 executes an error has not occurred on the statement Q=T/W.

For more details on ERROR and ERR, refer to the Statements Guide.

8.5.3 ERR$

The ERR$ function returns a string of text that describes the specified error-code, normally obtained from an ERR statement. The specified error-code is either a two-digit integer representing an original Wang 2200 error code or a 3 digit integer representing an extended NPL error code, that matches a valid error-code listed in Appendix B.

For example:

```
: DIM X$33
: X$=$ERR(01)
: PRINT X$
: Memory Overflow

: X$=$ERR(200)
: PRINT X$
: Handle table full / cannot expand
```

The text returned is identical to the literal message that appears when an untrapped error occurs in immediate mode. Text for error messages is stored in the file ERRORMSG.HLP with pointers to text stored in the file ERRORMSG.IDX. These files are included with every RunTime package and should be installed in the NPL directory. Refer to the appropriate NPL Hardware Supplement for further details on installation procedures.

NOTE: Text returned by ERR$ should be used for information only. Text is subject to change in future releases or may be modified by the developer.

If, for any reason, these files cannot be accessed, or if the specified error-code is invalid, ERR$ returns all spaces.

For example:

```
0010 $FORMAT DISK T/D10,<
    : ERROR DO<
    :   E=ERR<
    :   E$=ERR$(E)<
    :   PRINT "Error ";&E;" - ";E$;" occurred"<
    : END DO
```

For more details on ERROR and $ERR, refer to the Statements Guide.

8.5.4 $OSERR

The $OSERR function returns a string of text that contains the most recent native operating system error code and message received when an NPL error has occurred. $OSERR cannot appear on the left side of an equivalence statement.

The text returned comprises two parts:

1. The native operating system error code.

2. A descriptive message based on this code. The descriptive message is obtained from the file RTIxERR.HLP (where x is a single letter representing the host operating system--i.e., "I" for MS-DOS). If the correct file for the operating environment in use is not present or cannot be accessed (along with the associated RTIxERR.IDX file), a descriptive message is not returned by $OSERR.

NOTE: The RTIxERR.HLP file may be modified. If modified, it is necessary to execute the Indexed Help File Creation utility to create the associated RTIxERR.IDX file before the modified RTIxERR.HLP file can be properly accessed. Refer to Chapter 11 of the Programmer's Guide, for further information on indexed help files. Refer to the appropriate NPL Hardware Supplement for further information on the name and required location of RTIxERR files for the operating system.

The $OSERR value is returned even if the NPL error is trapped by ERROR or ON ERROR.

Typically, native operating system errors are generated only on I/O operations. If the NPL error was for a non-I/O related operation (i.e., X75-Illegal Number), $OSERR is typically blank.

NOTE: Each operating environment has its own set of individual operating system errors.

$OSERR is intended to be used only for informational purposes. Programmers are advised against attempting to base any logic decisions on the values returned by $OSERR. These values vary widely among different operating environments and may even vary widely on different revisions of the same operating system.

For example:

```
0010 $FORMAT DISK T/D10,<
    : ERROR DO<
    :   E=ERR<
    :   E$=ERR$(E)<
    :   PRINT "Error ";&E;" - ";E$;" occurred"<
    :   N$=$OSERR<
    :   IF N$<>" " THEN PRINT "Native operating system error - ";N$<
    : END DO
```

For more details on ERROR and $OSERR, refer to the Statements Guide.

8.5.5 Error Help Files

The NPL RunTime Package supplies two files which contain the literal text for the error codes that are generated. The following is a brief description of each of those files:

The ERRORMSG.HLP file contains the literal text for NPL error codes. The text is displayed by the RunTime in the case of a non-recoverable error or in the case of a recoverable error that is not trapped by the application. The associated text is also returned by the ERR$ alpha function. This file must be present in the appropriate operating system directory where the RunTime is installed to generate the text.

The RTIxERR.HLP file contains the literal text for native operating system error codes. The text is displayed by the RunTime in the case of a non-recoverable error or in the case of a recoverable error that is not trapped by the application, whenever the error is based on an error reported by the native operating system. The associated text is also returned by the $OSERR alpha function. This file must be present in the appropriate operating system directory where the RunTime is installed to generate the text. The file names and associated error messages are unique to each hardware version of NPL. Refer to the appropriate NPL Hardware Supplement for the correct naming conventions used.

NOTE: The ERRORMSG.HLP and RTIxERR.HLP files can be modified by the developer so that different error descriptions are displayed. This is particularly useful for developers of non-English applications. However, if these files are modified, they must be processed by the Indexed Help File Processor utility to rebuild the associated .IDX files. Refer to the Chapter 11 of the Programmer's Guide for details on Indexed Help Files.

CHAPTER 9

SYSTEM VARIABLES

9.1 Overview

NPL maintains numerous internal system variables which it uses to provide a wide variety of functions. These system variables are all initialized by NPL at the startup of the RunTime. In addition to providing information to NPL, these system variables can be interrogated by developers, allowing NPL's environmental features and options to be fine-tuned, from one native operating system to the next.

NOTE: This chapter provides an overview of each system variable and one or more of its potential uses. Each system variable discussed below is fully documented in the NPL Statements Guide.

Section 9.2 discusses $MACHINE.

Section 9.3 discusses $OPTIONS.

Section 9.4 discusses additional NPL system variables.

9.2 $MACHINE

$MACHINE is a 64-byte system variable containing information about the environment in which the RunTime program is currently operating. Developers may interrogate $MACHINE to determine information such as terminal types, the current operating system, available co-processors, and much more. This information can be used to implement conditional logic for selecting options based upon the current operating environment.

Unlike $OPTIONS (discussed below), NPL dynamically updates various bytes in $MACHINE (i.e., user counts, mouse coordinates, etc.) Therefore, $MACHINE values may not be modified.

NOTE: Placing $MACHINE on the left side of an assignment statement will result in a syntax error.

With NPL Release IV, the first 29 bytes of $MACHINE are currently used. New bytes will likely be added in future revisions.

NOTE: New values may be added for new hardware/operating environments supported by NPL. Refer to the appropriate NPL Supplement for details on specific $MACHINE values for the operating environment being used and for details of hardware-dependent features.

9.3 $OPTIONS

$OPTIONS is a 64-byte system variable containing initial default values for a wide variety of NPL options upon startup of the RunTime. The values maintained in $OPTIONS allow developers to take advantage of certain features that may not be available across all NPL environments. In essence, $OPTIONS allows developers to fine-tune their applications to a specific operating environment without sacrificing portability of the application.

Unlike $MACHINE (discussed above), developers can modify specific $OPTIONS bytes to take advantage of specific available features. For example, in a common boot or pre-boot program, the developer could interrogate BYTE 3 of $MACHINE (Monitor Type). Upon finding a value of HEX(43) ("C"olor), the programmer could conditionally set $OPTIONS bytes 22 and 23 with a set of predefined values to invoke background, foreground, underline and perimeter colors for the application. Terminals recognized by NPL as monochrome would branch through this logic.

The following is a small example of invoking several $OPTIONS features:

```
0010 DIM OPTIONS$64,MACHINE$64
0020 OPTIONS$=$OPTIONS                  : REM Get Current OPTIONS Values
0030 MACHINE$=$MACHINE                  : REM Get Current MACHINE Values
0040 IF STR(MACHINE$,3,1)=HEX(43)<
   : STR(OPTIONS$,2,1)=HEX(00)          : REM Turn Off Keyboard Status<
   : STR(OPTIONS$,3,1)=HEX(01)          : REM Set "NOISE" Suppression On<
   : ENDIF
0050 IF STR(MACHINE$,10,1)=HEX(01)       : REM Co-Processor Available?<
   : STR(OPTIONS$,16,1)=HEX(01)         : REM Use Co-Processor<
   : END IF
0060 $OPTIONS=OPTIONS$                  : REM Put New Options Into Effect
```

Programs which modify $OPTIONS should always store the current contents into a work variable, modify the work variable, and then reset $OPTIONS. The work variable should always be dimensioned to 64 bytes. This prevents problems, should additional bytes of $OPTIONS be implemented (these are always being added to). In many cases, it may be desirable to modify $OPTIONS based upon values identified in $MACHINE, as the above example illustrates.

NOTE: With NPL Release IV, the first 45 bytes of $OPTIONS are currently used. New bytes will likely be added in future revisions. Refer to Chapter 8 of the appropriate NPL Supplement for operating environment-specific details of the various $OPTIONS bytes

9.4 Other System Variables

The following section describes the other NPL system variables that are available.

9.4.1 $BOXTABLE

$BOXTABLE is a 17-byte system variable used to enable and select the character box set to be used for output of PRINT BOX statements.

NPL supports two forms of box graphics. "True" boxes and "Character" boxes. $BOXTABLE is used for the generation of character boxes only. Developers can enable character boxes by setting byte 1 of $BOXTABLE to HEX(01). If byte 1 of $BOXTABLE has been enabled, "Character" box generation will take precedence over "True" box generation.

NOTE: NPL will automatically generate "True" box graphics on devices capable of generating them, if byte 1 of $BOXTABLE is set to its default value of HEX(00).

Refer to the NPL Statements Guide for a complete discussion of $BOXTABLE.

9.4.2 $KEEPREMS

$KEEPREMS is a one-byte system variable which is used in the interpretive RunTime (RTI) to control generation of p-code used to maintain REM (remark) statements, program text indentation, and the display format of literals. Refer to the $KEEPREMS section in the NPL Statements Guide for more detailed information.

9.4.3 $KEYBOARD

$KEYBOARD is a 576-byte system variable which contains a keyboard translation table that is loaded from a KEYBOARD.xxx terminal support file. NPL determines which KEYBOARD.xxx file to load based upon the terminal kind determined by the RunTime at startup.

Since all terminals are not the same, NPL uses the $KEYBOARD translation table to translate hex values for keys received from the terminal in use, to a common set of hex values throughout the NPL operating environment.

Refer to Section 13.14 of this manual and the NPL Statements Guide, $KEYBOARD for a detailed discussion of keyboard support under NPL.

9.4.4 $NETID

The $NETID function returns the full network physical station number on those operating environments that support this option. This is a displayable 12-hex digit number and may be used wherever string literals are legal.

9.4.5 $PRINTER

$PRINTER is a 256-byte system variable which contains a printer translation table that is loaded using the default character set of the current native operating system.

The values in $PRINTER are character equivalents to be sent to a print device, in place of a character received from the NPL program. Byte 1 contains the replacement character for HEX(00), byte 256 contains the replacement character for HEX(FF).

NOTE **Printer translation is only performed if the XLA=Y clause has been specified in the $DEVICE statement of the print class device being used.**

Refer to Section 7.8.7 of this manual and the NPL Statements Guide, $PRINTER, for a detailed discussion of keyboard support under NPL.

9.4.6 $PROGRAM

$PROGRAM is a 64-byte system variable containing the same text string that is returned by the Program Load Sequence section of LIST DT. This text string contains the names of up to six program overlays, residing in the current "RUN" module in memory. The first program name is the first program loaded since the most recent CLEAR or LOAD RUN statement. The next five program names are the last five programs loaded since the most recent CLEAR or LOAD RUN statement. Refer to the $PROGRAM section in the NPL Statements Guide for more detailed information.

9.4.7 $REV

$REV is a 12-byte system variable containing the full revision of the NPL RunTime program currently executing. $REV may be assigned to any 12-byte alpha-numeric value and displayed. Refer to the $REV section in the NPL Statements Guide for more detailed information.

9.4.8 $SCREEN

$SCREEN is a 256-byte system variable which contains a screen translation table that is loaded from a SCREEN.xxx terminal support file. NPL determines which SCREEN.xxx file to load, based upon the terminal kind determined by the RunTime at startup.

Since all terminals are not the same, NPL uses the $SCREEN translation table to translate character equivalents for characters received from the terminal in use, to a common set of character equivalents throughout the NPL operating environment.

Refer to Section 13.15 of this manual and the NPL Statements Guide, $SCREEN for a detailed discussion of screen character translation under NPL.

9.4.9 $SER

$SER is a 6-byte system variable containing the serial number of the NPL RunTime program currently in use. $SER may be assigned to any 6-byte alpha-numeric value and displayed. Refer to the NPL Statements Guide, $SER, for more detailed information.

CHAPTER 10

$SHELL

10.1 Overview

The $SHELL statement effects a temporary exit from the NPL environment, and allows for interfacing with native operating system functions and programs.

Section 10.2 explains how the $SHELL statement may be used to execute specific native operating system commands, or to invoke the native operating system shell (command processor) for an interactive session.

Section 10.3 details how the $SHELL statement can be executed from Immediate Mode or under program control.

Section 10.4 explains how the $SHELL statement requires more memory than is used by the RunTime program itself.

Section 10.5 discusses how the $SHELL statement may be used for many useful applications and situations.

NOTE: The $SHELL statement is extremely dependent upon the native operating system in use. In this chapter, all discussions and examples of the usage of $SHELL relate specifically to the MS-DOS operating system. Refer to Chapter 8 of the appropriate NPL Supplement.

10.2 Uses of $SHELL

Following are several practical examples suggesting possible uses for $SHELL:

- When working within an application system, all native operating system commands and utilities are conveniently available to perform functions such as backups, restores, copying, deleting, renaming, etc., of native operating system files (of which diskimages, HELP files, demo scripts, and boot programs are a part).

- When entering native operating system shell commands, typographic errors or other necessary corrections may be easily corrected by entering shell commands from the NPL line editor (using the ! form of the command). This is especially convenient when the native operating system shell does not have a mechanism for recalling a command and correcting errors.

- Running a second copy of the Interpreter (if memory permits) may be useful in debugging program logic, where the exact position in an active partition should be preserved. In this manner, another NPL process can be executed like any other native program to track down a problem (e.g., to inspect the disk, load another program, etc.).

- Running a second copy of the non-interpretive RunTime, to perform the same function as point 3 above, however, with much smaller memory requirements.

- Running the Compiler in combination with the Interpreter. In some cases, the compiler has capabilities which the Interpreter does not, such as compiling batches of programs or creating Wang 2200 atomized code. These capabilities can be conveniently used by invoking the compiler from Immediate Mode.

- When developing an NPL application system, it may be useful to invoke a native operating system word processor to document features, bugs, notes, etc., and then return to NPL for further development or review, where the partition is undisturbed.

10.3 General Versus Specific Program Use

The $SHELL statement can be used to execute specific native operating system commands, or to invoke the native operating system shell for an interactive session from within NPL.

In either event, the status of the NPL program in memory at the time $SHELL is issued remains unchanged when $SHELL is completed. The contents of all variables is unchanged. Program execution will continue with the statement immediately following the $SHELL statement.

Operating system commands can be executed from a sub-shell (or task) created by the $SHELL command or as a parameter within the statement.

10.3.1 General Form

The general form of the $SHELL statement is:

Form 1:

```
$SHELL [literal-string] [,return-variable]
       [alpha-variable]
```

Form 2:

```
!  [literal-string]
   [alpha-variable]
```

Form 3:

```
! [command]
```

Where:

literal-string	=	string containing native operating system command to be executed.
alpha-variable	=	variable containing native operating system command to be executed.
return-variable	=	alpha-variable which will contain a return code generated (in binary) by the native operating system shell upon completion of the command. The size of this variable is operating environment specific (refer to the appropriate NPL Supplement)
command	=	any sequence of characters up to the end of the line (or a return graphic). A colon is not treated as a statement separator if it appears in the command.

Form 1 is recommended for use in programs. Form 2 can be used in programs, but not in Immediate Mode. Form 3 can be used only in Immediate Mode, and is specified without quotes surrounding the command argument. This statement, especially Forms 2 and 3, may also be referred to as the INVOKE command.

If no literal or alpha-variable is specified, or the specified literal or alpha-variable is blank, then the native operating system shell is loaded for an interactive session, at which point the user can then use any native operating system programs or functions available.

For example, entering the following in Immediate Mode invokes the native operating system shell.

```
:$SHELL
```

Upon completion of the interactive session with the native operating system shell, control is passed to the NPL environment, to the exact point of the initial exit. The current screen retains all information printed to it during the session, unless the shell was entered from the HELP display.

NOTE: Instructions for exiting the native operating system shell may vary. The RunTime always displays appropriate instructions before starting an interactive session.

The system performs an implicit $CLOSE on all files before $SHELL.

If it is desirable to retain the NPL screen, then the native operating system function of the HELP processor must be used.

10.3.2 Executing Specific Commands

Specific native operating system commands may be executed by specifying the command name explicitly in a literal or in a variable.

For example:

```
:$SHELL"DIR A:"    Would execute the command named DIR with parameter A:
                   (i.e., list the files in the directory of drive A)
```

(or)

```
:RUNWP$="WP"
:$SHELL RUNWP$     Would execute the command named WP (i.e.,
                   since WP is not a built-in command the system
                   attempts to locate a file called WP.BAT,
                   WP.COM or WP.EXE in the current directory)
                   Failing this, the system looks in other
                   directories in the current "path"
                   specification. If the file is found, it is
                   executed--as a program for .COM or .EXE     files,
                   or as a batch file if a .BAT file.
```

When a command is specified, control returns automatically to NPL upon completion.

10.3.3 Native Operating System File System

The ability to change native operating system default values relating to file location (such as "current directory" under MS-DOS) using the $SHELL statement is operating system-dependent. Refer to Section 8.2.4 of the appropriate NPL Supplement for details.

10.3.4 Return Code

The value of the return code, if specified, is operating system and command-specific. Refer to Section 8.2 of the appropriate NPL Supplement for details.

10.4 Command Usage

The Niakwa Programming Language provides three means of invoking the $SHELL command. One can execute it either from Immediate Mode, from the HELP processor or from program control. This section details each of these forms below.

10.4.1 From Immediate Mode

From Immediate Mode (the colon prompt), entry of the $SHELL statement begins the temporary exit from the NPL environment. The syntax for using this command is as described in Section 10.3 above.

An alternative syntax is supported for convenient use in Immediate Mode:

```
![command]
```

NOTE: The native operating system command must be specified with no quotes. In addition, no return-variable is supported with this form of the command.

From Immediate Mode the following commands are equivalent:

```
B$="DIR B:": $SHELL B$
$SHELL"DIR B:"
!DIR B:
```

10.4.2 From The HELP Processor

The HELP processor contains a new menu item which performs the same function as the $SHELL statement. Execution of the NATIVE OS option in HELP causes a temporary exit from the NPL environment, for an interactive session with the particular native operating system shell.

Upon completion of the interactive session with the native operating system shell, return is passed to the HELP screen.

NOTE: Instructions for exiting the native operating system shell may vary. The RunTime always displays appropriate instructions before starting an interactive session. For further details on the HELP Processor, refer to Chapter 11 of this guide.

10.4.3 From Program Control

Interfacing with the native operating system can also be accomplished from within an application or utility program. The $SHELL statement may be executed just like any other statement in the program, with the literal or alpha-variable containing the command to be executed. Some useful examples of this feature include:

- Support of an external tape backup utility. Temporarily exit the RunTime program environment, invoke a tape backup routine necessary for continuation of the application software. After completion of the backup, return to the next statement in the application program and continue processing.

- Print spooling from within a Novell NetWare environment. Instead of exiting the RunTime program to issue the CAPTURE command necessary to begin the local port spooling to a network printer, simply invoke the command from within the application program, using $SHELL, returning immediately to continue processing.

  For example:

  ```
  10 $SHELL "CAPTURE S=COMM Q=CAP NB NT NFF TI=5"
  ```

- An NPL program can act as a shell for the user, allowing the user to specify which native operating system programs to run under program control. Furthermore, menu programs can be developed in NPL which can then be customized to the end-user's environment so that NPL applications and native operating system applications can appear on the same menu.

For example:

```
10 REM THIS PROGRAM WILL EXECUTE A USER SPECIFIED COMMAND
20 LINPUT "Enter command to execute " -X$
30 $SHELL X$
40 REM DONE

10 REM THIS PROGRAM WILL LIST A USER SPECIFIED DIRECTORY
20 DIM X$40,Y$50
30 LINPUT "Enter name of the directory to display "-X$
40 Y$="DIR "&X$
50 $SHELL Y$
60 REM DONE
```

NOTE: Directory commands vary depending upon operating systems and hardware versions.

10.5 Memory Requirements

Both the Interpretive and Non-interpretive versions of the RunTime program require a certain amount of memory in which to operate. Any memory required for $SHELL operation programs typically is added to the standard RunTime requirements. Refer to Section 8.2 of the appropriate NPL Supplement, for details on memory requirements for $SHELL on the operating system in use.

Depending upon the nature of the interactive session planned using the Invoke command, memory requirements may prohibit access to the proposed program or function in certain environments. Should this condition arise, an insufficient memory message would appear (ERR A01 - Memory Overflow), effectively canceling the planned process.

For further details on memory use, refer to Chapter 3.

CHAPTER 11

THE HELP PROCESSOR

11.1 Overview

The HELP processor is a feature of the RunTime program which provides the user with access to a series of informative screens and options relative to the operation of a particular application program or the RunTime program itself. The operation of the HELP processor is discussed in detail in Chapter 2. This chapter discusses the preparation of HELP files which may be displayed by the HELP processor and use of the $HELP and $HELPINDEX statements to access specific HELP entries.

Section 11.2 discusses the creation of user-definable HELP screens for use with a particular application program.

Section 11.3 discusses the special sequences which may be included in HELP text.

Section 11.4 discusses the "nesting" of HELP screens to allow one HELP file to reference another, providing the operator with more detailed information.

Section 11.5 discusses the feature of indexing HELP files and combining several smaller HELP files into one larger one.

Section 11.6 provides a general discussion on HELP files.

11.2 User-Definable Help Screens

The incorporation of a HELP system into any application is beneficial in many ways. Embedding a detailed HELP system within an application lends additional credibility to it. If familiarized properly with the HELP system, end-user's can in many instances find answers to their questions without the use of a manual or placing a support call.

In addition, the HELP system can be demonstrated and used as an affective sales tool for prospective customers. HELP screens can be easily customized to individual customer requests, providing an additional level of flexibility to an application.

This section describes entry of HELP text, stand-alone HELP files, indexed HELP files and the $HELP statement.

11.2.1 The HELP ENTRY

The middle 19 lines of the standard HELP display are reserved for user-definable HELP text. This text area may be used to provide operator instructions for the various screen displays, prompts, and activities encountered during the execution of an application system.

If the HELP ENTRY occupies more than 19 lines of text, the first 19 lines of text will be displayed along with 2 additional HELP options; "Page Up" and "Page Down". These options as with all HELP screen options may be highlighted and executed from the keyboard or a mouse. Both the "Page Up" and "Page Down" options are displayed only when there is additional text to be displayed.

NOTE: In addition to the "Page Up" and "Page Down" options, the operator may choose to page forward or backward through the HELP text using the "PREV" and "NEXT" keys. On previous releases of NPL, the "PREV" and "NEXT" keys were used to move to the first or last HELP screen option.

The HELP text displayed may be automatically varied depending on which program of the application system is executing. Further, the text may be automatically varied depending on which portion of a program is executing. This capability allows for very specific instructions to the operator, depending on the exact circumstances.

There are two components to implementing HELP screens for an application system. First, a HELP ENTRY is created for each desired area of an application. Second, $HELP statements are strategically placed throughout the application to invoke the appropriate HELP ENTRY.

HELP ENTRIES are created and stored in NPL HELP files. An NPL HELP file is a native operating system file with an extension of .HLP (if permitted by the native operating system). A NPL HELP file may contain a single stand-alone HELP ENTRY or multiple HELP ENTRIES.

11.2.2 Stand-alone HELP Files

A stand-alone NPL HELP file contains a single HELP ENTRY. The name of this file is user defined. However, HELP filenames must be compatible with file naming conventions on the native operating system.

HINT: To be safe, restrict filename characters to the uppercase letters A-Z and numerals 0-9.

These files may be created using standard text editors. In the file, enter the instructional text to be displayed for a given program of an application should the HELP key be pressed. For example, presume a file "ARINFO.HLP" has been created and contains general AR informational text.

```
:$HELP = "ARINFO"
```

Adding the above program text to the beginning of AR entry in an existing application will cause the RunTime to reference the HELP ENTRY text contained in "ARINFO.HLP". There is no need to include the extension .HLP within the program.

11.2.3 Indexed HELP Files

Multiple HELP ENTRIES combined into a single file are referred to as INDEXED HELP FILES. Indexed help files can simplify the maintenance of a HELP system by reducing a large number of stand-alone HELP files into a single HELP file with an associated index. Indexed help files are discussed in detail in Section 11.5.

11.2.4 The $HELP Statement

To inform the RunTime program that a HELP ENTRY is available during execution of a given program, an NPL statement called $HELP is used. By assigning the name of the HELP ENTRY to the $HELP system variable in the program, the user is effectively forming a link from the program to its corresponding HELP ENTRY. The general form of the statement is:

```
$HELP = alpha-expression
```

Where alpha-expression may be up to an 8 character help filename or the name of an entry within an INDEXED HELP FILE.

For example:

```
$HELP="ARINFO"
$HELP="OEENTRY1"
```

Alternatively, the current status of $HELP can be examined using the form:

```
Alpha-variable = $HELP
```

For further details refer to the Statements Guide, $HELP.

When the HELP key is pressed during program execution, the RunTime program performs the following steps (assuming Indexed HELP Files are not used):

1. The current contents of the screen are saved (the application screen).

2. The RunTime program searches for the specified HELP file. If located, the HELP ENTRY text is displayed, beginning with the first 19 lines. If the file cannot be found or $HELP is blank, the message "No Help Information Available" is displayed.

3. If a HELP ENTRY is found, the operator may review the HELP text provided and resume execution of the application when ready.

NOTE: The method used by the RunTime program to search for specified HELP files is operating system-dependent. For example, under MS-DOS, the "current directory" is searched. Refer to the appropriate NPL Supplement for details. When supported by the native operating system, an extension of .HLP is assumed for the HELP filename.

11.3 Special Sequences

The HELP processor supports several special sequences which, when encountered in HELP ENTRIES, invoke special features of the HELP processor. These special sequences should be located in the actual HELP ENTRY text. The delimiter used for all special sequences are braces, "{}"

These special sequences are:

Tab Sequence - {T decimal Tab value }

This sequence causes text following the sequence to be displayed starting at the column specified by the tab value.

For example:

{T40} causes text following to be displayed starting at column 40.

HEX Sequence - {H hex string }

This sequence allows characters which are not normally enterable through the standard keyboard to be entered into HELP text. All such characters must be included in the hex string. This feature is particularly useful for creating HELP text which contain characters from the international character set.

NOTE: When the contents of the HELP file are displayed by the RunTime program, screen character translation, as described in Section 7.4 does take place. Therefore, to display the international character set, the NPL hex codes must be entered in the hex string. Refer to Appendix D for details on the NPL character set and the appropriate NPL Supplement for details on the implementation of the NPL character set for the machine being used.

For example:

{H98} causes the brace character, "{", to be displayed.

Cursor Location Sequence - {@ row value,column value}

This sequence locates the cursor to a specified row and column of the display. Rows and columns are numbered from 0.

For example:

{@10,15} causes the cursor to be displayed in row 10 starting at column 15.

NOTE: The braces should not be used for anything other than these special sequences. If it is necessary to display braces on the HELP display, the hex string special sequence must be used as described above.

The following sections discuss several other special sequences related to the specific section topic being discussed.

11.4 Nested Help Entries

Many times it is useful to have one HELP ENTRY reference another HELP ENTRY to provide the operator with more detailed information, but only if requested. This can be done using nested HELP ENTRIES.

A given HELP ENTRY can reference another HELP ENTRY by inclusion of the following special sequence in the HELP text.

```
{=HELPname,Reference Name}
```

where:

HELPname	The name of the next HELP ENTRY to display (sub-HELP). This may be the name of a stand-alone HELP file or the name of a HELP ENTRY in an INDEXED HELP FILE. The filename must be native operating system compatible, up to 8 characters maximum.
Reference Name	The text intended to appear on the screen to signify that more HELP is available. This text is highlighted, if possible, and is treated as a user-selectable HELP option.

This special sequence is termed the "HELP EXTERNAL REFERENCE".

To view the sub-HELP display, the operator moves the input block to the highlighted Reference Name and presses EXECUTE or selects the entry with the mouse (where mouse support is available). The sub-HELP display is then located (using the provided HELP-name) and displayed.

Upon selecting a sub-HELP entry, an additional HELP option, "Previous Help", is displayed in the upper left corner of the HELP screen, along with the "Leave Help" option. The "Previous Help" option allows the operator to move backward through one or more sub-HELP entries, while "Leave Help" is used to exit from the help processor back to the application.

Any HELP display may contain up to 64 sub-HELP special sequences. Sub-HELP displays themselves may also contain up to 64 references and so on. At any time the operator may press the CANCEL key to return to the original HELP display.

When using nested sub-HELP entries, NPL will maintain the last 5 sub-HELP screens referenced plus the originating HELP screen, for purposes of traversing backwards through the sub-HELP entries using "Previous Help". In the event a sub-Help entry is referenced recursively, all intermediate values are removed from the sub-HELP stack.

Nested HELP ENTRIES may be stand-alone HELP files or individual entries within an INDEXED HELP FILE (refer to Section 11.5 below). If they are stand-alone files, the rules stated in Section 11.2.4 above apply to location and display of the entry.

11.5 Indexed Help Files

While access to stand-alone HELP FILES is the simplest form of incorporating HELP entries into an application, it is not the most efficient method to use when considering a larger more complex, context sensitive help system. Here, developers may choose to integrate a number of HELP ENTRIES into one larger file. HELP ENTRIES combined in this manner are called "INDEXED" HELP files.

11.5.1 Advantages and Disadvantages of Indexed HELP Files

The advantages of Indexed HELP screen files are:

- Disk space occupied by a set of HELP files may be reduced.
- The number of files created is reduced. Access to files may degrade on some operating systems if the number of files is large.
- The number of indexed entries is unlimited.

The disadvantages of an indexed HELP file are:

- Each time the indexed HELP file is changed, it must be submitted to an indexing procedure. This indexing procedure is required to obtain rapid retrieval of individual HELP Entries (more on this below).
- The number of index entries is 3270 (unlimited on 32-bit operating environments), however, only 4K bytes of index entries (256 entries) are loaded and searched at time. Subsequently, access to later index keys may become progressively slower.

11.5.2 Creating an Indexed HELP File

To create an indexed HELP file, use a text editor appropriate for the particular hardware and operating system being used to create an indexed HELP file or to combine individual HELP files. The file should be saved with a name in any directory (if directory structures are supported by the native operating system), but the extension (if extensions are permitted) of the indexed HELP file must be .HLP.

11.5.3 Defining HELP Entries Within an Indexed File

Individual HELP ENTRIES within the INDEXED HELP FILE are defined by adding a label to the start of each entry. This label is termed the HELP Entry Definition. This must be added at the start of each individual HELP entry in the following format:

```
{=^HELPname,Reference Name}
```

where "HELPname" is up to an 8 character code used to retrieve the entry (by setting $HELP="HELPname" from a program).

If the Indexed file has been created by combining individual HELP files, set the HELP-name to be the name of the old HELP text file. Thus, any $HELP statements already referencing that name would not have to be modified. The Reference Name is used to resolve internal references to the entry (refer to next section). It should be a brief description (like a title) of the entry. The Reference name does not appear as part of the HELP display. If no other text appears on the HELP Entry Definition line, the display starts on the next line (otherwise, text on the remainder of the line is included in the display).

There must also be a way of determining the end of each individual HELP ENTRY. The method used is to place a pair of empty braces { } at the end of each HELP ENTRY.

11.5.4 Creating the Index for the File

To make an Indexed HELP file accessible by the HELP processor, the file must be processed by a special NPL utility. This utility reads the Indexed HELP file, resolves all HELP Entry Definitions (does not allow duplicates) and creates a secondary file which contains an actual index to the Indexed HELP file. This index file has the same name as the Indexed HELP file but an ".IDX" extension (on operating systems which permit extensions--on other operating systems, conventions for the Indexed HELP files are documented in the appropriate NPL Supplement.. Each time an Indexed HELP file is modified, this utility must be run to ensure that the index file is up-to-date. For further details on the HLPINDEX utility, refer to Section 13.18.

11.5.5 To Use an Indexed HELP File

Access to HELP ENTRIES in an INDEXED HELP FILE is similar to accessing a series of smaller HELP files. The name of the appropriate HELP ENTRY must be assigned to the $HELP system variable before the operator presses the HELP key and the name of the INDEXED HELP File containing the entry must be assigned to the $HELPINDEX system variable.

The general form of this statement is:

```
$HELP = alpha-expression
```

where alpha-expression is the native operating system file specification of a file containing the HELP ENTRIES.

To inform the system that the entry is contained in an INDEXED HELP FILE, the name of the INDEXED HELP FILE must be assigned to the system variable "$HELPINDEX".

For example:

```
$HELPINDEX="TUTORIAL"
```

This instructs the HELP processor to look for HELP ENTRIES in the INDEXED HELP FILE "TUTORIAL.HLP" (if extensions are permitted), with its companion index file "TUTORIAL.IDX".

Both the INDEXED HELP FILE and its INDEX file should reside in the same directory (if directory structures are supported under the native operating system).

It is also possible to examine the current value of the $HELPINDEX system variable by assigning it to a string variable (defined length is 50 bytes).

For example:

```
A$=$HELPINDEX
```

This places the current value of $HELPINDEX in the variable A$.

NOTE: Utility subroutines which wish to use a specific index file for HELP screens without affecting the current value, normally store the $HELPINDEX value and replace it before returning to the calling program.

Since a single Indexed HELP file can contain many individual HELP ENTRIES, very few $HELPINDEX statements are required. Typically, the $HELPINDEX statement can be placed in the startup program for a specific module, or even in the BOOT program for a small application.

For further details refer to the Statements Guide, $HELPINDEX.

11.5.6 Alternative Search

If a specified HELPname is not located in the INDEXED HELP FILE, or if $HELPINDEX is blank, or the index is missing or contains no matching entry for that name, the HELP processor looks in the current directory for "HELPname.HLP". If that file is not available, "No Help is Available" is displayed.

This search procedure is followed for all HELP files used by the RunTime program, including the following special names. For this reason, these special "HELPnames" should be reserved for their intended purpose:

RUNERROR	Displayed in the event of a non-recoverable error.
CTRLSEQ	Displayed if HELP is pressed in the Control Sequence field of the Printer Control display.
DUMPFILE	Displayed if HELP is pressed in the Dump File Name field of the Diagnostic Dump display.

11.5.7 External References within Indexed HELP Files

References to other HELP files may be included in indexed HELP files in the usual way (with a HELP External Reference):

```
{=HELPname,Reference name}.
```

11.5.8 Internal References within an Indexed File

If a reference to another HELP entry within the same indexed file is required, a more efficient method can be achieved by replacing the "HELPname" with a "pointer value". This is a special HELPname consisting of a decimal point and a number of digits (at least 5 are recommended, 6 may be required if the indexed HELP file is large).

This special sequence is termed a HELP INTERNAL POINTER and has the following syntax:

```
{=.pointer value,Internal Label}
```

The Internal Label is used both for displaying highlighted text on the screen as an indication that sub-HELP is available (like nested HELP screens) and as the reference to the HELP Entry Definition for the HELP Entry to be displayed. When the index file is created (by the Indexed HELP File Processor Utility, HLPINDEX), the Internal Labels with HELP Internal Pointers are resolved and all of the pointer values are corrected to point to the HELP Entry Definition based on the Reference Name of the HELP Entry Definition.

For example:

```
{=.00000,Format Floppy Disks}
```

The above example is taken from the UTILHELP.HLP Indexed HELP file which is provided as part of the NPL Development Package. This is the entry as it was originally entered into the file. It refers to the HELP Entry Definition:

```
{=^ATFORMAT,Format Floppy Disks}
```

After processing by the HLPINDEX utility, the HELP Internal Pointer appears as:

```
{=.00820,Format Floppy Disks}
```

Here, 820 is the calculated location of the HELP Entry Definition for ATFORMAT.

The result of this example is that when the HELP option Format Floppy Disks is selected during HELP processing of the UTILHELP Indexed HELP file, the HELP ENTRY for HELPname ATFORMAT is displayed.

NOTE: The spelling of the Internal Label must precisely match the Reference Name from the HELP Entry Definition. If it does not match, the HLPINDEX utility flags it as an error.

Internal references which use pointers are preferred over those using names, because the HELP entries they refer to can be displayed without referring to the HELP index.

The NPL Development Package contains an example of an indexed HELP file, UTILHELP, which defines HELP information for the NPL utility programs.

11.6 General Notes on Help Files

The contents of the $HELP and $HELPINDEX pseudo variables remain intact even when a new NPL program is loaded. Therefore, if a given program does not contain a $HELP statement, but the previous program executed did, the HELP file from the previous program is displayed if HELP is pressed during the current program. This could cause confusion for the operator. This can be avoided by the inclusion of a $HELP=" " or $HELPINDEX=" " statement in any program which uses defined HELP screens. This statement should be placed immediately before the exit point of the program.

$HELP may be assigned as many values as desired throughout the execution of a program or programs. Whatever HELPname is in the $HELP pseudo variable at the time of HELP key depression, is the HELPname used for HELP entry look-up and display.

For further details refer to the Statements Guide, $HELP and $HELPINDEX. For further details on the operation of the HLPINDEX indexed HELP file processor utility, refer to Section 13.18 of the Programmer's Guide.

11.7 Additional HELP Processor Features

11.7.1 Mouse Support

The HELP processor automatically supports the use of a mouse and all mouse functions where supported by NPL. Refer to Section 7.6 for a discussion on NPL Mouse support.

11.7.2 Keyboard Functionality

As an alternative to using the standard NPL "EXEC" key to execute HELP options within the HELP processor, a new $OPTIONS byte has been implemented to enable the HELP Processor to treat the "RETURN" key as if it were the "EXEC" key. Developers can enable this feature by setting Byte 46 of $OPTIONS to HEX(01).

The BACKTAB key (SHIFT+ TAB) now performs the same function as the LEFT arrow key. Within the HELP Processor, this key performed no function in previous releases of NPL

As discussed in Section 11.2.1, the "PREV" and "NEXT" keys are now used to page forward and backward through HELP ENTRIES with greater than 19 lines of text.

CHAPTER 12

$DEMO

12.1 Overview

$DEMO provides a mechanism which allows the application software to receive information from a special ASCII file as opposed to receiving it from the keyboard. Not only can this file provide keystrokes, it can also display information in the form of text boxes. This feature allows developers to create demonstration versions of their software, self-running tutorials, testing scripts, etc. In addition, $DEMO has a built-in macro recording feature for storing $DEMO scripts.

The following describes these capabilities can be used.

- **Automatic Software Demonstration**. Application software can be demonstrated using a prepared "script" to automatically fill all operator entered fields. The operator is provided with informational text along the way, and is only periodically required to press the space bar to continue with the demonstration.

- **Automatic Tutorials**. An automated training session can be prepared which allows for self-training at end-user sites on application software.

- **Built-in Macro Facility**. Developers can easily record keystrokes for their own $DEMO script files by using the SELECT LOG statement. These files can be used later with a "quick-key" macro.

Section 12.2 provides details on the $DEMO statement.

Section 12.3 provides details on the format of the script file.

Section 12.4 discusses keystrokes contained in the script file.

Section 12.5 discusses the display of informational text.

Section 12.6 discusses the adding of comments to the script file.

Section 12.7 discusses limitations of $DEMO.

Section 12.8 discusses the keyboard-logging option of $DEMO.

12.2 $DEMO Statement

The following section provides a discussion of the syntax of $DEMO, its effects, and how it can be analyzed.

12.2.1 Syntax

To enable the DEMO script function, the name of the script file to be accessed is assigned to the $DEMO system variable. The correct syntax for using this command is:

```
$DEMO=alpha-expression
```

The alpha-expression contains the native operating system file specification for the script data file. When the statement is executed, the DEMO processor is invoked and subsequent keystrokes and informational text are retrieved from the script file, until the end of the DEMO script is reached.

At any point in the program, the contents of the $DEMO variable may be inspected through the following syntax:

```
alpha-variable=$DEMO
```

12.2.2 Effect of Execution

When the $DEMO statement is executed, all keyboard input is redirected to the specified file. No keyboard keys function, except the following NPL keys:

Space Bar	This is used to continue with the DEMO script when it has been halted by a special screen display. Refer to Section 12.5 below for details on generating special screen displays using the BOX statement from within the DEMO script file.
CANCEL	Pressing CANCEL when the DEMO script has been halted by a special screen display exits from DEMO mode and returns the keyboard to its normal status.
Keypad Arrow Keys	These may be used to interactively reposition and resize special screen displays at execution time. Refer to Section 12.5 for details.
PLUS and MINUS keys	These may be used to speed-up or slow-down the rate of keystroke entry from the DEMO script file. Refer to Section 12.4.4 below for details.

NOTE: Refer to Appendix D for the actual keys to use on the terminal.

When the end of the DEMO script file is reached, the value of the $DEMO variable is not removed, but keystrokes and text are no longer read from the file. Keystrokes are then accepted normally from the keyboard. If the DEMO script file specified by the $DEMO system variable does not exist or is invalid, the $DEMO statement causes keystrokes to be accepted from the keyboard.

12.2.3 Determining $DEMO Status

The current status of keyboard redirection from a $DEMO script is maintained in byte 19 of $MACHINE. The HEX(01) bit is set to a value of 1 whenever keyboard redirection from a $DEMO file is in effect and is set to 0 on all conditions which terminate the $DEMO script. For details refer to the NPL Statements Guide, $MACHINE.

12.3 DEMO Script File

The DEMO script file, as specified by the $DEMO statement, can contain:

- Keystrokes to be passed to an NPL program in response to requests for keyboard input.
- Special text to be displayed in a box on the screen at different points within the DEMO.
- Comments.

Keystrokes for the script file may be created by use of the keyboard logging feature of NPL. However, typical use of a script file would include use of additional $DEMO features such as BOX and REM. Text for these items would have to be added to the keystroke file by use of a standard text editor (refer to Section 12.8 below for further details). Alternatively, the script file may be created by any standard text editor which can generate ASCII files (with no special control codes).

Any valid native operating system name may be used for the script file and it may be stored anywhere within the native operating system file system.

12.4 Keystrokes

Portions of the DEMO script file which are not part of a BOX display (refer to Section 12.5) or other special text (refer to Section 12.6) are treated as keystrokes to be passed to the NPL program in response to requests for keyboard input.

12.4.1 Keystroke Processing

Keystrokes are stored in the DEMO script file in the same order as an operator would have entered them (except special keys--refer below) while running the actual application software.

For example, a DEMO script file might contain the sequence:

```
This is a test(-RETURN-)
128.50(-RETURN-)
```

These keystrokes would be passed to the NPL program in response to subsequent keyboard input requests. All sequences surrounded by parentheses "()" are always treated specially in the DEMO script. To enter just a left parenthesis as a keystroke, use the special sequence (().

NOTE: The special sequence (-RETURN-) passes the keycode equivalent to the RETURN key (HEX(0D)). Refer to Section 12.4.2 for further details on special keys.

The following NPL keyboard input statements accept keystroke entry from a DEMO script file:

```
INPUT
LINPUT
KEYIN (non-polling form only)
```

For further details, refer to the Statements Guide for INPUT, LINPUT, and KEYIN.

The number of keystrokes passed depends on the program. A KEYIN statement accesses one keystroke at a time. However, a LINPUT or INPUT statement accesses keystrokes until the special keystroke (-RETURN-) is encountered.

For example, using the above stated DEMO script entry, the statements below would work as follows:

10 KEYIN A$	The first byte of the script sequence ("T") would be placed in A$. The next keyboard input statement would receive keystrokes starting with the second character ("h").
10 LINPUT X$	Would access keystrokes until the special (-RETURN-) keystroke is encountered ("This is a test"). Subsequent keyboard input statements would access keystrokes starting with the key after the (-RETURN-).

12.4.2 Special Keys

As the demonstration progresses, it becomes necessary to automatically insert special keystrokes which the operator would normally enter but may not be possible to store in a script file. Consequently, special keys have to be defined in a certain manner. Special keys are defined by the following special syntax:

```
(-special-key-)
```

For example:

```
12/31/92(-RETURN-)
```

This entry in the DEMO script file would cause insertion of the date 12/31/92, followed by emulation of the RETURN key being pressed.

The format of the "special key" is surrounded by parentheses and hyphens for many keys. Following is a list of the supported special keys:

(-RETURN-)	(-EXECUTE-)	(-SOUTH-)	(-INSERT-)
(-CANCEL-)	(-TAB-)	(-EAST-)	(-DELETE-)
(-BACKSPACE-)	(-NORTH-)	(-WEST-)	(-CLEAR-)
(-CONTINUE-)	(-LOAD-)	(-SHIFT-ERASE-)	
(-PRINT-)	(-UNDERSCORE-)	(-EDIT-)	(-HELP-)
(-RECALL-)	(-DEC-TAB-)	(-PREV-SCREEN-)	
(-NEXT-SCREEN-)	(-TAB-)	(-FN-)	(-GL-)
(-SHIFT-CANCEL-)	(-SHIFT-EDIT-)	(-SHIFT-NORTH-)	
(-SHIFT-SOUTH-)	(-SHIFT-EAST-)	(-SHIFT-WEST-)	
(-SHIFT-INSERT-)	(-SHIFT-GL-)	(-SHIFT-NEXT-SCREEN-)	
(-SHIFT-TAB-)	(-SHIFT-FN-)	(-SHIFT-PREV-SCREEN-)	
(-SHIFT-DELETE-)			

(() = LEFT PARENTHESIS

NOTE: (-HELP-) is disabled during the use of $DEMO.

Also supported are the special function keys, entered as (0) through (255).

12.4.3 Repeat Keys

Automatic repetition of keystrokes from a script file may be specified by entering a number followed by an "*" (asterisk), enclosed in parentheses, immediately followed by the character to be repeated.

For example:

```
(20*)-
```

produces 20 dashes (hyphens) as keyboard input data.

12.4.4 Timing

The rate at which keystrokes are processed (retrieved) from the DEMO script file is approximately .25 seconds per keystroke at the start of the demonstration. This rate may be increased any time during the demonstration by pressing the "+" key, or decreased by the "-" key. Each incidence of pressing a "timing" key changes the speed of retrieval by .01 second. Therefore, pressing the "+" key 25 times reduces the delay to zero.

Another method for automatically setting the retrieval speed of keystrokes from the DEMO script file is a special sequence with the following syntax:

```
(xx+)
```

This sets the retrieval speed to .xx (xx hundredths of a second) per keystroke.

NOTE: If no number is specified, the default value for "x" is the last value specified in a previous timing sequence. This rate continues until the (-) special sequence is reached in the DEMO script, which returns the rate to the original .25 seconds per key default (or to another value as modified by the operator by use of the + or - keys).

WARNING--Be aware that during the period in which the timing statement is in effect in the DEMO script, use of the "+" or "-" keys has no effect on the demonstration speed. They are disabled.

12.5 Display of Informational Text

When running a program using $DEMO scripts, it may be desirable to display comment boxes during execution to let users know what the program is doing. These comment boxes can be defined by use of a special sequence in the DEMO script file. The following is a detailed discussion on how to implement comment boxes.

12.5.1 The BOX Statement

Within the DEMO script file, use of the BOX statement causes the demonstration screen to display an informational box containing commentary about what is happening on the screen. Proper syntax for this command is:

```
(BOX)atrow,atcol,boxrows,boxcols
```

The starting row and column for the screen display of the box is indicated (atrow,atcol), followed by the size of the box (boxrows, boxcols). No other information should appear on the remainder of the line following the BOX special sequence. The information which is displayed in the box should appear in the script on the line following the BOX statement, and may consist of several lines of text.

For example:

```
(BOX)15,05,05,50
This is a test display of informational text
which appears on the screen.{}
```

This example would display a box on the screen starting at row 15, column 5, with a size of 5 rows by 50 columns as the dimensions of the box. As everywhere in NPL, screen lines and columns are numbered starting at 0, the first line inside the box is always blank (reserved for the top row of the actual box--refer to Section 12.5.2 below). Left-justified inside the box, the statement "This is a test ..." would appear on the second line, with the remainder of the script appearing on the third line.

NOTE: The {} at the end of the last line is the special sequence used to indicate the end of informational text. Refer to Section 12.5.3 below.

The last line within every box contains the centered message "Press SPACE BAR to continue." as its final instruction. Because the first and last lines of the box are "reserved" for the box itself, and the last line within the box is reserved for the "Press SPACE BAR ..." message, be sure to add 3 rows to the calculation for the depth of the box.

When the space bar is pressed, the box and its contents are erased and replaced with the correct previous contents of the screen, and the demonstration proceeds.

12.5.2 Special Sequences

The DEMO processor supports several special sequences which, when encountered in DEMO text, invoke special features of the DEMO processor. These special sequences should be located in the special text section of the DEMO script file. The delimiters for all special sequences are braces, "{ }" (HEX(7B7D)). Refer to Appendix D for the specific keystrokes used by the terminal to generate braces.

These special sequences are:

Tab Sequence - {T(decimal Tab value)}

This sequence causes text following the sequence to be displayed starting at the column specified by the tab value.

For example:

{T40} causes text following to be displayed starting at column 40.

HEX Sequence - {H(hex string)}

This sequence allows characters which are not normally enterable through the standard keyboard to be entered into DEMO text. All such characters must be included in the hex string. This feature is particularly useful for creating DEMO text which contains characters from the international character set.

NOTE: When the contents of the script file are displayed by the RunTime program, screen character translation as described in Section 7.4.3 does take place. Therefore, to display the international character set, the NPL codes must be entered in the hex string.

For example:

{H0202020E} enables use of the NPL alternate character set.

{H98} causes the brace character, "{", to be displayed from the NPL alternate character set.

Cursor Control Sequence - {@(row value,column value)}

This sequence positions the cursor at the screen location specified by the row and column values. This allows for printing pointers (e.g., "^" or "<--") to highlight important screen information which is outside the box. Such sequences should appear after all text which should remain inside the box display.

For example:

25 = {@5, 8} locates the cursor at row 5, column 8 on the screen.

NOTE: The brace characters should not be used for anything other than these special sequences. If it is necessary to display braces on the BOX text, use the HEX string special sequence as described above.

For example:

(BOX)15, 05, 05, 50
{T20}This is a test display of informational text
{H98}which appears on the screen.{}

The {T20} causes the text on the first line to be indented (TAB) 20 positions.

The {H98} displays a left brace ({) at the start of line 2.

12.5.3 End-of-Text Delimiter

Termination of informational text for the DEMO script is completed by entering the characters "{}" (open and closed braces) at the end of the informational text. Refer to Sections 12.5.1 and 12.5.2 for examples of the use of end-of-text delimiters.

12.5.4 Modifying Box Parameters

As a programming convenience, the parameters to the BOX statement may be interactively changed during execution of the DEMO script by use of the arrow keys.

When the informational box appears on the screen, the box may be repositioned to the left, right, up or down, simply by pressing the appropriate directional arrow on the keypad.

By pressing the shifted arrow keys, the size of the box may be changed through movement of the right border of the box to the left or right, or the lower border of the box up or down.

NOTE: If the size or position of the box is changed during the program, the parameters of the BOX statement in the DEMO script file are also dynamically changed to reflect the new position. Due to this dynamic changing of the DEMO script file, it is important that all parameters within the BOX statement should be expressed as 2-digit numbers (e.g., 05 for the fifth row, 09 for the ninth row). Otherwise, there may not be sufficient space in the DEMO script file for the updated box parameters to be rewritten.

12.5.5 BOX Graphics Display

The implementation of "true" box graphics requires a monitor that can support dual text and graphics mode. If such a monitor is present and is supported by the RunTime program, then "true" box graphics are displayed.

In all other cases, whether the monitor is supported but does not have dual text and graphics capabilities, or the monitor is simply not supported (generic screen handling used), "true" box graphics are not available.

Application programs may examine byte 4 of the $MACHINE system variable to determine if "true" box graphics are available on the machine currently being used. A value of "G" indicates that "true" box graphics are available. A value of " " indicates that they are not. For further details, refer to the Statements Guide, $MACHINE and $BOXTABLE.

If "true" box graphics are not available, use of the $BOXTABLE function can substitute "character" graphics instead. For further details, refer to the Statements Guide, $MACHINE and $BOXTABLE.

HINT: It is strongly recommended that software authors design their DEMO script files for use with character boxes and explicitly set byte one of $BOXTABLE during testing so that character boxes are always used, regardless of the "true" box capabilities. This ensures that the DEMO displays properly on all machines supported by NPL. However, note that the value of $BOXTABLE affects BOXes displayed by the application program as well as BOXes displayed by the script file. For applications which are dependent upon the features of "true" box graphics, this fact has to be considered.

12.5.6 Redefining the Prompt

Automatically appearing on the last line within every box is the message "Press SPACE BAR to continue." The message "prompt" may be changed by:

```
(MEMO)new prompt specification
```

This will allow for redefining the prompt to another message, or one that is printed in another language. Whatever message or prompt appears, the space bar is still required to continue the demonstration.

NOTE: This change is permanent for the duration of the RunTime session. Restarting a demonstration script does not reset the prompt to the startup value. No other information should appear on the line after a MEMO special sequence.

12.6 Adding Comments

As a part of the DEMO script, but not a displayable part of the demonstration, it is possible to insert comments or other informational text into the body of the script. Use the following syntax:

```
(REM)any commented information
```

This allows for the insertion of "documentation" within the actual script. This is never to be displayed on the screen during the live demonstration.

NOTE: All information to the end of the line is treated as a comment.

12.7 Restrictions

Several restrictions must be considered while using $DEMO. Among the restrictions considered are: the availability of a HELP key, the reliability of a polling KEYIN statement and the return value of the $IF ON statement. The following sections describe each of these issues.

12.7.1 The HELP Key

During the processing of the DEMO script ($DEMO), the HELP key is disabled. The demonstration may be terminated by pressing the CANCEL key any time an informational box is displayed on the screen. Exiting from the demonstration in this manner reverts to the normal operations of the program.

12.7.2 Polling KEYIN

Use of the polling form of the KEYIN statement may not operate with DEMO script files. If a polling form of KEYIN is encountered while a DEMO script is active, it appears to the program that no key is available, and execution continues to the next statement. If a program uses the polling form to accept keyboard information required by the demonstration, any such statements must be modified to use the non-polling form of KEYIN. For further details, refer to the Statements Guide, KEYIN.

12.7.3 $IF ON

$IF ON /001 (keyboard status check) always returns a value of OFF (no keystrokes available) during $DEMO.

12.8 Keyboard Logging

As of Release III, $DEMO script files may be created automatically by the NPL application by use of the keyboard logging feature. Keyboard logging allows the application to specify the name of a log file (using the $DEVICE statement) and then to start keyboard logging use of the SELECT LOG statement.

For example:

```
0010 $DEVICE(/219)="LOG.DAT"
0020 SELECT LOG /219
```

This causes automatic logging of keystrokes to the file LOG.DAT.

Files produced by keyboard logging may be used as input to $DEMO with no modifications. Special keys generate the special keywords expected by $DEMO as defined in Section 12.4.2.

NOTE: For actual demonstration purposes, it may be desirable to edit the log file and add various special sequences including BOX statements.

For further details refer to the NPL Statements Guide, SELECT LOG.

CHAPTER 13

NPL UTILITIES

13.1 Overview

Niakwa has supplied the following utility programs for use with NPL. They are provided at no charge as a convenience to the application developer.

NOTE: MS-DOS delimiters, device names and file-naming conventions are used for all examples throughout this chapter. Refer to the Chapter 5 of the appropriate NPL Supplement for delimiter, device names and file-naming conventions for the operating system being used.

Section 13.2 explains how to start NPL utilities.

Section 13.3 discusses the NPL utilities menu.

Section 13.4 discusses the Menu utility.

Section 13.5 explains how to format floppy disks.

Section 13.6 discusses the General Backup utility.

Section 13.7 discusses the General Recovery utility.

Section 13.8 discusses the General File Copy utility.

Section 13.9 discusses listing diskimages.

Section 13.10 discusses new diskimage creation.

Section 13.11 explains how to change diskimage size.

Section 13.12 explains how to scratch single files.

Section 13.13 explains how to change the Utilities Device Equivalence Table.

Section 13.14 discusses the Keyboard Translation Table editor.

Section 13.15 discusses the Screen Translation Tables editor.

Section 13.16 discusses the Font Table editor.

Section 13.17 discusses the Printer Control Table editor.

Section 13.18 discusses the Indexed HELP File Processor.

Section 13.19 discusses the Diagnostic Dump Variables list.

Section 13.20 discusses the Printer Translation Table.

Section 13.21 discusses the Edit Options utility.

Section 13.22 discusses the OTHER Program utility.

Section 13.23 explains how to generate 2200 source.

13.2 Starting the NPL Utilities

These utility programs are located in the \BASIC2C\UTILITY.BS2 diskimage (/D35). To run the utilities, enter the following at the MS-DOS prompt:

```
cd \BASIC2C

RTP UTILITY

(or)

RTI UTILITY
```

The first program loaded is the ATDEVICE utility, which allows users to set-up device equivalences for those diskimages they wish to manipulate during a session with the utility programs. Refer to Section 13.13 for details on the ATDEVICE program.

When the ATDEVICE program exits, the full utilities menu is displayed. This menu has been created using the Niakwa 2CMENU program. Refer to Section 13.4 below for details.

NOTE: In all utilities, the TAB key exits the utility and returns to the "START" program.

13.3 The NPL Utilities Menu

```
                           NPL Menu Program
      Select Option by SPACE, BACKSPACE, UP/DOWN Arrow Keys, or LETTER
                   Then press EXECUTE (RUN) to Proceed

        ■ Format Floppy Disks (ATFORMAT)
        . General Backup Utility (2CBCKP)
        . General Recovery Utility (2CRCVR)
        . General File Copy Utility (2CCOPY)
        . Diskimage Listing (LISTDCT)
        . New Diskimage Creation (SCRDISK)
        . Change Diskimage Size (MOVEEND)
        . Scratch Single Files (SCRATCH)
        . Change Utilities Device Equivalence Table (ATDEVICE)
        . Keyboard Translation Tables Editor (EDKEYBOA)
        . Screen Translation Tables Editor (EDSCREEN)
        . Font Table Editor (EDFONT)
        . Printer Control Table Editor (EDPRINTC)
        . Indexed Help File Processor (HLPINDEX)
        . Diagnostic Dump Variables List (DIAGDUMP)
        . Printer Translation Tables Editor (EDPRINTE)
        . Edit Options (EDOPTION)
        . Other Program (OTHER)
```

13.4 Menu Utility

This Main Menu allows access to all NPL Utility programs.

The following is a brief description of the utility program options listed on the Utilities Main Menu.

Format Floppy Disks

Formats diskettes for use as "raw" diskettes on the system.

General Backup Utility

Backs up full disks or selected files to multiple diskettes. It is intended to be suitable for porting programs and data from a 2200 to approved systems running under NPL.

General Recovery Utility

Recovers from backup diskettes created by General Backup utility.

General File Copy Utility

Copies selected files from one disk platter to another. Files are copied in standard catalog format.

Diskimage Listing

Lists the catalog of a specified disk address on the screen or printer.

New Diskimage Creation

SCRATCHes a disk to a size specified by the operator.

Change Diskimage Size

Increases or decreases the physical size of a diskimage file.

Scratch Single File

SCRATCHes a single file specified by the operator.

Change Utilities Device Equivalences Table

Allows the operator to modify, add, or delete entries in the device equivalence table.

Keyboard Translation Tables Editor

Allows keyboard translation files to be modified or created.

Screen Translation Tables Editor

Allows screen translation files to be modified or created.

Font Table Editor

Allows downloadable fonts files to be modified or created.

Printer Control Table Editor

Allows modifications and creation of a data file which contains printer control codes for the various printer control options.

Indexed Help Table Editor

Creates the index for indexed HELP files. Prior to executing this utility, the combined HELP files must have been set up. The program resolves all internal references within a HELP file, updates pointer values within the HELP file, and creates an index file which contains one entry for each reference name with its associated offset in bytes from the start of the HELP file.

Diagnostic Dump File Processor
Used to analyze the dump file, produced by the DIAG option in the HELP processor, and produce a variable listing, with values, of all NPL variables in memory at the time of the dump

Printer Translation Table
Allows translation table files to be modified and created.

Edit Options
Allows contents of the $OPTIONS variable to be examined and modified.

Other Program
Loads and executes an operator-specified program.

13.4.1 Operation

The keys defined in the table below are active for the Menu Program screen. Instructions are generally provided at the top of the screen. Once a utility is selected, the main view screen for that utility appears.

Key	Function
Space Bar	Moves acceptance block down
Backspace	Moves acceptance block up
Return	Moves acceptance block down
Up Arrow	Moves acceptance block up
Down Arrow	Moves acceptance block down
Letter Keys	Moves acceptance block to next option with matching letter
EXECUTE Key	Selects options

In environments where mouse support is available, the mouse functions as:

Mouse Function	Function
Left Button, Single Click	Move Acceptance block to item on same row as mouse cursor.
Left Button, Double Click	Execute item on same row as mouse cursor.

13.4.2 Menu Options Setup

Menu options are contained in NPL program(s). These programs must contain data statements starting at line 9500. No line number may be greater than 9800. Up to 18 data statements may be contained in a single program. Each data statement must have three fields:

Description — Up to 60 characters.

Disk address — Of the program to be loaded (up to three characters). Blank defaults to currently selected disk.

Program name — Up to 8 characters.

A source version of the default menu options program is provided. This program is 2CMNDATA.SRC in \BASIC2C on the NPL Development Package Utilities diskette. The compiled version of 2CMNDATA is in diskimage UTILITY.BS2.

13.4.3 Accessing the Menu Options Program

The 2CMENU program checks the contents of variable N$ for the name of the menu options program file to load. If N$ is blank then the default menu options file is used. This default is 2CMNDATA. Different menu options files can be used by setting up a short START program which defines N$ as a common variable, assigns the name of the menu options file to N$, and then LOADs 2CMENU.

For example:

```
10 REM START
20 COM N$8
30 N$="GLMENU"
40 LOAD T "2CMENU"
```

This program causes 2CMENU to execute with the menu options contained in program GLMENU.

NOTE: 2CMENU executes a COM CLEAR before LOADing the selected program.

No end of options indicator is required. Options are displayed until the first blank PROGRAM NAME is encountered or until the end of DATA statements. If there are fewer than 18 data statements in the specified program options file, no error results.

Linking of menus can be easily accomplished by specifying a program such as the START example above as the program to load in one of the data statements. This would cause 2CMENU to be reloaded but with a different menu options program file.

13.4.4 Error Handling

The NPL utilities perform standard error trapping. The following error conditions are detected and reported by the utility programs:

- Invalid disk address from which to load program.
- Disk platter specified not found.
- Program specified not found.
- File name specified is not a program (data file).

If one of these errors occur the utility program prompts for a valid entry. The utility programs will not execute until a valid entry is entered.

13.5 Format Floppy Disks

This utility formats diskettes for use as "raw" diskettes on the system. For information regarding supported devices refer to Chapter 5 of the appropriate NPL Supplement.

13.5.1 Features

- FORMATs the disk.
- SCRATCHes the disk.

13.5.2 Operation

```
                    Format Disk Utility
Key FN/TAB to exit.


Enter the address of the floppy drive to be used. D10
$DEVICE(/D10)="A: 1.2=Y"
Mount the diskette which is to be formatted in the floppy drive (/D10).
(If your GOLD KEY diskette is in the diskette drive, now is a really good time
to take it out.)  Enter Y to format
Enter number of index sectors required 10
Disk Scratched











                                                            CAPS LOCK
```

When first executed, the format utility asks for the NPL disk address to use. Next, the operator is prompted to mount the diskette to be formatted and reminded to remove the Gold Key diskette. The next step requires "Y" to continue. Then a $FORMAT DISK statement is executed for the selected disk address. After the format is complete, the operator is asked for the number of index sectors desired. A SCRATCH DISK statement with LS= the number of sectors supplied from the entered variable and END set to the appropriate value for the type of diskette being accessed is executed.

The prompt for the disk address to use accommodates the possibility of multiple diskette types used on the system. Device equivalence statements must be correctly set up for any NPL address used. Upon entry of the NPL address, the corresponding device is displayed for verification.

The FORMAT utility scratches the diskette after formatting. However, the catalogue end specification is determined by the program, based upon the device specification used. Catalogue end is set for the following "raw" diskette types as follows:

5 1/4"	320 K	1279 sectors
5 1/4"	360 K	1439 sectors
5 1/4"	720 K	2879 sectors
5 1/4"	1.2 MB	4799 sectors
3 1/2"	720 K	2879 sectors
3 1/2"	1.4 MB	5759 sectors
3 1/2"	2.88MB	11519 sectors

If the size of the diskette cannot be determined by the device specification used, the end catalog is set to 1279. The number of index sectors to use is still requested by the program.

NOTE: **Supported "raw" diskette formats are extremely operating system dependent (not all formats are supported on all environments). For information regarding supported "raw" diskettes and their device equivalences, refer to the Chapter "Supported Devices" of the appropriate NPL Supplement.**

13.6 General Backup Utility

This utility is a general backup utility for diskimages. It backs up full diskimages or selected files to multiple diskettes. It is intended to be suitable for porting programs and data from a 2200 to approved systems running under MS-DOS, and for use by end-users as a backup utility. The companion program, 2CRCVR, is required to recover files created by this utility.

NOTE: **All of the utilities supplied support any diskette type supported by NPL on the operating system in use. For porting from a Wang 2200, only the 360K and 320K format may be used. Note that the 360K format is supported on the Wang 2200 by use of the "PC INTERCHANGE" format. Refer to Chapter 15 of the Programmer's Guide for devices supported in porting to and from the Wang 2200.**

This utility (as well as 2CRCVR and 2CCOPY) has been modified to support extended (larger than 16MB) diskimage files. If extended diskimage files are being used, do not use any version of these utilities prior to Revision 2.01.

This version of the utility attempts to accommodate diskimages that are not larger than 16MB, but that have non-HEX(00) values in bytes 7 or 8 of sector zero. If bytes 7 or 8 are non HEX(00), but the actual sector number referred to cannot be read, the utility treats the disk as though bytes 7 and 8 were HEX(00). Please refer to Chapter 7 of the NPL Programmer's Guide for further details on extended diskimages and the use of bytes 7 and 8 of sector zero.

13.6.1 Features

- Full disk or selected file backup options are supported.
- Output may be stored on any diskette format supported by NPL on the operating system in use.
- Backup identification is requested from the operator and is stored on each output diskette.
- Diskettes need not be pre-formatted. The utility formats diskettes if required.

NOTE: The above feature may not be available for all diskette types on all systems.

- If errors are encountered on any output diskette, the program restarts at the beginning of that diskette. The entire backup does not have to be rerun from the start.

NOTE: Errors on the hard drive are fatal--the program must be rerun with bad area skipped (by using the selected file option).

- Data is verified on the input disk before copying, and on the output diskette after copying.

13.6.2 Operation

```
                         NPL Backup Utility

                    Please Select Option 1 or 2:  2

                         1) Full Disk Backup
                       2) Select Specific Files

             Please Enter Disk Address to Copy From:  D35
              Please Enter Disk Address to Copy To:  D10
      Please Enter Size (END=value) Sectors for Diskettes  4799
                        Backup Identification

                 Please Enter Operator Initials:  MWA
      Please Enter Description:  BACKUP EXAMPLE
                   Please Enter Backup ID#:  005308
                  Please Enter Date (MMDDYY):  072093

                          OK TO PROCEED?  Y




                                                        CAPS LOCK
```

The first screen of 2CBCKP requests the type of backup (full disk or selected file), input and output disk addresses, the size of the output diskette, and backup identification information. Before proceeding, the operator may review the information entered.

All entries which require specific responses are checked by the program. If a response is invalid, the bell sounds and the program does not continue until an acceptable response is given. Yes/No responses must be entered as upper case "Y" or "N". Disk addresses entered are checked for validity. No edit checking is performed on the backup identification information.

If the "Full Disk" option is selected, copying commences after the first screen. This is done using the COPY statements. Refer to Section 13.6.4 below.

The program can be terminated by pressing TAB/FN at any time until the actual copy process begins. After copying begins, the TAB/FN key is disabled.

13.6.3 Selecting Files

The option to back up selected files invokes a two-part file selection routine. The first part of the routine requests lower and upper range limits for file selection, and whether to copy program files, data files, or both. If range limits are left blank, all active files are selected in this first step.

NOTE: Scratched files are never selected for copying using the selected file option.

Following the file limit entries, the program scans the catalog of the input disk and selects all files falling within the selection criteria. These selected files are then displayed on the screen in alphabetical order, in groups of 60 files. For each screen displayed, the operator may select the specific files to be backed up by entering a numeric range (i.e., 1-20, 3-4, etc.).

NOTE: The hyphen, "-", is required except for single file selections. Files selected for copying are highlighted. The operator may change this selection by entering a range to change. This works like an on/off switch--the first time any file is included in a range, it is highlighted as selected for backup, the next time this file is included in a range, it is deselecte, the next time it is again selected, and so on.

When the range is left blank, the program continues on to the next screen. After the last screen has been displayed, the prompt "OK to Proceed" appears. If answered "N", the program loops back to the first screen of the secondary file selection process with files previously selected still highlighted. If "Y", then the backup process begins.

13.6.4 The Copying Process

Once all required information has been entered and the operator has given the OK to proceed, the actual copying takes place. The number of diskettes required is displayed on the screen along with status information throughout the process. The only prompts which should occur once the copying process has begun are related to mounting diskettes and reporting any error conditions.

The copy itself is an actual COPY statement. The program does not attempt to break up the range of sectors to be copied into multiple statements. Thus, when the actual copying is taking place, the program does not indicate how far along the copy is. In addition, during actual copying or verifying, the program cannot be interrupted by HELP.

When a new diskette is required, the program issues an appropriate prompt. When a new diskette is inserted, the program checks to see if it is formatted. If not, the program asks if the diskette is to be formatted. The program also checks that the diskette mounted is not part of the current set. If the diskette mounted is formatted and is not part of the current set, the program reports that the diskette appears to contain valid data and asks the operator to verify that it should be used.

If errors on the diskette are detected, the program so advises the operator and requests that a new diskette be mounted. Copying is restarted at the start of the diskette. If errors on the hard drive are detected, the operator is so advised and the program is terminated. To work-around hard drive errors, it is necessary to use the selected file backup option and to bypass the file(s) which are located in the sections of the hard drive with errors.

13.6.5 Limitations

The output diskettes are always scratched and created in an intermediate format. Each output diskette contains a file called INFO which contains the backup identification, and a file called DATAxx (where xx is the number of the diskette within the series) which contains backup data. The first diskette of a backup set contains a file called INDEX, which contains a listing of files backed up. In the case of a full disk backup, INDEX contains an exact replica of the index of the input disk.

NOTE: This means that this utility is not suitable for simply copying files from one diskimage to another. A formal general file copy program, 2CCOPY, has been included for this purpose. Refer to Section 13.8 below.

The SELECT FILE option assumes displayable file names.

If modified and recompiled, the REM $ compiler option must be set to "ON". If recompiling is done without the REM $ option ON, a message to that effect is given when an attempt is made to execute the program.

Both input and output platters are hogged for the duration of the backup. If a disk cannot be $OPENED, a message to that effect is displayed.

The SELECT FILE option can support a maximum of 500 files meeting the first level of selection criterion. Should there be more than 500 files, they may either be backed up in multiple runs of the programs using the first level of selection to limit the number of files backed up on any one run or, if the user has more than 28K, the program can be modified to allow more than 500 files.

To modify the program:

Change the DIM statements on line 11 for F$(), to the number of files needed.

NOTE: Each additional file requires 20 bytes of memory. Thus, in a 56K partition, over 1800 files can be backed up.

The utility reports that a diskette is not formatted if an attempt is made to access a diskette that is not in the drive. If this occurs, respond that the diskette is not to be formatted. The operator is then prompted to mount a new diskette.

13.7 General Recovery Utility

This utility is the companion to 2CBCKP. It restores data from backup diskettes created by 2CBCKP. It always expects to access diskettes which have been created in the intermediate format used by 2CBCKP. It always expects to recover to a fixed diskimage file (or 2200 platter). It also determines the size of the input diskette from information stored on the diskette during the original backup procedure.

NOTE: This utility (as well as 2CRCVR and 2CCOPY) has been modified to support extended (larger than 16MB) diskimage files. If using extended diskimage files, do not use any version of these utilities prior to Revision 2.01.

This version of the utility attempts to accommodate diskimages that are not larger than 16MB, but that have non HEX(00) values in bytes 7 or 8 of sector zero. If bytes 7 or 8 are non-HEX(00), but the actual sector number referred to cannot be read, the utility treats the disk as though bytes 7 and 8 were HEX (00). Please refer to Chapter 7 of the Programmer's Guide for further details on extended diskimages and the use of bytes 7 and 8 of sector zero.

13.7.1 Features

- Selected files or full disks can be recovered.
- Output disk is automatically scratched if full disk recover is selected.
- Option to overwrite existing files.

- The operator is informed of files that already exists if the option to overwrite in not selected.

13.7.2 Operation

```
                    NPL Recovery Utility

                   Select Files to Recover

                From Diskette D10 to Disk D35


                Recover All Active Files?  N

         Please Enter LOW Limit for File Selection
        Please Enter HIGH Limit for File Selection

     Copy Data (D) Files; Program (P) Files; or Both (B)  B
           Do you want to Overwrite Existing Files?  N





                     OK TO PROCEED?  Y

                                              CAPS LOCK
```

The operation of the utility is very similar to 2CBCKP as described above. There are a few differences:

The option to recover the entire disk is present only if the option to back up the entire disk was used to create the backup diskettes. The option to recover specific files is always present.

The recover SELECTED FILES option contains a provision for recovering all active files. If this option is used, the file selection routines are skipped and all active (non-scratched) files are recovered on a file-by-file basis. In addition, if this option is used, the operator is able to specify to scratch the output disk before proceeding.

NOTE: The output disk is always scratched on a FULL DISK recover, but is never scratched on a selected file recovery unless the recover all active files option is selected.

If recovering selected files, an option to specify whether or not to overwrite existing files is also present. If set to "N", files which already exist are not overwritten and the operator is advised of this on a file-by-file basis. If the overwrite option is set to "Y", the following errors may occur:

Wrong type. The file to be overwritten is a different type (program versus data) from the file being recovered. This file is skipped.

File to recover is smaller than file to be overwritten. The operator is advised of this situation and has the option of continuing or skipping this file. If continued, the file is overwritten and the trailer sector is also overwritten with the trailer sector of the recovered file.

File to recover is larger than file to overwrite. If sufficient space is available, the old file is renamed to JUNKXXXX and scratched. The recovered file is added as a new file. No operator prompt is displayed in this situation.

13.7.3 Error Handling

If a hard drive error is encountered, the program is aborted with an appropriate message to the operator.

If an error on a diskette is encountered during full-disk recovery, the program is aborted.

If an error on a diskette is encountered during selected file recovery, the operator is given the option of skipping that particular file or ending the program.

13.7.4 Limitations

The SELECT FILE option can support a maximum of 500 files meeting the first level of selection criterion. Should there be more than 500 files, they may either be backed up in multiple runs of the programs using the first level of selection to limit the number of files back up on any one run or, if the user has more than 28K, the program can be modified to allow more than 500 files.

To modify the program:

Change the DIM statements on line 10 for F$(), L$(), and W$() to the number of files needed.

NOTE: Each additional file requires 20 bytes of memory. Thus, in a 56K partition, over 1800 files can be recovered.

13.8 General File Copy Utility

This utility copies selected files from one disk platter to another. Files are copied in standard catalog format. Although the program supports copying to multiple diskettes as an option, no single file may be larger than the output diskette. That is, no "spanning" of diskettes is supported. If "spanning" is needed, use the general backup program 2CBCKP, with the select specific files option. The operator may specify whether existing files on the output disk are to be overwritten or not.

This utility is intended primarily for use by trained technical staff for porting and development. It does not provide the level of control required for use as a backup program in an end-user installation. End users should be advised to use the 2CBCKP general backup utility for backup purposes.

NOTE: This utility (as well as 2CBCKP and 2CRCVR) has been modified to support extended (larger than 16MB) diskimage files. If using extended diskimage files, do not use any version of these utilities prior to Revision 2.01.

This version of the utility attempts to accommodate diskimages that are not larger than 16MB, but that have non HEX(00) values in bytes 7 or 8 of sector zero. If bytes 7 or 8 are non HEX(00), but the actual sector number referred to cannot be read, the utility treats the disk as though bytes 7 and 8 were HEX(00). Please refer to Chapter 7 of the Programmer's Guide for further details on extended diskimages and use of the bytes 7 and 8 of sector zero.

13.8.1 Features

- Two levels of file selection.
- Ability to optionally overwrite existing files.
- Ability to copy onto multiple diskettes (but does not "span" any given file).
- Ability to optionally scratch output diskettes.

13.8.2 Operation

```
                    NPL File Copy Utility




          Please Enter Disk Address to Copy From:  D35
           Please Enter Disk Address to Copy To:  D10
              Will You Be Copying to Diskettes?  Y
   Please Enter Size (END=value) Sectors for Diskettes  4799







                     OK TO PROCEED?  Y

                                               CAPS LOCK
```

```
                    NPL File Copy Utility

                     Select Files to Copy

                   Copy All Active Files?  N

            Do you want to Scratch Output Disk?  Y
       Do You Want to use the new (') Hashing Method?  N

      Please Enter LOW Limit for File Selection
     Please Enter HIGH Limit for File Selection

    Copy Data (D) Files; Program (P) Files; or Both (B)  B



          WARNING Output Disk D10 will be SCRATCHED
   Please Enter Number of Index Sectors for Output Disk  10
 Please Enter Number of Sectors to Create Output Disk at  4799


                     OK TO PROCEED?  Y

                                               CAPS LOCK
```

The first screen of 2CCOPY requests the disk addresses to copy from and to. The program then allows the user to specify whether or not the output address is a diskette.

This option allows the program to copy files to multiple diskettes (although no single file may span diskettes). If set to "Y", the program gives the operator an opportunity to specify the size of the output diskette (default=1279). It then allows the operator to mount a new diskette if there is not sufficient room on the output disk for a file being copied, providing the size of the file to be copied is less than the specified diskette size.

The second screen requests the following information:

Copy All Active Files? Specifying "Y" causes the program to bypass the file selection routines and copy all active (non-scratched) files.

"Do You Want to Scratch the Output Disk" Answering "Y" causes the program to:

Request the number of INDEX sectors and CATALOG sectors at which to scratch the output diskette(s).

SCRATCH the output disk. If copying to diskettes, all output diskettes are scratched before copying to them.

If the output disk is not to be SCRATCHed, the program asks whether or not to OVERWRITE existing files. This option tells the program whether or not to replace files of the same name on the output platter with files to be copied or to skip (not copy) those files. Refer to Section 13.8.4 below for details on this feature.

Following this, if the option to "Copy All Active Files" has been set to "N", the file selection routines begins. Otherwise, the program proceeds directly to the copy phase (refer to Section 13.8.4 below).

13.8.3 Selecting Files

The actual file selection is a two-part routine. The first part of the routine requests lower and upper range limits for file selection, and whether to copy program files, data files, or both. If range limits are left blank, all active files are selected in this first step.

NOTE: Scratched files are never selected for copying.

Following the file limits entries, the program scans the catalog of the input disk and selects all files falling within the selection criteria. These selected files are then displayed on the screen in alphabetical order, in groups of 60 files. For each screen displayed, the operator may select the specific files to be backed up by entering a numeric range (1-20 or 3-4 or 2; etc.).

NOTE: **The hyphen, "-", is required (unless single files are specified). Files selected for copying are highlighted. The operator may change this selection by entering a range to change. This works like an on/off switch--the first time any file is included in a range, it is highlighted as selected for backup, the next time this file is included in a range, it is deselecte, the next time it is again selected, and so on.**

When the range is left blank, the program continues on to the next screen. After the last screen has been displayed, the prompt "OK to Proceed" appears. If answered "N", the program loops back to the start of the second level file selection routines with files previously selected still highlighted. If "Y", then the copy process begins.

13.8.4 The Copying Process

Copying is done on a file-by-file basis for all files selected. For each file, the input platter is verified, the file is copied, and the output platter is verified.

NOTE: **Verification is only done for the specific sectors copied from and to. Status information is maintained on the screen throughout the process.**

The output disk is always maintained in standard cataloged disk format. No intermediate format (such as used by 2CBCKP) is used. Each file copied to the output disk is cataloged normally. The input disk is never modified by this program.

Exceptions and errors are noted as they occur. Most exceptions have to do with insufficient space to create a new file, or various overwrite exceptions (refer below for details). Other errors, such as disk hardware errors, are reported as they occur. In these cases, the operator is given the option of quitting the program or skipping the file currently being copied and continuing the program.

Exceptions

Insufficient space for output file. This error is handled differently based upon whether the Copy to Diskettes option has been specified as "Y" or "N".

If not copying to diskettes, the operator simply has the choice of skipping this file or quitting the program.

If copying to diskettes, and the file being copied does not exceed the maximum size for copying to a diskette, the operator has the choice of skipping this file, quitting the program, or mounting a new diskette. If the size of the file exceeds the maximum, then the mount option is not given.

Overwrite Exceptions

If OVERWRITE has been specified as "N", the program notifies the operator of all files which are not copied due to the existence of a file of the same name on the output disk. The operator has to acknowledge this message.

If OVERWRITE has been specified as "Y", the following exceptions may occur:

Wrong type; the file to be overwritten is a different type (program versus data) from the file being recovered. This file is skipped.

File to copy is smaller than file to be overwritten. The operator is advised of this situation and has the option of continuing or of skipping this file. If continued, the file is overwritten and the trailer sector is also overwritten with the trailer sector of the recovered file.

File to copy is larger than file to overwrite. If sufficient space is available, old file is renamed to JUNKXXXX and scratched. Recovered file is added as a new file. No operator prompt is displayed in this situation.

13.9 Diskimage Listing

This utility lists the catalog of a specified disk address on the screen or printer. It is the equivalent of typing in a LISTDCT statement.

13.9.1 Features

- The program allows the user to specify whether the listing is to be in Index (I) order; Name (N) order; or Disk Location (S) order.
- Wildcard name selection is supported. The program lists only files matching a wildcard entry. An asterisk, "*", as the last character in the wildcard string indicates that all characters from the position of the "*" to the end of the filename are accepted. A question mark, "?" in any position of the wildcard string indicates that any character in that position in the filename is accepted.

For example:

*______ lists all files

GL*___ lists all files with the first two characters of the filename equal to "GL".

?L*____ lists all files with the second character of the filename equal to "L".

13.9.2 Operation

```
Disk Listing Utility

Key FN/TAB to exit.

Disk address D35
Order (I=index,N=name,S=start) I
Wildcard search (? matches any, * at end ignores rest) *
Printer address (005=screen, 215=printer) 005
INDEX SECTORS =        4
CURRENT END   =      520
END CATALOG   =      520

FILE     TYPE    START      END     USED     FREE    DATE     TIME
LISTDCT   P          4       16       13        0  92/12/22 11:32:58
@DEVICES  D         17       23        7        0  93/06/08 15:52:14
@DEVICE   P         24       31        8        0  92/12/22 11:32:50
DIAGDUMP  P         32       69       38        0  92/12/22 11:32:52
EDPRINTC  P         70       85       16        0  92/12/22 11:32:55
MOVEFILE  P         86      100       15        0  92/12/22 11:32:59
OTHER     P        101      103        3        0  92/12/22 11:32:59
SCRATCH   P        104      107        4        0  92/12/22 11:32:59
START     P        108      109        2        0  92/12/22 11:33:00
UTSTART1  P        110      113        4        0  92/12/22 11:33:00

                                                              CAPS LOCK
```

When the utility executes, the operator is asked for the address of the disk to be listed. If an invalid address is supplied, the program prints "Disk Error #xx" and asks for the address again. The operator is then asked for the desired printer address (e.g., /005, /215, etc.). The diskimage listing utility then performs the appropriate $OPEN and $CLOSE statements on all devices accessed. A listing of the disk catalog on the specified output device (output to the screen stops at the end of each page and request a keystroke to continue) is produced. When complete, the program loops back to the first step and requests another disk address.

13.10 New Diskimage Creation

Under NPL, new diskimage files are created by the SCRATCH DISK command. The physical size of the file, as well as the logical catalog end, is determined by the END parameter.

This program SCRATCHes a disk specified by the operator to a size specified by the operator.

13.10.1 Features

- SCRATCHes a disk.
- Inserts number of index sectors and end of catalog.
- Informs of successful/unsuccessful SCRATCH.

13.10.2 Operation

```
                    Scratch Disk Utility

Key FN/TAB to exit.
Enter address of Disk to be Scratched  D10
$DEVICE(/D10)="A:FLOPPY.BS2"
Enter number of index sectors required 10
Enter required end of catalog 1023
Use alternate hashing method (SCRATCH DISK ') N















                                                            CAPS LOCK
```

When the program is executed the operator is asked for the address of the disk to be scratched. If an invalid address is supplied, the program prints "Disk Error #xx" and asks for the address again. Next, the number of index sectors desired and the required end of catalog need to be entered. Then the program executes a SCRATCH DISK statement with LS= the number supplied above and END= the number supplied. A "Disk Scratched" message prints if no errors were encountered or "Scratch Error xx" is displayed and asks for the index sectors and end catalog parameter again. The most frequent error is that there is insufficient room on the hard drive for the size specified. When completed, the program loops back to the first step and requests the address of the disk to be scratched.

NOTE: **On some operating systems, the SCRATCH DISK statement actually writes to every sector. This makes operation of this utility somewhat slow on these operating systems. Refer to Chapter 5 of the appropriate NPL Supplement for further details on issues relating to creation of diskimage files.**

13.11 Change Diskimage Size

Under NPL, the physical size of diskimage files can be altered dynamically. One statement which does this is the MOVE END statement, which is executed against a disk address specified by the operator.

13.11.1 Features

- Increase/decrease physical size of a diskimage.
- Informs if operation was successful/unsuccessful.

13.11.2 Operation

```
                    Move End of Disk Utility

Key TAB/FN to exit.
Disk Address D35
INDEX SECTORS =       4
CURRENT END   =     520
END CATALOG   =     530
Enter New END CATALOG value










                                                            CAPS LOCK
```

When the program is executed, the operator is asked for the address of the diskimage to be changed. If an invalid address is supplied, the program prints "Disk Error #xx" and asks for the address to be reentered. The current INDEX SECTORS, CURRENT END, and END CATALOG of the specified disk prints. Next, the program prompts for the new END CATALOG value. A MOVE END statement using the specified disk address and new END CATALOG AREA is executed. If no errors were encountered, "done" is displayed or "Move End Error xx" is displayed. After completion, the program loops back to the first step and requests another disk address.

NOTE: Operation of the MOVE END statement is operating system dependent. Refer to Chapter 5 of the appropriate NPL Supplement for further details on issues relating to this command.

13.12 Scratch Single Files

This program SCRATCHes a single file specified by the operator.

13.12.1 Features

- SCRATCHes a file.
- Informs if operation was successful/unsuccessful.

13.12.2 Operation

When this program is run, it asks for the name of the file to be scratched and the address of the disk on which the file is located. If an invalid address is supplied, the program prints "Disk Error #xx" and asks for the address again. A SCRATCH T/(specified address)(specified filename) is executed. A "File Scratched" message is displayed if no errors were encountered or a "Scratch Error xx" message is displayed and asks for the name and disk address again. When completed the program loops back to the first step and requests the name of the file to be scratched.

13.13 Change Utilities Device Equivalences Table

This program allows the operator to modify the Device Equivalences Table maintained by the RunTime program.

NOTE: **Device equivalences are established each time the RunTime program is executed. The equivalences set up in this program are not automatically used if the RunTime is executed with any [progname] other than UTILITY. Therefore, setting up device equivalences here does not substitute for setting up device equivalences in the existing version of the BOOT.OBJ program.**

13.13.1 Features

- Displays device equivalences stored in a data file (ATDEVICES).
- Displays default device equivalences set up by the RunTime program.
- Allow the user to specify the device address to be changed.
- Change device equivalences.

NOTE: **Device equivalences set up by the $DEVICE statements in a modified startup program are not automatically displayed. Only device equivalences stored in the ATDEVICES data file and the default device equivalences are displayed.**

13.13.2 Operation

```
                          NPL Device Configuration

Press TAB/FN key to cancel

WARNING - the device equivalences set up here are ONLY valid for this
execution of RTP.
/D10 = A: 1.2=Y
/D11 = platter1.bs2
/D12 = platter2.bs2
/215 = /dev/prn
/204 = /dev/prn
/D35 = UTILITY.BS2







                 Enter Device Code to change (blank if done)



                                                  CAPS LOCK
```

Specifying an address already listed allows the user to modify the device equivalence for that address or delete that address from the table by specifying an equivalence of spaces. Specifying an address not currently in the table allows the user to add that address to the table.

The above step is repeated until the device address is left blank. At this point, the modified device table is set using the $DEVICE statements and the table is saved on disk in the data file ATDEVICES.

NOTE: If the program is exited using FN/TAB, the device table and the ATDEVICES data file is not updated.

Disk address /D35 has been set up as the diskimage UTILITY.BS2 on the NPL Utilities diskette provided by Niakwa. Be sure not to change this. If it is changed, the RunTime program generates a P48 error when this program is exited.

Please refer to Chapter 7 of the NPL Programmer's Guide for details on device equivalences, and the NPL Statements Guide for further details on the $DEVICE statement.

13.14 Keyboard Translation Tables Editor

Keyboard translation is accomplished from disk-based keyboard translation files. If, for any reason, the default values are not acceptable for the application being used, there is the option of modifying these files. Refer to Chapter 7 of the Programmer's Guide for an overview of keyboard handling under NPL.

Modification of the default keyboard translation table can be done dynamically through the use of the $KEYBOARD statement (refer to the Statements Guide, $KEYBOARD) or by setting up a disk file containing the modified table. The EDKEYBOA utility can be used to set up this disk file.

The RunTime program looks for a disk file whose name is based on the terminal type currently set up as being used. For example, when using a Wyse 60 terminal under SuperDOS, the RunTime looks for a file named KEYBOARD.WY6 when executed.

If the proper keyboard translation table file is found, the keyboard translation table is loaded from this file. If the file is not found, the RunTime uses built-in defaults. However, the built-in defaults may not be suitable for the terminal in use. Refer to Appendix D of the Programmer's Guide for information on default keyboard translation table assignments for specific terminals. Refer to the Chapter 6 of the appropriate NPL Supplement for keyboard translation table naming conventions.

13.14.1 Features

- Displays the contents of keyboard translation files.
- Allows modification of keyboard translation files.

13.14.2 Operation

```
                    NPL Keyboard Customization


The key which generates simple 45 currently corresponds to NPL key 45("E").
Enter 'Y' if the key should be treated as a Special FN key by NPL N
Enter the NPL key code to assign to the key 45
 Normal keys generated by NPL keyboards include:

 00 - Do not generate a key.
 08 - Backspace
 0D - Return Key
 82 - EXEC/RUN Key
 A1 - LOAD Key
 81 - CLEAR Key
 84 - CONTINUE Key
 E5 - ERASE Key (shifted on DW)
 A0 - PRINT



                                                   CAPS LOCK
```

When the EDKEYBOA program is being run, the user is prompted for the name of the file containing the translation table. The default name determined by EDKEYBOA is based on the terminal type and operating system in use. If the name is blanked, the currently loaded table (whether from the internal default table or a disk based table) is used instead.

The program first requests that the FN/TAB key be pressed (this information is necessary to ensure that the user can exit from the program using the keyboard). The program then asks whether the actual keyboard for which the table is being designed is being used. If "Y" is answered, it is not necessary to know what codes the keyboard is generating to set up the table. The program then asks the user to press the key to be redefined, and report the codes generated by it (if it does not report anything when the key is pressed, either the key does not generate a code or it generates special function HEX (E1) = (HELP) which cannot be redefined using this option).

If "N" is answered, the user must supply the hex code designation of each key to be redefined. "N" must also be answered if the user wishes to redefine the key that generates special key HEX(E1), since this key cannot be redefined by the first method. The program asks whether code generated by the key is "special" or not, and for the hex code designation of the key.

Once the user has defined the key by either of the above methods, the program reports the hex code that is currently assigned to that key (a code is reported as 2 hex digits, preceded by an apostrophe "'" if it is a special function key). The program then asks whether that key should be returned as a special function key to NPL applications or not. It then displays a table of NPL keyboard codes and asks for the key value to return to the NPL application. Keys which are defined as non-special and returning a value of HEX(00) are suppressed entirely. That is, no code is generated.

Repeat this operation for all keys which must be reassigned.

When all corrections are made, press the FN/TAB key. The program first checks that the following critical NPL keys have been defined:

EXECUTE
CANCEL
RETURN
HELP

If any of these keys has been left undefined, the user is informed of this by the program and given a chance to correct the problem before saving the file.

The system then presents a screen with the following options:

0 End Editing	This ends the keyboard editing session without saving any modifications.
1 Load table	This prompts for the name of another file containing a keyboard translation table with which to begin a different editing session.
2 Edit table	This begins the editing session of the keyboard translation table again.

3 Save table	This either saves a new keyboard translation table or overwrites an already existing table.
4 Print table	This prompts for a printer address and then prints the current keyboard table to the appropriate output address.
5 Use table	This automatically begins using the new keyboard translation table including any new modifications from the present editing session.

If, during a "Save table" operation, the table cannot be saved, a diagnostic message appears and the user may enter a different name.

NOTE: The RunTime program looks for the keyboard translation file in the current directory or in the NPL default directory. If it is desirable to have separate tables for different applications, the user may either have different copies of the keyboard translation file in different directories, or must rename or copy the file that is to be used when switching from one application to another.

Keyboard reassignment is *not* done in the normal course of running the program. To test the effect of the modified keyboard table, the table must be copied to the Keyboard Translation file name used by the RunTime and then the RunTime program must be rerun, or the "Use table" option can be selected.

13.15 Screen Translation Tables Editor

The RunTime program contains built-in defaults for mapping NPL characters to the appropriate display device. These default values are set up differently based upon the specific hardware version of the RunTime program being used. If, for any reason, the default values are not acceptable for the application being used, there is the option of modifying these default values.

Modification of the default screen translation table can be done dynamically through the use of the $SCREEN statement (refer to the Statements Guide, $SCREEN) or by setting up a disk file containing the modified table. The EDSCREEN utility can be used to set up this disk file.

The RunTime program looks for a disk file whose name is based on the terminal type that is currently set up as being used.

For example:

> When using a Wyse 60 monitor under SuperDOS, the RunTime looks for a file named SCREEN.WY60 when executed. If the file is not found, the RunTime uses built-in defaults. However, the built-in defaults may not be suitable for the terminal in use. Refer to Appendix D of the Programmer's Guide for information on default screen translation table assignments for specific terminals. Refer to Chapter 6 of the NPL Supplement for screen translation table naming conventions.

13.15.1 Features

- Displays the contents of screen translation files.
- Allows modification of screen translation files.

13.15.2 Operation

When EDSCREEN is run, the user is prompted for the name of the file which contains the translation table. The default name determined by EDSCREEN is based on the terminal type and operating system in use. If the name is blanked, the currently loaded table (which may be the standard built-in default), is used instead. If the editor is being used for first time, this file has not been created yet, so blank out the default.

The program then displays two character set tables. The one on the left hand side of the screen is the characters as they appear using the table as it is currently defined. The position in the table indicates which 2200 hex code generates the character. The one on the right hand side shows the characters which may be used for replacement.

```
      NPL CHARACTERS                        MS-DOS CHARACTERS
   0 1 2 3 4 5 6 7 8 9 A B C D E F       0 1 2 3 4 5 6 7 8 9 A B C D E F

0x|-------not displayable---------|  0x|  ☺ ☻ ♥ ♦ ♣ ♠ • ◘ ○ ◙ ♂ ♀ ♪ ♫ ☼|
1x|â ê î ô û ä ë ï ö ü à è ù Ä Ö Ü|  1x|► ◄ ↕ ‼ ¶ § ▬ ↨ ↑ ↓ → ← ∟ ↔ ▲ ▼|
2x|  ! " # $ % & ' ( ) * + , - . /|  2x|  ! " # $ % & ' ( ) * + , - . /|
3x|0 1 2 3 4 5 6 7 8 9 : ; < = > ?|  3x|0 1 2 3 4 5 6 7 8 9 : ; < = > ?|
4x|@ A B C D E F G H I J K L M N O|  4x|@ A B C D E F G H I J K L M N O|
5x|P Q R S T U V W X Y Z [ \ ] ↑ ←|  5x|P Q R S T U V W X Y Z [ \ ] ^ _|
6x|° a b c d e f g h i j k l m n o|  6x|` a b c d e f g h i j k l m n o|
7x|p q r s t u v w x y z § £ é ç ¢|  7x|p q r s t u v w x y z { ¦ } ~ ⌂|
8x|  ◆ ▶ ◀ → ∟ ¦ å ± ` ^ ■ ‼ ↕ ß ¶|  8x|Ç ü é â ä à å ç ê ë è ï î ì Ä Å|
9x|» æ ▲ ▼ ↓ ¬ √ ○ { } △ ~        |  9x|É æ Æ ô ö ò û ù ÿ Ö Ü ¢ £ ¥ ₧ ƒ|
Ax|      ─   ┐ ┌ ┬   ┘ └ ┴ │ ┤ ├ ┼|  Ax|á í ó ú ñ Ñ ª º ¿ ⌐ ¬ ½ ¼ ¡ « »|
Bx|                               |  Bx|░ ▒ ▓ │ ┤ ╡ ╢ ╖ ╕ ╣ ║ ╗ ╝ ╜ ╛ ┐|
Cx|  # # ▀ # # # # # # # # # # # ▀|  Cx|└ ┴ ┬ ├ ─ ┼ ╞ ╟ ╚ ╔ ╩ ╦ ╠ ═ ╬ ╧|
Dx|# # # # # ▌ # # # # # # # # # #|  Dx|╨ ╤ ╥ ╙ ╘ ╒ ╓ ╫ ╪ ┘ ┌ █ ▄ ▌ ▐ ▀|
Ex|# # # # # # # # # # ▌ # # # # #|  Ex|α ß Γ π Σ σ µ τ Φ Θ Ω δ ∞ ø ε ∩|
Fx|▄ # # # # # # # # # # # ▄ # # █|  Fx|≡ ± ≥ ≤ ⌠ ⌡ ÷ ≈ ° ∙ · √ ⁿ ² ■  |

Key TAB/FN to quit.
NPL Hexcode to change.
Use edit keys to move cursor.  Key EXEC to choose character.
Space/backspace will also move cursor.  Key FN/TAB or 0 when finished.
                                                       CAPS LOCK
```

To make a correction, position the cursor to the character which appears incorrectly on the left hand side, and press EXEC. The cursor moves to the character in the right hand side which is currently displayed for the selected NPL code. To change the character's display value, move the cursor to the table entry (on the right side) which shows the correct character, and press EXEC. The cursor moves back to the left hand side, and the display is updated to show the new status of the table. Repeat this operation for all characters whose appearance is incorrect.

When all corrections are made, press the FN/TAB key. The system asks whether or not to save the table as it is currently defined. If "N" is answered, any changes made to the table are not saved. If "Y" is answered, the system requests the name of the file which is to receive the table. The default is the same name that the table was loaded from, or, if this is a new table, the default name is determined by EDSCREEN based on the terminal type and operating system in use. Correct the displayed name if this is necessary and press RETURN. The selected file is overwritten with the new table value. If the table cannot be saved, a diagnostic message appears and the user may enter a different name.

NOTE: Upon initial start up, the RunTime program looks for the screen translation file in the current directory or in the NPL default directory. If it is desirable to have separate tables for different applications, the user may either have different copies of the screen translation file in different directories, or can rename or copy the file to be used when switching from one application to another.

Screen translation file reassignment is not done in the normal course of running the program. To see the effect of the modified screen table, the table must be copied to the screen translation filename used by the RunTime and then the RunTime must be rerun.

13.16 Font Table Editor

Mapping of the NPL character set to terminals which are capable of handling downloadable fonts is handled by a series of downloadable character font files. These downloadable fonts can be modified using the NPL Font Table Editor (EDFONT).

13.16.1 Features

- Make changes to the individual font table files (e.g., FONTAL5.TBL for Altos V terminals).
- Create actual downloadable font files (e.g., VTFONT.AL5 for the Altos V terminal).

Refer to Appendix D for information on terminals which support downloadable fonts, and Chapter 6 of the appropriate NPL Supplement for controllers that support downloadable fonts.

13.16.2 Operation

When EDFONT is run, the user is prompted for the type of fonts being modified. A prompt is displayed requesting the name of the file which contains that particular font table. The default name depends on the type of terminal or controller being used. If the name is blanked, the currently loaded table is used instead.

Next, a prompt is displayed for the name of a reference font file. A reference font file can be useful if it is desirable to copy previously defined font entries. Enter blanks if there is no need to use a reference font file.

The program then requests the NPL value of the character whose font design is to be changed. If the character is enterable from the keyboard, key it in and make the second character blank. If the character is not enterable from the keyboard, enter instead the hex values of the character.

```
                              NPL Font Editor
                       Target Device=IBM EGA graphics

Select an edit option or leave blank to select another code
0 = edit font, 1=replace from reference font
Hex Code = 45
Character= E        ................ <- TOP VISIBLE
                    ................
                    ........XXXXXXX. <- TOP OF CAPITALS
                    .........XX..XX.
                    .........XX...X.
                    .........XX.X...
                    .........XXXX...
                    .........XX.X...
                    .........XX...X.
                    .........XX..XX.
                    ........XXXXXXX. <- BOTTOM OF CAPITALS
                    ................
                    ................
                    ................ <- BOTTOM VISIBLE
                    ................
                    ................

COLUMNS VISIBLE ->           ↑L    R↑
                                                        CAPS LOCK
```

An enlarged version of the character is displayed for verification. The program then allows the user to select how the font entry is to be corrected. If the display format of the character (or one that is close) is available from any of the entries in the reference font file (FONT.TBL), a character from this font file may be selected by choosing option 1.

NOTE: Available font codes are dependent upon terminal type (e.g., the Altos V terminal only allows for editing font codes A1-FE).

If detailed editing of the entry is required, option 0 allows for moving the cursor around on the enlarged character, and turning characters on (using the "X" key) or off (using the "." key). When all necessary detailed editing corrections have been made to the font entry, pressing FN/TAB asks for a new edit option. When the displayed font entry is correct, enter blank as an edit option and a prompt is displayed for the next character which requires correction.

Repeat this operation for all characters whose appearance is incorrect.

When all corrections are made, press the FN/TAB key. The system asks whether or not to save the table as it is currently defined. If "N" is answered, any changes made to the table are not saved. If "Y" is answered, the system requests the name of the file which is to receive the table. The default is the same name that the table was originally loaded from.

The user is then asked if a downloadable file for this table should be created. If "N" is answered, then no downloadable file is created. If "Y" is answered, the system requests the name of the file which is to receive the downloadable fonts.

Different versions of the font tables and downloadable font files may be saved in different directories if it is desired to vary the fonts for different applications. When saving font tables or preparing downloadable font files, correct the displayable name if this is necessary and key RETURN. The selected file is overwritten with the new table value. If the table cannot be saved (or the downloadable font file cannot be created), a diagnostic message appears and the user may enter a different name.

NOTE: The RunTime automatically searches for the specific names, first in the current directory, and then in the BASIC2C directory.

Downloading of fonts is automatically performed by the RunTime for PC class monitors when downloadable fonts are supported. However, serial terminals require fonts to be downloaded manually. For information on which terminals and controllers support downloadable fonts, refer to Appendix D of the Programmer's Guide. Refer to Chapter 6 of the appropriate NPL Supplement for operating system specific information on downloading fonts.

13.17 Printer Control Table Editor

The RunTime program has the capability of sending printer control sequences to the printer using the Printer Control function of the HELP processor. Refer to Chapter 5 of the NPL Supplement(s) for details on this feature. The RunTime program has built-in defaults (different defaults depending on which hardware version of the RunTime program is being used) which should be adequate in most cases. For situations where the built-in defaults are not adequate, this utility allows the user to establish, or modify, a data file (PRINTCTL.TBL) which contains the printer control codes to be sent for the various printer control options.

Like all tables loadable by the RunTime program, RTP (or RTI) looks first in the current directory for PRINTCTL.TBL and then in the default NPL directory (e.g., /BASIC2C under MS-DOS). If the file is not found, then the built-in defaults are used.

13.17.1 Features

- Allows the operator to establish, or modify, a data file (PRINTCTL.TBL) which contains the printer control codes to be sent for the various printer control options.

13.17.2 Operation

The program first asks for the name of PRINTCTL.TBL file to modify. If executing this utility for the first time, this must be left blank.

The program then allows the user to define up to four values of characters for characters per inch, and four values for lines per inch. Both the literal numbers which appear on the printer control screen and the actual corresponding control sequences may be defined. In addition, the control sequences for form feed and line feed may be defined.

Following definition of the control sequences, the newly created, or modified, file may be saved. In this case, there is probably no need to have different PRINTCTL.TBL files for different NPL applications, so it is best to save the file in the NPL default directory so that all applications may access it.

13.18 Indexed HELP File Processor

This utility creates the index for indexed HELP files. Prior to executing this utility, the combined HELP files must have been set up. The program resolves all internal references within a HELP file, updates pointer values within the HELP file, and creates an Index file which contains one entry for each "Reference Name" with its associated offset in bytes from the start of the HELP file.

Please refer to the Chapter 11 of the Programmer's Guide for details on setting up indexed HELP files and for definitions of the terms used here. Please refer to files UTILHELP.HLP and UTILHELP.IDX in the Development Package for an example of an indexed help file with its associated index.

NOTE: This utility must be run whenever any change is made to an indexed HELP file. Otherwise, attempts to reference the file may result in incorrect portions of the file being displayed.

13.18.1 Features

- Creates the INDEX for indexed HELP files.
- Upon completion, the program allows the user to review the results by executing appropriate $HELP and $HELPINDEX statements.

13.18.2 Operation

When executed, this utility requests the name of the indexed HELP file to index. The program then searches through the specified HELP file, locates all ENTRY DEFINITIONS, calculates the relative byte number of the ENTRY DEFINITION, stores these calculated byte locations in the POINTER VALUES of the HELP file, and creates an index file (HELP file name.IDX) with a list of the HELPNAMES and locations.

Upon completion, the program allows the user to review the results by executing appropriate $HELP and $HELPINDEX statements. This is optional.

13.18.3 Error Handling

The following errors may be reported by the Indexed Help File Processor:

- **FATAL: Missing "}" near xxxxxxxxxxxxx**

Help sequences begin and end with Brace characters. Be sure all braces are matched in the file. The "xxxx" shows the part of the file where the problem was encountered.

- **FATAL: Too many forward references**

There is a limit to the number of Internal Pointers which may be outstanding at any time. The current limit is 3270 or limited only by physical memory on 32-bit operating environments.

- **FATAL: Too many reference points**

There is a limit to the number of Help Entry Definitions which may appear in a single index file. The current limit is 3270 or limited only by physical memory on 32-bit operating environments.

- **ERROR: Insufficient space for reference to xxxxxxxxxxx at byte nnnn**

The number of digits allocated for a pointer value is not enough to store the value. The file must be edited (e.g., change {=.000,xxx} to {.000000,xxx}) so that a larger pointer can be stored.

- **FATAL: Missing comma near xxxxxxxxxxxxx**

A comma must separate the "helpname" and Reference name on Help Entry Definitions. This is also the case for Pointer Values and Internal Labels in Help Internal Pointers.

- **Outstanding references:**
  At byte nnnnnn to xxxxxxxxxxxxx

This indicates that an Internal Label with a pointer variable does not refer to any Help Entry Definition in the same file. Check spelling and spacing of the Entry Definition for discrepancies.

- **Duplicate definition of helpname xxxxxxxx**
- **Duplicate definition of reference xxxxxxxxxxxxxxxxxxx**

This indicates that a Help entry definition appears more than once in the same file. Both the "helpname" and the Reference Names must be unique. Reference names are not case-sensitive and only the first 20 characters are inspected. A trailing "s" on reference name is dropped to allow the use of references to be both singular and plural where the topic is a noun such as toaster(s).

13.19 Diagnostic Dump Variables List

This utility is no longer supported under NPL Release IV.

13.20 Printer Translation Table

The Printer Translation Table option permits the automatic translation of characters as they are sent to a printer (or spool file) using a simple lookup table. This simplifies application programs which deal with non-English character sets and extended character sets when the type of printer to be used is not known.

The Printer Translation Table option is implemented by the use of the 256-byte system variable $PRINTER. An optional clause in the $DEVICE specification has also been implemented to specify that printer translation should take place. For exact details on the implementation of the $PRINTER system variable refer to Chapter 7 of the Programmer's Guide for details.

NOTE: Some devices, especially serial printers, may have difficulty in receiving characters outside the standard ASCII range (HEX(20) to HEX(7F)) unless the correct number of bits (8) and parity (none) are set.

- This type of translation does not allow for the possibility that some characters available on a printer can only be accessed by the use of a control code sequence. Such characters can only be accessed if provision has been made by the application software to send the control sequence.

The $PRINTER system variable can be loaded from a disk file when executing the Run-Time program. The RunTime looks for a file named PRINTER.TBL in the current directory. If the file is not found in the current directory, the RunTime looks in the default NPL directory. If this file is found, the printer translation table is loaded from the file.

13.20.1 Features

- Create a printer translation table.
- View contents of existing printer translation tables.
- Modify existing printer translation tables.

13.20.2 Operation

When EDPRINTE is run, the user is prompted for the name of the file which contains the translation table. The default name is "PRINTER.TBL". If the name is blanked, the currently loaded table (which may be the standard built-in default) is used instead. If using the editor for the first time, this file has not been created yet, so blank out the default.

The next prompt to appear inquires whether or not to print the untranslated character set for reference purposes. This printed report indicates the characters available on the printer being used. The program then displays the NPL character set (or as close to it as the terminal can display). The position in the table indicates which NPL hex code generates the character.

To make a change, position the cursor to the particular character which requires correction, and press the EXECUTE key. A prompt then appears asking which native operating system hexcode to use for the character when printer translation is in use. Refer to the printed report for the code which is appropriate for this printer. After entering the required Hexcode and pressing RETURN, a new hex value for the character is updated within the table.

NOTE: The table display does not reflect the particular change made, however. Repeat this operation for all characters whose print appearance is incorrect.

When all corrections are made, press the FN/TAB key. The system asks whether or not to save the table as it is currently defined. If "N" is answered, any changes made to the table are not saved. If "Y" is answered, the system requests the name of the file loaded. Correct the displayed name if this is necessary and press RETURN. The selected file is overwritten with the new table value. If the table cannot be saved, a diagnostic message appears and the user may enter a different name. The "P" option shows how close the printer is to the NPL character set.

NOTE: Upon initial start-up, the RunTime program looks for the printer translation table file in the current directory. If the file is not found in the current directory, it looks in the default NPL directory. If found, the printer translation table is loaded from this file. If separate tables for different applications are desired, there may be different copies of the printer translation file in different directories. Also remember that printer translation is performed only for print devices defined with the XLA=Y clause in the $DEVICE statement for that device.

13.21 Edit Options

Using an NPL program, the contents of the $OPTIONS variable may be examined and modified, to allow for particular manipulation of the various features available. Refer to the Statements Guide, $OPTIONS, for a detailed explanation of the $OPTIONS system variable and for a description of available options and default values. On-line help also has a short explanation of each option.

13.21.1 Features

- View contents of $OPTIONS variable.
- Modify the $OPTIONS variable.

13.21.2 Operation

```
                              Edit Options
 1.Underline colours (Blue/LWhite)  1F 17.132 column screen support       N
 2.Keyboard Lock State Display      Y  18.Extra 'normal' characters from  00
 3.Monitor noise suppression        Y  19.Extra 'normal' characters to    00
 4.PRINTUSING '$' replacement       $  20.'Raw' output in remote mode     N
 5.PRINTUSING ',' replacement       ,  21.Display 'bright' as 'dim'       N
 6.PRINTUSING '.' replacement       .  22.CGA normal colour (disabled)    00
 7.Complex Key Start value          00 23.CGA perim/und (Blue/Green)      12
 8.Terminal output type(IBM PC)     04 24.Graphic bright/blink            00
 9.Complex Key End value            00 25.EGA normal (Black/White)        07
10.Complex Key 2nd Start value      00 26.EGA bright (Black/*LWhite)      01
11.Complex Key 2nd End value        00 27.EGA blink (Black/Red)           04
12.Disable Reset and Step from HELP N  28.EGA bright/blink (*LWhite/Red)  14
13.Disable Halt Key                 N  29.EGA palette 0&1 (Black/LWhite)  0F
14.Implied $BREAK after disk access N  30.EGA palette 4&7 (Red/White)     47
15.HEX(A0-FF) font on VT100/Altos3  A  31.Emulator restrictions           00
16.Enable use of math coprocessor   N  32.Cursor style (default)          00
          Enter blanks to see other options or key FN/TAB to exit

                         Enter Option to change

                                                  CAPS LOCK
```

Selection of this utility program first displays each of the current default values for the $OPTIONS system variable, as set by the RunTime program. By selecting the item number for a particular option, and then entering an appropriate new value, the contents of $OPTIONS are modified.

NOTE: Any new values entered through this utility program are in effect only until the RunTime program is exited. Upon reentry to the RunTime, the original default values are back in place.

13.22 OTHER Program

This program loads and executes an operator-specified program from a specified disk address.

13.22.1 Features

- Programs may be executed without leaving the utilities program.

13.22.2 Operation

```
                    Run Other Program Utility

Key FN/TAB to exit.
Name of program to run PROG1
Disk Address D99














                                                          CAPS LOCK
```

This utility asks for the name of the desired program to execute and the address of the disk on which the program is located. If the program is not found, or other error conditions are present, the message "program not found" is printed and the program prompts for the name and location of the program again. Else, the specified program is executed.

NOTE: The program executes a SELECT DISK statement to the disk specified before loading the program specified.

13.23 Generating 2200 Source

2200 source format versions of the general backup (2CBCKP), general recovery (2CRCVR), and general file copy (2CCOPY) utilities can be generated through the NPL Compiler for ease in porting from a Wang 2200 to an IBM PC. These compiled programs are contained in diskimage file UTILITY.BS2 on the NPL Development Utilities diskette. This diskimage file is defaulted to address D35 in the NPL utilities menu. To move the utility programs from the IBM PC to the 2200 requires that the programs first be compiled into 2200 atomized format (specifying the output as A:, a raw 320k format diskette), after which the diskette may be used directly on the 2200 for the copying procedure. Please refer to Chapter 15 of the Programmer's Guide for exact defaults on the specifications to compile these programs, and porting in general.

NOTE: Program 2CCOPY contains a SCRATCH DISK ' statement at line 3075. If this program is operated on a 2200 with an operating system previous to 2.5, this statement must be commented out using the REM statement. Of course, when operating the program, do not scratch the output disk with the ALTERNATE hashing method. This is a feature of operating system 2.5 and greater.

Also, if programs 2CCOPY and 2CRCVR are modified and recompiled, the compiler generates warnings about SAVE statements. These warnings can be ignored since the SAVE statements are used only to create catalog entries for files being copied.

CHAPTER 14

COMPILER OPERATION

14.1 Overview

The NPL Compiler is a multi-purpose program whose purpose is to provide a mechanism for converting NPL program code from one form to another. Three forms of program code are supported:

1. Niakwa NPL executable format (p-code)

2. ASCII format

3. Wang 2200 format (atomized)

The conversion between any of these three forms is possible by the correct selection of the available compiler options.

The compiler is intended as a utility to perform convenient conversion of NPL programs between the above-supported formats on a "batch basis". It is not intended for use in the development or debugging of programs, for which use of the Niakwa Interpreter (RTI) is intended. Some practical uses of the Compiler are:

- Initial porting of programs from a Wang 2200 to a supported NPL environment.
- Compression of programs for distribution to end users.
- Creating Wang 2200 atomized code from p-code for porting back to a Wang 2200.
- Creating ASCII format program files from internal p-code format files for manipulation by MS-DOS operating system utilities.

This chapter discusses all facets of the compiler, and provides several examples of typical compilations using different combinations of the compiler options. In addition, refer to Chapter 9 of the appropriate NPL Supplement for operational features of the compiler that are operating system specific.

Section 14.2 discusses the compiler command line, the "shorthand" method of invoking the compiler.

Section 14.3 discusses the actual selection of input programs which are to be submitted to the compiler for compilation.

Section 14.4 discusses the general form and uses of the various compiler options which are available.

Sections 14.5 through 14.18 discuss in detail each compiler option and its default setting.

Section 14.19 discusses the user-friendly compiler display screen, an alternative to the compiler command line.

Section 14.20 discusses the use of batch files, including those supplied by Niakwa for quick compilation of programs using some common compiler options.

Section 14.21 discusses the issue of Wang 2200 compatibility when generating Wang 2200 format (atomized) code through the compiler.

Section 14.22 discusses the compatibility issue relating to programs compiled under earlier revisions of the RunTime.

Section 14.23 provides several examples indicating typical compilations using different combinations of compiler options.

14.2 The Compiler Command Line

Generally, the compiler is operated under the native operating system's command processor. When using the command processor, the general form of the compiler command line is as follows:

14.2.1 Format

```
B2C [option [option]...] srcprogs [[option poption]...]srcprogs..
```

where:

```
option        ={ /SRCLOC   [drive:][\pathname][\diskimage] }
                { /OBJLOC  [drive:][\pathname][\diskimage]  }
                { /OBJFORMAT  {SCRAMBLED, UNSCRAMBLED}      }
                { /OBJEXTRA  sectors                        }
                { /LSTLOC  [drive:][\pathname][\filename].  }
                {/LSTFORMAT  {.SRC, .LST, 2200, 2200S}      }
                { /ERRLOC [drive:][\pathname][\filename].   }
                { /ERRLOC [drive:][\pathname][\filename].   }
                { /WARNINGS {ON, OFF}                       }
                { /NUMBERS           {ON, OFF}              }
                { /DISPLAY           {ON, OFF}              }
                { /REM$              {ON, OFF}              }
                { /KEEPREMS          {ON, OFF, DEC}         }
                { /TRANSLATE hexquad{{.]hexquad]...         }
                { /LCASE             {ON, OFF}              }
```

where:

```
drive     = a native operating system drive des-
            ignation.

pathname  = a native operating system path name.

diskimage = a native operating system filename
            containing a diskimage file.

filename  = a native operating system filename
            and optional extension.

hexquad   = 4 hexdigits for from/to filename
            translation.

sectors   =  an unsigned integer where 0<= sec-
            tors<=65535.

srcprogs  = a list of input program names to be
            compiled, each separated by at
            least one space. An input program
            name may be either a literal prog-
            name or a wildcard.
```

where:

```
Progname  = an up to 8 character string program
            name. May not contain the '?' and
            '*' characters.

Wildcard  = an up to 8 character string contain-
            ing the special wild card charac-
            ters '*' and/or '?'.
```

where:

```
'?' matches any one character of a program name
'*' matches any characters up to the end of a program name.
The '*' should only appear at the end of a wildcard.
```

NOTE: The options specified on the compiler command line may be entered as either upper or lower case. For consistency, in the discussion in this chapter, options are always displayed as upper case.

14.2.2 Discussion

The designation "B2C" at the beginning of the native operating system command line, refers to the name of the NPL Compiler, and causes the operating system to invoke it. Upon execution, the compiler inspects the balance of the command line to determine the desired compiler options and the list of input programs to compile.

Compiler options are a list of parameters supplied by the programmer to control the behavior of the compiler. With these options, the programmer specifies such things as where the input programs are stored, where the resultant object (or other form) programs are to be stored, whether a listing is desired, and so on. If no options are specified in the command line, the compiler assumes that:

1. The input programs to be compiled are in the current directory as ASCII files with an extension of ".SRC".
2. The resultant object programs are to be stored in the same directory with an extension of ".OBJ" in UNSCRAMBLED format.
3. No source listing is to be produced.
4. Errors are to be printed to the screen.
5. Compiler warnings are to be provided.
6. All other miscellaneous options are OFF.

These are the default compiler options, any one of which may be changed by explicitly specifying the desired option on the command line. A detailed discussion of each compiler option is discussed following Section 14.4.

Following the compiler options, the programmer specifies the input programs to be compiled. The NPL Compiler may compile one or more programs in a single compiler run. In addition to specifying literal program names to compile, the programmer may also specify wildcard patterns which select a group of programs to be compiled based upon a wildcard search of the specified diskimage or directory.

14.3 Selecting Input Programs for Compilation

After the options area on the command line, list the name of each input program to be compiled, each separated by at least one space. As many programs as required may be listed, up to the maximum length of the native operating system command line.

For example, the maximum command line length under MS-DOS is 127 characters.

14.3.1 Literal Program Selection

A literal program name may be up to eight characters in length (more on program names below) and must not contain the "*" or "?" characters (these are wildcard characters). When specifying the input programs for compilation, one need only be concerned with the actual 8-character names of the program. All other considerations such as program location, format, required filename extensions, etc., are all handled by the SRCLOC option.

Example 1:

```
B2C PROG1 PROG2
```

In this example, the two programs "PROG1" and "PROG2" are compiled, in the given order. Since no options were specified, the compiler assumes default options. That is, the compiler searches the currently selected directory for two ASCII files with the names "PROG1.SRC" and "PROG2.SRC". The ".SRC" extensions are added by the compiler automatically. The resultant object files "PROG1.OBJ" and "PROG2.OBJ", created by the compilation, are stored in the same directory.

Example 2:

```
B2C /SRCLOC \source\2200disk.bs2 PROG1 PROG2
```

where:

"disk.bs2" is a diskimage

In this example, the programmer specified that the input programs are in a diskimage file called "disk.bs2" in the \source directory. The compiler scans the index of the diskimage for the programs "PROG1" and "PROG2". Since the programs are in a diskimage, no extensions are added to the program names. Since no OBJLOC option was specified, the resultant object programs are stored in the currently selected directory as ".OBJ" files.

14.3.2 Wildcard Program Selection

A second and more general method of selecting input programs for compilation exists. In addition to single program names, the compiler accepts "wildcard" names. The presence of a wildcard name in the input program list causes the compiler to compile all programs in the specified directory or diskimage (as specified with SRCLOC) that "match" the wildcard pattern. A wildcard name is any program name which contains at least one of the "?" or "*" wildcard characters.

A "?" in a wildcard matches any single character in the same position of the input program name.

A "*" in a wildcard matches any characters and no characters to the end of the program name.

For example:

"AR?UPD" matches "AR1UPD", "ARXUPD", "AR@UPD", but not "AR1UPD2".

"AR*" matches all file names beginning with "AR".

```
B2C /SRCLOC A: /OBJLOC platter1.bs2 *
```

compiles all programs on the 320K format diskette in drive A, placing compiled output in diskimage file PLATTER1.BS2 in the current directory.

```
B2C /SRCLOC A: /OBJLOC platter1.bs2 AR??001
```

compiles all programs with the first two characters "AR" and characters 5-7 as "001" from the 320k format diskette in drive A to diskimage file PLATTER1.BS2.

When wildcard scanning a directory, the scanner automatically adds the .SRC extension. When wildcard scanning a diskimage, the scanner ignores data files even if they match the wildcard pattern.

14.4 Compiler Options

The compiler accepts program input in three different formats--Wang 2200 Basic-2 atomized format, ASCII format, and compiled NPL p-code. Refer to Section 14.5 below for more details on input formats. The 2200 format recognition allows for the direct compilation of programs transferred from the Wang 2200. Once compiled, the compiler can optionally produce 2200 atomized source files which may then be ported directly back to the 2200 for use.

To facilitate software conversion and full system implementation, the compiler is designed to operate on multiple input programs in a single run. This capability allows the programmer to compile an entire application system library in a single compiler run.

During the compilation process, the compiler produces various formats of source listing reports and error reports. Through the use of compiler options, these reports may be printed directly to the system printer or spooled to disk for later review.

The compiler does allow for application software security from the implementation of optional object code encryption.

Since the compiler can read "source" (input) programs in a number of different formats, and can produce output of various kinds, the "options" entries are used to specify the exact options desired in a given compiler run.

Each option to the compiler starts with the switch character / ("/" is used in the examples throughout this chapter--refer to Section 4.2.1 of the appropriate NPL Supplement for the appropriate switch character to use on the system). The switch character is then followed immediately by a keyword which specifies the option, followed by a space and then a single word which gives the information required by that option.

For example:

```
/OBJFORMAT SCRAMBLED
```

OBJFORMAT indicates the "object format" option, "SCRAMBLED" is the parameter for that option. Wang 2200 programmers, in particular, must take care to place spaces between options and the keywords that follow them.

The options then specified on the command line are in effect for ALL input programs listed in the immediately following "srcprogs" list.

14.4.1 Default Options

If a compiler option is not explicitly specified in a command line, the compiler assumes a default value. The option defaults are explained in the documentation of each compiler option following this section.

14.4.2 Changing Options During a Compiler Run

It is possible to change a given option or options during a single compiler run. This is done by simply specifying another set of options followed by another srcprogs list.

For example:

```
B2C /OBJFORMAT SCRAMBLED PROG1 PROG2 /OBJFORMAT UNSCRAMBLED PROG3
```

This command line causes input programs "PROG1" and "PROG2" to be compiled and saved with SCRAMBLE protect, however, "PROG3" is compiled and saved UNSCRAMBLED.

NOTE: A compiler option retains its setting throughout the entire command line until it is explicitly modified in a subsequent options specification.

14.4.3 Specifying Path Names and Device Names

Many of the compiler options require specification of a native operating system pathname, "raw" diskette name or print-device name. Valid pathnames and device names vary widely from one operating system to another. Refer to Chapter 5 of the appropriate NPL Supplements for pathname and device-name usage on the operating system being used.

14.5 The SRCLOC Option

The SRCLOC option specifies the location of the input programs to be compiled. The general form for this option is as follows:

14.5.1 Format

```
     /SRCLOC [pathname][diskimage]
where:
     pathname  = the native operating system path-
                 name containing .SRC format input
                 programs or a diskimage.

     diskimage = the filename containing a diskimage.
default:
     /SRCLOC current-directory  (refer to 14.5.4 below)
```

14.5.2 Discussion

The /SRCLOC option is used to specify the location of the input programs to be compiled. Currently, NPL input programs may be located in one of three "containers" on the system. The "container" implicitly indicates the format of the input program. These "containers" are discussed in the following sections.

14.5.3 Diskimage Files

1. In a diskimage file containing either 2200 atomized format programs, or already compiled programs.

Diskimages are files which contain an internal structure identical to that of a Wang 2200 catalog (refer to Section 7.3 for details). If the SRCLOC is specified as a diskimage, the compiler scans the index area of the diskimage for the specified input programs or wildcards to compile.

It is also possible to specify the input location as a diskimage file containing already compiled NPL programs. This is useful for creating 2200 atomized programs as described in Section 14.9 below.

Programs which were previously compiled from source code directly into a diskimage file can now be recompiled directly from that diskimage file, without the need for the original source code.

Simply enter the name of the diskimage containing the compiled programs as the SRCLOC.

The compiler determines whether it is accessing 2200 atomized programs or compiled NPL programs by examining the header sector of each program.

Programs which are SCRAMBLE protected cannot be compiled.

2. Native Disk Environment

If the SRCLOC is specified as a directory within the native operating system, the compiler expects the input programs to be regular files of the native operating system in that directory. Each file contains one input program only, in ASCII format, and must have an extension of ".SRC".

This is the default SRCLOC if none is specified, in which case, the pathname is assumed to be the current directory.

3. "RAW" Diskettes

Diskettes with a standard 2200 catalog structure are directly readable by the compiler on some hardware versions of NPL. This type of diskette is called a "raw" diskette as it contains no native operating system file stricture whatsoever. Refer to the chapter "Supported Devices:" of the NPL Supplement(s) for information on "raw" diskette support on the operating system being used.

For example, under MS-DOS, SRCLOC can be specified as:

```
/SRCLOC A:
```

The "A:" drive designation is treated by the compiler (and RunTime program) as a special case meaning "the floppy drive" in 320K format. As in the case of diskimages, the compiler scans the index of the diskette for all input programs (or wildcard patterns) to be compiled.

This is the normal input format for programs being converted to a PC under MS-DOS when a 2275 diskette drive is available on the 2200 system and has been used to produce a 320K "raw" format diskette.

14.6 The OBJLOC Option

The OBJLOC option is used to specify the storage location of the resultant programs of the compile. the general form of this option is as follows.

14.6.1 Format

```
     /OBJLOC [pathname][diskimage]
where:
     pathname  = the native operating system path-
                 name to contain the resultant ob-
                 ject programs directly or that
                 contains a diskimage file.

     diskimage = the name of the diskimage to con-
                 tain the resultant object pro-
                 grams.
default:
     /OBJLOC current-directory (refer to 14.6.4 below)
```

14.6.2 Discussion

The OBJLOC option is used to specify the desired storage location of the resultant object programs of a compile. Only NPL compiled code is produced as the output stored in the OBJLOC location. The compiler may store the object programs in one of two types of locations. These locations are discussed in the following sections.

14.6.3 Diskimage Files

This is the most common OBJLOC. In this case, the compiler stores the object programs directly into the specified diskimage. The compiler catalogs the name of each program (uses the input program name where possible, refer to Section 14.17--The TRANSLATE Option) in the index area of the diskimage and stores the object code in the catalog area as a program file, much like a Wang 2200 except that the compiler is saving object code.

If a program file already exists in the diskimage with the same name, the compiler overwrites the old program file with the new program. If the new program exceeds the available space allotted the old program, the compiler allocates space from the free space area of the diskimage, if available. The START and END pointers of the old program are adjusted by the compiler to point to the newly allocated space. The programmer is notified by the message:

"Object program was moved to end of object diskimage."

NOTE: In this event, the catalog space occupied by the old program is not represented in the catalog index and is therefore inaccessible. The only way to recover this space is to reorganize the disk (e.g., MOVE DISK).

If, during this relocation operation, the compiler determines that there is insufficient free space in the diskimage to store the new program, the compile terminates with the message:

"Out of space in object diskimage"

and the content of the old program file is unpredictable.

NOTE: The compiler fully recognizes and maintains the structure of a diskimage during its compilation activities. This includes the newly supported extended diskimage files (16MB). Refer to Section 7.3.10 for details on extended diskimages. It is therefore quite acceptable to have data files in an OBJLOC diskimage.

The compiler does not create a diskimage if it does not already exist. The programmer may create a diskimage file in a number of ways. Refer to Section 13.10 of this guide for details on creating a diskimage file.

WARNING--It is possible to specify a single diskimage as both the SRCLOC and OBJLOC. If this is done, the compiler attempts to compile each input program, reading the source code from the catalog and writing the object to the same location. Although this works correctly in some limited cases, it is unreliable and should be avoided.

14.6.4 Native Disk Environment

If the OBJLOC is specified as a directory within the native operating system, the compiler stores all resultant object programs as files of the native operating system on that directory, one file per program. The filename is derived from the input program name (where possible, refer to Section 14.17--the TRANSLATE Option) with an extension of ".OBJ". Object files of this type can be executed directly by the RunTime program as stand-alone or bootstrap programs.

NOTE: **This form of OBJLOC is the default if none is specified. The directory is the current directory.**

14.6.5 No Object Programs Produced

The OBJLOC can be specified as the null device to suppress generation of compiled object programs (refer to Chapter 5 of the appropriate NPL Supplement for the device name to use as the null device on the operating system being used). This may be desirable if the compiler is being used to convert between types of files (.SRC from 2200 atomized, 2200 atomized from compiled, etc.) using the LSTLOC option, or to perform a preliminary syntax check without creating object programs. Refer to the discussion on the LSTLOC option in Section 14.9 for more details.

14.6.6 Scrambled Program Considerations

To provide a level of application program security, NPL object code may be stored in SCRAMBLED format. If this option is selected, the compiler encrypts the resultant object code of a compile before saving to the specified OBJLOC. For details on the option, refer to Section 14.7.3.

14.7 The OBJFORMAT Option

The OBJFORMAT option is used to encrypt the resultant program of a compile. The general form of this option is as follows:

14.7.1 Format

```
/OBJFORMAT {SCRAMBLED, UNSCRAMBLED}
```

default:

```
/OBJFORMAT UNSCRAMBLED
```

14.7.2 Discussion

To provide a level of application program security, NPL object code may be stored in SCRAMBLED format. If this option is selected, the compiler encrypts the resultant object code of a compile before saving to the specified OBJLOC.

14.7.3 Scrambled Program Considerations

When a SCRAMBLED program is loaded for execution, the RunTime program disables the program DUMP option of the HELP display (refer to Section 2.7.2 for a discussion on HELP Options). When executing under the Interpreter, program LIST and SAVE functions are disabled when a SCRAMBLED program is loaded. This option remains disabled until the SCRAMBLED program is fully overlaid.

NOTE: **Although the method of encryption used discourages unauthorized tampering of application object code, Niakwa does not guarantee 100% security from determined efforts.**

14.8 The OBJEXTRA Option

The OBJEXTRA option causes the compiler to allocate extra free sectors to an object program when creating a new program file in a diskimage. The general form of this option is as follows:

14.8.1 Format

```
/OBJEXTRA sectors
```

where:

```
sectors =   an unsigned integer where 0<= sec-
            tors <=65535
```

default:

```
/OBJEXTRA 0
```

14.8.2 Discussion

The OBJEXTRA option causes the compiler to allocate extra free sectors to an object program when creating a new program file in a diskimage.

This feature increases the chances that the same catalog space already allocated to a given application program can be reused on subsequent modifications and recompiles.

This option is ignored if the OBJLOC is not a diskimage.

14.9 The LSTLOC Option

The LSTLOC option is used to indicate the desired location to receive the listing. This option is used in conjunction with LSTFORMAT. The general form is as follows:

14.9.1 Format

```
/LSTLOC {[pathname][filename]}
        {print device}
where:
     pathname =  the native operating system path-
                 name to contain the resultant pro-
                 gram listing(s) directly.

     filename =  the name of the file to contain the
                 resultant program listing(s).
default:
     /LSTLOC nul device
```

14.9.2 Discussion

The NPL compiler optionally produces "source" program listings from the input programs in several formats, based on the specification of the LSTFORMAT options. The LSTLOC option is used to indicate the desired location to receive the listing. There are five possible LSTLOCs. These are described in the following sections.

14.9.3 Diskimage Files

If LSTFORMAT is 2200 or 2200S, LSTLOC can be specified as the name of a diskimage file. This is used for specifying the location of the output 2200 atomized programs. This output diskimage file can then be copied to "raw" 320K or 360K (PC Interchange) format diskettes using Niakwa-supplied utilities for operation on a Wang 2200.

The diskimage file specified must exist. The compiler does not create it. If programs of the same name as those being compiled already exist on the specified LSTLOC diskimage file, they are overwritten. If there is not enough space, files are located to the end of the diskimage, space permitting. Else, an appropriate error message is generated.

NOTE: The OBJLOC cannot be used to specify the output for anything other than compiled NPL programs.

The LSTLOC should be a diskimage file which is different from the SRCLOC. Otherwise, input compiled programs are overwritten.

The OBJLOC can be specified as the null device when creating 2200 atomized programs. This suppresses generation of a second set of compiled NPL programs.

14.9.4 Native Directory

If the LSTLOC is specified as a native operating system directory, all source listings are stored in that directory, one file for each source listing. The filename is derived from the input program name, with an extension of .LST or .SRC, depending on the selected LSTFORMAT (refer to Section 14.10). If the derived filename already exists in the directory, it is overwritten.

14.9.5 Native Files

If the LSTLOC is specified as a native operating system file, and LSTFORMAT is either .SRC or .LST, the compiler writes all source listings (concatenated) to that one file. The format of the listing is determined by the LSTFORMAT option.

14.9.6 "RAW" Diskettes

Diskettes with a standard 2200 catalog structure are directly readable by the compiler. This type of diskette is called a "raw" diskette as it contains no native operating system file structure whatsoever. Refer to the chapter "Supported Devices" of the NPL Supplement(s) for information on "raw" diskette support on the operating system being used.

For example, under MS-DOS, LSTLOC is specified as:

```
/LSTLOC A:
```

The "A:" drive designation is treated by the compiler (and RunTime program) as a special case meaning "the floppy drive" in 320K format. This allows for the creation of 2200 format programs when using 2200 or 2200S options of LSTFORMAT or wildcard patterns to be compiled.

This is the normal output format for programs being converted to a Wang 2200 with a 2275 diskette drive.s available on the 2200 system and has been used to produce a 320K "raw" format diskette.

14.9.7 Print Device

Hardcopy listings may be produced by specifying LSTLOC as the name of a native operating system print device. Refer to the chapter "Supported Devices" of the NPL Supplement(s) for a description of valid print-device names for the operating system being used.

14.10 The LSTFORMAT Option

The LSTFORMAT option selects one of four forms for program listings.

14.10.1 Format

```
/LSTFORMAT {.SRC, .LST, 2200, 2200S}
```

default:

```
/LSTFORMAT .LST
```

14.10.2 Discussion

The LSTFORMAT option of the NPL compiler produces four formats of program listings. The following sections discuss each list format in detail.

14.10.3 .LST Format

This format of source listing includes supplementary information which may be of value to programmers. The standard extension for listing files is ".LST". The source of each statement appears, as well as the offsets (in hexadecimal) of the start of each statement. Should a non-recoverable error occur while a program is executing under the non-interpretive RunTime (RTP), these offsets may be used to locate which statement of a multi-statement line was in program when the error occurred (refer to Section 14.13).

14.10.4 .SRC Format

Specification of the .SRC format instructs the compiler to create ASCII text files which may be re submitted to the compiler. This is used primarily to obtain an editable format of the input file from a non-editable one (in particular, from Wang 2200 atomized format). The standard extension of these files is ".SRC".

WARNING--If compiling from a .SRC file, do not use the LSTFORMAT .SRC option--if the listing location (LSTLOC) is the same as the input location (SRCLOC) the input file is overwritten

14.10.5 2200 Format

Use of the LSTFORMAT option, "2200", creates programs in 2200 atomized format. These programs are fully compatible with the Wang 2200 and can then be ported directly to the 2200 without any other intermediate processes.

Refer to Section 14.21 below for details regarding compatibility when generating 2200 atomized code.

14.10.6 2200 "Compressed" Format

Use of the LSTFORMAT option, "2200S", creates programs in 2200 atomized format, (just as the "2200" option does), however, in addition, this option REMOVES spaces in program code which are not syntactically significant. This option is useful when regenerating a 2200 version of "compressed" programs, especially when spaces have been added to the code during development for readability.

NOTE: All spaces in REM statements are still preserved.

These programs are fully compatible with the Wang 2200 and can then be ported directly to the 2200 without any other intermediate processes.

14.11 The ERRLOC Option

The ERRLOC allows the summary of activity to be redirected to either the printer or a file. The general form is as follows:

14.11.1 Format

```
/ERRLOC {[pathname][filename]}
        {print-device        }
```

where:

```
pathname =  the native operating system path-
            name to contain the resultant pro-
            gram error report file.

filename =  the name of the file to contain the
            resultant program error report.
```

default:

```
/ERRLOC nul device
```

14.11.2 Discussion

The compiler normally prints an abbreviated summary of activity to the screen as it compiles. The ERRLOC option allows for placing this output on disk for review later, or for redirecting the output to the system printer.

14.11.3 Native Files

If the ERRLOC is specified as a native operating system file, all error information is directed to the file as it appears on the display. Any legal native filename and extension is acceptable. If the file already exists in the directory it is overwritten.

14.11.4 Print Device

Hardcopy listings may be produced by specifying ERRLOC as the name of a native operating system print device. Refer to the Chapter 5 of the appropriate NPL Supplement for a description of valid print-device names for the operating system being used.

NOTE: Even if the error report is redirected to a file or to the printer, it also appears on the terminal (standard output).

14.12 The WARNINGS Option

The WARNINGS option will suppress the generation of warning messages. The general form is as follows:

14.12.1 Format

```
    /WARNINGS {ON, OFF}
default:
    /WARNINGS ON
```

14.12.2 Discussion

In addition to syntax errors, the compiler may produce a number of messages that warn of certain conditions which, though not necessarily errors, are likely to cause RunTime errors. The warnings do not affect the correct generation of compiled code, and a program which receives only compiler warnings is still executable. Such messages always commence with "Warning - " to distinguish them from fatal errors.

If, for any reason, it is desired that this level of error checking be suppressed by the compiler, the WARNINGS option is used to do this.

For example:

```
/WARNINGS OFF
```

tells the compiler to suppress generation of warning messages.

14.13 The NUMBERS Option

The NUMBERS option instructs the compiler to generate additional object code to keep track of which statement number within a line is being executed. This option is used for tracing an error in multi-statement line programs. The general form is as follows:

14.13.1 Format

```
/NUMBERS {ON, OFF}
```

default:

```
/NUMBERS OFF
```

14.13.2 Discussion

In the event that the non-interpretive RunTime program encounters a non-recoverable error, it displays information concerning the line number on which the error occurred, the NPL error code, and the offset within the line (in terms of bytes of generated code). When the source code does not make extensive use of multiple statement lines, this information should be sufficient to locate the cause of the problem.

If the code makes extensive use of multiple statement lines, the .LST file generated by the compiler can be used to determine which of the statements is at fault (the offset is frequently that of the END of the statement on which the error occurred).

Should this prove inadequate, the NUMBERS option instructs the compiler to generate additional object code to keep track of which statement number within a line is being executed. Please be aware that this option incurs a small overhead penalty both in size of code (2 bytes per statement) and in execution time.

For example:

```
/NUMBERS ON
```

instructs the compiler to generate object code to keep track of sub-statements.

14.14 The REM$ Option

When the REM option is turned ON, the compiler assumes that ALL statements which start with "REM $ PC" have been added with 2200 compatibility in mind, and compiles the program code contained in these statements. The general form is as follows:

14.14.1 Format

```
/REM$ {ON, OFF}
```

default:

```
/REM$ OFF
```

14.14.2 Discussion

Programs written for the Wang 2200 can be upgraded to make use of extensions added for the NPL version, while maintaining compatibility with the 2200 system. This is done by coding statements whose syntax is not 2200-compatible within special types of REM (remark) statements. Since all REM statements are ignored on the 2200, the program code in these REM statements causes no difficulties on the 2200.

When the REM option is turned ON, the compiler assumes that ALL statements which start with "REM $ PC" have been added with 2200 compatibility in mind, and compiles the program code contained in these statements. Otherwise, REM statements are ignored by the compiler. (Refer to the Statements Guide for further details on the use of REM $ PC.)

The default value of the REM$ option is OFF (no compilation of any REM statements is performed).

NOTE: If REM$ is ON, REM statements that begin with "$" but not with "$PC" are flagged with warnings. This is because some earlier versions of the compiler did not always require the "PC" keyword.

14.15 The KEEPREMS Option

The KEEPREMS compiler option may be set to ON to retain all REMs and program spacing when compiling programs. This option is used for downward compatibility to rev. 1.03. The general form is as follows.

14.15.1 Format

```
/KEEPREMS {ON, OFF, DEC}
default:
/KEEPREMS OFF
```

14.15.2 Discussion

The KEEPREMS option is used to either eliminate remarks and extra spacing from programs or to reformat statements, placing each statement on a separate physical line.

The KEEPREMS option affects only the OBJLOC, not the LSTLOC.

14.15.3 The KEEPREMS Compiler Option

The KEEPREMS compiler option may be set to ON to retain all REMs and program spacing when compiling programs, or to OFF to suppress generation of code related to REMs and program spacing. The DEC option causes multiple statements on a line always to be broken up into separate lines when LISTed or recalled for edit, much the same as the decompressed LIST function of the Interpreter. The DEC option is basically the same as the ON option, except that program spacing is not retained. The option may be entered as part of the command line or on the compiler display.

NOTE: The REM$ option is indirectly affected by the KEEPREMS option such that:

If REM$= ON and KEEPREMS= ON then REM$ statements are compiled and the REM$ PC designation is retained in the object program.

If REM$= ON and KEEPREMS= OFF then the statement following the REM $ PC is compiled, but the REM $ PC designation is removed from the object program.

If REM$= OFF and KEEPREMS= OFF then the entire REM $ PC and the associated statement are removed from the object program.

If REM$= OFF and KEEPREMS= ON then the REM $ PC and the associated statement are retained but the comma is inserted between REM and $ which makes the statement a true REM in the object program. That is, the statement is not executed by the RunTime unless the comma is removed.

Note that the above comments refer only to the generation of object code. Output to the LSTLOC is never affected by the KEEPREMS or REM $ PC options. The REM $ PC designation is always retained.

To retain the original form of literals when compiling programs, specify the KEEPREMS option as ON or DEC.

If KEEPREMS is OFF, then any literals which contain any character which is outside of the range HEX(20) to HEX(7F) or contain the character HEX(22) (double quote) are displayed as HEX. Otherwise, the literal is displayed as ASCII.

The KEEPREMS option can also be used to generate "compressed" code for installation at end-user sites, if desired, by specifying KEEPREMS=OFF.

14.16 The DISPLAY Option

The DISPLAY option is used to invoke the compiler menu. The general form is as follows:

14.16.1 Format

```
/DISPLAY {ON, OFF}
```

default:

```
/DISPLAY OFF
```

14.16.2 Discussion

The NPL compiler features an alternate method of specifying compiler parameters (i.e., options and program names on a native operating system command line). Compiler parameters may be entered using a more "user-friendly" compiler options display (refer to Section 14.19). The following sections discuss the two ways to access this display.

14.16.3 Explicitly Invoking the Compiler Display

Presence of the DISPLAY ON option in any compiler command line invokes the compiler options display. Any other options or parameters on the command line, entered in addition to the DISPLAY option, appear on the compiler options display in the appropriate fields. The display then allows for easy review and editing of the displayed parameters.

This method is most useful when using native operating system batch files with the compiler (refer to Section 14.20). It allows the programmer to review and edit the options specified in the batch file before committing to the compile.

14.16.4 Implicitly Invoking the Compiler Display

The compiler display is implicitly invoked by entering the compiler name ("B2C") on the command line with options (if desired) but no prognames. Here, the display appears with the Source Prognames field blank. For those options specified they appear in the appropriate fields; for those options not explicitly entered, defaults appear.

14.17 The TRANSLATE Option

The TRANSLATE option is used to inform the compiler of the assumed character equivalences between native file names and file names within a diskimage file. The general form is as follows:

14.17.1 Format

```
/TRANSLATE {hexquad   }[[.][hexquad   }]...
```

where:

```
hexquad =    {hh}{gg}[[.]{hh}{gg}]...
             {=x}{=y}    {=x}{=y}
```

where:

```
hh =         two hexdigits specifying a charac-
             ter (hex(hh)) which may appear in
             the native file name.

=x =         specification of a character ("x")
             which may appear in the native file
             name.

gg =         two hexdigits specifying a charac-
             ter (hex(gg)) which may appear in a
             diskimage file name.

=y =         specification of a character ("y")
             which may appear in a diskimage
             file name.
```

14.17.2 Discussion

The file naming conventions of the various NPL supported operating systems are not entirely compatible with the names of files within NPL diskimages. Specifically, within diskimages, file names may be any 8 characters, with no restrictions. However, MS-DOS for example, allows 8 character file names, but the permitted character set is restricted. Only the letters A-Z, digits 0-9, and a few additional graphic characters are allowed. Exactly which ones depends on the release of the operating system.

Consequently, problems may arise when the compiler attempts to take the name of a program within a diskimage (example: GL G/L >") and create a .SRC file with "the same name" under the MS-DOS operating system, since "/" and "> " have special meanings in MS-DOS file names. The problem has no general solution, but, if a system does not use too "strange" a subset of ASCII on filenames in the diskimage, it can be circumvented with the TRANSLATE option.

The TRANSLATE option is used to inform the compiler of the assumed character equivalences between native file names and file names within a diskimage file. The TRANSLATE verb is followed by a series of groups of four hex digits. Each group specifies a FROM/TO pair of hexcodes, the first two hexdigits representing the native character, the second two representing the equivalent character to be used for file names within diskimage files.

For example:

```
./TRANSLATE 5F20
```

informs the compiler that underline characters (HEX(5F)) in native file names are equivalent to space characters (HEX(20)) in names of files within NPL diskimage files. Consequently, whenever the compiler needs to place names of files from within NPL diskimage files directly into an native directory, it replaces the embedded blanks in the filenames from within the diskimage file with underlines. Conversely, when the compiler needs to place a native filename into a diskimage, it replaces underlines with spaces.

For legibility, groups of codes may be separated by a decimal point (".").

For example:

```
/TRANSLATE 5F20.7B3C.7D3E.=%=/
```

Defines the following equivalences (where D.I. stands for Diskimages):

NATIVE (HEX)	5F	7B	7D	25
NATIVE (display)	-	{	}	%
Filenames in D.I. (display)	-	<	>	/
Filenames in D.I. (HEX)	20	3C	3E	2F

Trailing spaces in a file name are not considered to be part of the name. For obvious reasons, use of file names which use only upper case A-Z, 0-9 and space is preferred in most environments and is strongly encouraged.

14.17.3 Translation Is Performed Only When Required

Translation of file names takes place only when it is required. Consequently, compiles where the SRCLOC and OBJLOC are both diskimages need the TRANSLATE option only for listing files.

Similarly, compiles where the SRCLOC and OBJLOC are both native operating system directories require the TRANSLATE option only if LSTFORMAT is 2200 or 2200S, and LSTLOC is a diskimage.

14.18 The LCASE Option

With the LCASE option, 2200 filenames which contain lower case characters are supported to a limited extent. The general form is as follows:

14.18.1 Format

```
     /LCASE {ON, OFF}
default:
     /LCASE OFF
```

14.18.2 Discussion

NOTE: This section discusses upper and lower case sensitivity as it applies to the MS-DOS operating system. For information on case sensitivity within specific NPL supported operating systems, refer to Chapter 5 of the appropriate NPL Supplement.

Much like the TRANSLATE option, the LCASE option is used to circumvent file naming convention problems between the 2200 and the various NPL supported operating systems. On the 2200, file names which include lower case characters are legal, whereas on the PC, MS-DOS converts all lower case characters in filenames to upper case before storing.

With the LCASE option, 2200 filenames which contain lower case characters are supported to a limited extent. When the LCASE ON option is selected, the case of each character of the source program names specified for compilation on the command line are preserved when translating from MS-DOS filenames to 2200 file names.

For example:

```
B2C /OBJLOC platter1.bs2 /LCASE ON HIGHlow
```

compiles the MS-DOS file "HIGHLOW.SRC" from the current directory into the diskimage "platter1.bs2", but the file name used in the diskimage is "HIGHlow" (by preserving the case as specified in the command line). If the LCASE option had not been specified, the filename used in the diskimage would have been "HIGHLOW".

14.19 The User-Friendly Compiler Display

The NPL compiler features an alternate method of specifying compiler parameters (i.e., options and program names on a native operating system command line). Compiler parameters may be entered using a more "user-friendly" compiler options display. There are two ways to access this display: both involve entering a form of the normal compiler command line in the native operating system command processor but with special options.

14.19.1 Invoking the Display

1. Explicitly invoking the compiler display.

   The presence of the DISPLAY ON option in any compiler command line entered in the native operating system command processor invokes the compiler options display. Any other options or parameters on the command line, entered in addition to the DISPLAY option, appear on the compiler options display in the appropriate fields. The display then allows for easy review and editing of the displayed parameters.

This method is most useful when using native operating system batch files with the compiler. It allows the programmer to review and edit the options specified in the batch file before committing to the compile.

2. Implicitly invoking the compiler display.

   The compiler display is implicitly invoked by entering the compiler name ("B2C") on the command line with options (if desired) but no prognames. In this case, the compiler display appears with the Source Prognames field blank. For those options specified they appear in the appropriate fields; for those options not explicitly entered, standard defaults appear.

NOTE: The following examples use MS-DOS naming conventions, delimiter characters and keyboard characteristics. For delimiter equivalences and naming conventions within specific NPL supported operating systems, refer to Chapter 5 of the appropriate NPL Supplement. For keyboard characteristics of specific terminals, refer to Appendix D of the Programmer's Guide.

14.19.2 The Display Screen

When the display option is requested by either of the methods discussed above, the following display appears.

```
                            Niakwa Runtime Compiler
                              B2C Rev 4.00.08.00.I

SOURCE  Programs :
        Location :

OBJECT  Location :
        Format   :             [UN]SCRAMBLED   Extra Sectors:

LISTING Location :
        Format   :        { .SRC, .LST, 2200, 2200S}

ERRORS  Location :

OTHER            :  Warnings:    {ON, OFF}   Compile REM$   :    {ON, OFF}
                 :  Numbers :    {ON, OFF}   Keep Lower Case:    {ON, OFF}
                 :  Display :    {ON, OFF}   Keep REMs      :    {ON, OFF, DEC}
                 :  Progname $TRAN

No Source Programs specified.
                                        EXECUTE - Proceed
                                        CANCEL  - Cancel
```

As shown, each possible compiler option is represented as a field on the display. If a field has specified required input, the acceptable responses are shown after the field. The normal conventions for { }, [], symbols, etc., are employed (refer to the Preface of this guide).

The RETURN and TAB keys move the input block from field to field in a left to right path through the display. When the end of a line is encountered the input block moves to the next line.

The BACK-TAB key moves the input block from field to field in a right to left path through the display. When the beginning of a line is encountered, the input block moves up to the previous line.

The EAST, WEST and BACKSPACE keys move the cursor in the usual direction one character at a time within a field. The DELETE and INSERT keys are active and also perform the usual functions.

Depression of the EXEC key at any time, causes the compiler to proceed with the options as displayed. Before compilation takes place, the compiler performs input field validation. If any discrepancies are found, a message appears at the bottom of the display describing the problem. The input block is then placed at the field in error to allow for editing. To proceed after editing, simply press EXEC.

Depression of the END key at any time cancels the compiler run and returns to the MS-DOS operating system.

14.19.3 Limitations of the Compiler Display

Using the compiler display does introduce some limitations not found when using the native operating system command line method. These are:

- Field lengths are fixed in length on the display, as opposed to native operating system command lines where field lengths are variable.
- Only one set of options may be specified for all programs to be compiled, whereas in the command line method the compiler options may be varied throughout a compile.

14.19.4 Compiler Display Example

Entering the following compiler command line when in MS-DOS:

```
B2C /DISPLAY ON PROG1 PROG2 PROG3
```

produces the following compiler options display:

```
                              Niakwa Runtime Compiler
                                B2C Rev 4.00.08.00.I

SOURCE  Programs :  prog1 prog2 prog3
        Location :  .

OBJECT  Location :  .
        Format   :  UNSCRAMBLED [UN]SCRAMBLED   Extra Sectors:0

LISTING Location :  /dev/nul
        Format   :  .LST   { .SRC, .LST, 2200, 2200S}

ERRORS  Location :  /dev/nul

OTHER            :  Warnings:ON  {ON, OFF}   Compile REM$    :OFF {ON, OFF}
                 :  Numbers :OFF {ON, OFF}   Keep Lower Case:OFF {ON, OFF}
                 :  Display :OFF {ON, OFF}   Keep REMs      :OFF {ON, OFF, DEC}
                 :  Progname $TRAN 5F20

No Source Programs specified.
                                         EXECUTE - Proceed
                                         CANCEL  - Cancel
```

All three programs are displayed in the Source Programs field as specified. Since no options were specified on the command line, the compiler displays default options.

NOTE: The "." in the SOURCE Location and OBJECT Location fields is the symbol MS-DOS uses to indicate "the current directory". The compiler uses the same notation

14.20 Batch Files

Batch files are functions which allow the user to set up and execute files containing native operating system commands and parameters.

14.20.1 Discussion

Batch files can be used to invoke the NPL Compiler. For example, under MS-DOS, the following example compiler command line:

```
B2C /SRCLOC platter1.bs2 /OBJLOC platter2.bs2 /LSTLOC .
/LSTFORMAT .src AR*
```

can be set up as a batch file called COMPAR.BAT (the .BAT extension is required), and then be executed from the command processor by simply typing in the name of the batch file:

```
COMPAR
```

In this case, the batch file would perform exactly the same as the example command line it was taken from.

It is possible to generalize batch files so that some options may be entered during execution of the batch file. This is handy for specifying the list of input program names to compile. This can be done by a simple modification to the example above. The modification would be to replace the literal program name designation with a replaceable parameter value ("%N") where N is a number from 1 to 9:

```
B2C /SRCLOC platter1.bs2 /OBJLOC platter2.bs2 /LSTLOC .
/LSTFORMAT .src %1 %2 %3 %4 %5 %6 %7 %8 %9
```

This batch file would now be able to accept up to nine source program names.

For example:

```
COMPAR AR* GL101 OE*
```

compiles the following sets of programs:

- All programs with the prefix "AR"
- Program GL101
- All programs with the prefix "OE"

The same compiler options are used for all programs.

14.20.2 Example Batch Files

Niakwa has provided some convenient batch files. Of course, users may want to set up their own in addition to these. Please refer to Section 4.3 of the appropriate NPL Supplement for information pertaining to batch files within the operating system being used.

14.21 2200 Compatibility Issues

Section 14.10.5 above explained the use of the LSTFORMAT option, "2200", to generate 2200 atomized code. There are several points to consider related to compatibility in the generation of 2200 atomized code:

1. NPL extensions.

   NPL contains many extensions over Wang 2200 Basic-2. These statements, of course, are not executable on the 2200. The compiler does not report the presence of any Wang 2200 incompatible statements.

2. REM $ PC.

   Any Wang 2200 incompatible statements can be encoded into NPL programs by use of the REM$ PC statement. Both the Revision 2.00 compiler and RunTime retain the REM $ PC designation but execute statements prefaced with REM $ PC. Use of the REM $ PC statement for all NPL statements not supported on the 2200 ensures compatibility with the 2200.

   However, programs containing REM $ PC compiled with the Revision 1.03 compiler do NOT have the REM$ retained. If these programs are then transferred back to the 2200, they may fail at execution time if the REM $ PC statements are in fact specific to NPL. In addition, if these statements were intended for execution only under NPL, this may cause further difficulties.

   The only solution to this is to either check programs for use of REM $ PC or to recompile original 2200 versions with KEEPREMS ON and REM$ ON under the Revision 2.00 compiler.

3. NPL supports a much longer program line length than Wang Basic-2. The Revision 2.00 compiler detects program lines which are longer than those supported on the 2200 and automatically breaks up such lines into two or more program lines. A message to that effect is displayed during compilation.

NOTE: In this situation the compiler attempts to use the next sequential line number for the second program line. However, if the next sequential line number is already used, compilation of the program is aborted, and the error is reported. A change must be made to the program before it can be reatomized for use on a 2200 system.

If lengthy program lines are used, the LSTFORMAT 2200S option may be beneficial. This results in shorter program lines being generated and thus reduces the possibility of lines having to be split.

4. The compiler cannot generate SCRAMBLE protected 2200 code. If this is a requirement, a Wang 2200 is required.

5. The compiler does not access NPL programs saved in a SCRAMBLE protected format, whether generated by the SAVE! or by the SCRAMBLE option of the compiler when initially compiled.

6. The procedure for de-compiling object code into 2200 atomized code is not without certain anomalies, however. Reconstruction of some specific statements and syntactic elements of a program produces code that is functionally equivalent, but somewhat different in structure. This is an important consideration.

NOTE: These same anomalies apply to the use of the Interpreter as well.

Refer to Appendix F for specific examples of source code de-compilation differences.

14.22 Compatibility with Earlier Revisions

In addition to the compatibility issues related to generation of 2200 atomized code, there are issues related to downward compatibility to earlier RunTime revisions.

14.22.1 New Statements

All new NPL statements introduced in the current revision of NPL that is being run, are not compatible with previous revisions of the RunTime. If unsure of a statement's compatibility with a previous release of NPL, refer to the Language Compatibility chart in contained in Appendix B of the NPL Statements Guide.

14.22.2 Program Stamp

In all cases where a program is not executable under an earlier release of NPL, the header sector of the program is stamped to indicate the RunTime revision or greater required to run the program. Attempting to execute a program with an incompatible revision of the RunTime results in a D88 - Wrong Record Type error.

14.22.3 Using the Compiler

If it is not certain whether or not program structure has been modified using the Interpreter, the best way to ensure compatibility with an earlier RunTime revision is to run them through the compiler with the KEEPREMS option OFF. The compiler generates appropriate warning messages on statements requiring a specific RunTime revision or greater.

14.23 Commonly Used Compiler Options

A series of examples showing some typical uses of the compiler are presented in the chapter "Compiler Operation" of the NPL Supplements. The examples in each Supplement are functionally identical to the examples in any other Supplement but illustrate the operating system specific features of the compiler.

CHAPTER 15

PORTING PROGRAMS AND DATA

15.1 Overview

This chapter discusses the various methods of transferring programs and data between NPL-supported environments or from a Wang 2200 to an NPL-supported operating environment.

Section 15.2 describes porting options by serial communication or diskette transfer.

Section 15.3 describes the Niakwa general backup and recovery utilities and explains the steps involved in a sample port.

Section 15.4 describes the Niakwa general file copy utility and outlines the steps involved in a typical port.

Section 15.5 discusses the use of user-written transfer utilities for the purpose of porting programs and data.

Section 15.6 discusses the methods of transferring programs and data between a Wang 2200 and NPL-supported systems.

Section 15.7 discusses the various file types available for porting and the methods of transferring them.

15.2 Porting Options

There are two possible options for transferring NPL applications and data. These are:

- Serial Communications
- "RAW" Diskette Transfer

These options are discussed in the following sections.

15.2.1 "RAW" Diskette Transfer

The ability of NPL to access "raw" diskettes provides a means of copying NPL diskimages from one machine to another where the native operating system's file formats are incompatible. A "raw" device is a physical disk which does not contain a native file system.

NOTE: Support of these "raw" devices is extremely hardware-dependent. A hardware-specific compatibility table detailing which "raw" devices are supported on systems running NPL is provided in Appendix G of the Programmer's Guide. Further information about accessing "raw" diskettes across the various operating environments supported by NPL is discussed in Chapter 5 of the appropriate NPL Supplement.

Although, in most cases, any of the supported "raw" diskettes can be used for porting, the remainder of this chapter assumes that a 360K "raw" diskette is being used.

NPL provides support for 5-1/4" 320K, 360K, 1.2MB, and 3-1/2" 720K, 1.44MB, 2.88MB "raw" formatted diskettes. Most NPL-supported platforms handle one or more of the above formats (refer to Appendix G for a list of all NPL-supported media). Provided that diskette compatibility exists between two supported platforms, the following steps are required to port between one NPL-supported platform and another.

1. Select a compatible "raw" format for the destination systems (being transferred to),

2. Use either the Niakwa Backup and Recovery utilities or a user's custom transfer utilities to transfer programs and data between systems.

NOTE: Although the "raw" formatted diskettes used are identical from one system to the next, the naming conventions of these diskettes may change from one machine's native operating system to the next.

For example:

Under Novell	$DEVICE(/D10)="A: 360=Y"	Instructs NPL to treat A: as a "raw" 360k device.
Under SuperDOS	$DEVICE(/D10)="1: 360=Y"	Instructs NPL to treat 1: as a "raw" 360k device

Refer to Chapter 5 of the appropriate NPL Supplement for supported diskette naming conventions. Refer to Appendix G of the Programmer's Guide for a listing of currently supported "raw" disk formats on supported platforms.

HINT: When possible, pre-format "raw" diskettes on the destination system. Although this is not required, it can alleviate some diskette integrity problems that may result from a diskette formatted on a system with a dirty or misaligned floppy drive.

15.2.2 Serial Communications

As an alternative to file transfer on magnetic media, developers may choose to upload and download their program and data files using serial communications.

Direct transfer from a Wang 2200 to supported NPL environments using serial communications is possible. The methods used for file transfer from the 2200 to a supported NPL environment vary from one operating environment to the next. Generally, file transfer considerations fall into two categories:

1. Porting to an MS-DOS based environment.

When porting between an MS-DOS based system, a number of communication packages on the market may be used to communicate with and transfer files to and from a Wang 2200. Among these packages, the most commonly used is a Wang 2236 terminal emulation product called PC2200. PC2200's file transfer capabilities provide a convenient alternative to "raw" diskette transfer.

2. Porting to systems other than MS-DOS.

   When porting from a Wang 2200 to an NPL-supported native operating system other than MS-DOS, we are unaware of any communication product that allows direct transfer of files from the 2200.

   If serial transfer is required, the suggested method of porting from a 2200 to a non-MS-DOS system, is to perform the port to an MS-DOS system as an intermediate step.

   First, use a package such as PC2200 to transfer programs and data to the MS-DOS system. Once programs and data are on the MS-DOS system, they can be easily ported to any NPL-supported operating systems, using a variety of terminal emulation or communication products.

   The above information is provided as a guideline to use when considering serial communications. Please refer to the appropriate NPL Supplement for specific Hardware operating environment porting considerations.

NOTE: It is not necessary to have NPL installed on the MS-DOS system being used as the intermediate system. PC2200 is capable of creating NPL diskimages on the fly.

Any file transfer product used must be capable of transferring 8-bit binary data.

15.2.3 Native Operating System Utility Options

In addition to the methods described above, other methods of transferring application program and data files between NPL-supported environments do exist. For example, a developer may choose to use the UNIX "tar" command to transfer a diskimage from a development system to a user's site. The UNIX DOS copy utility is also a commonly used vehicle for transferring files between UNIX and DOS.

NOTE: **The naming conventions and use of native operating system transfer utilities can vary from one system to the next. For example, the UNIX DOS copy utility may be "doscp" under one flavor of UNIX while it may be "doscopy" on another. Developers should consult operating system-specific documentation for specific details.**

15.3 NPL General Backup/Recovery Utilities

The Niakwa 2CBCKP (Backup) and 2CRCVR (Recover) utilities may be used to perform backups or file transfers of NPL diskimage or Wang 2200 disk platters which would require multiple diskettes. For instance, 2CBCKP may be used to backup an entire 2200 disk of programs or data onto a series of "raw" diskettes. 2CRCVR could then be used to recover these diskettes to a diskimage on any NPL-supported operating environment.

These utilities have been included with the NPL Development Package on the Utilities Diskette (in compiled form). To use the 2CBCKP Utility on a Wang 2200, it has to be decompiled and transferred to the 2200. Once this is done, it can be used to back up the 2200 disk platter to a series of "raw" format diskettes. Then use the compiled 2CRCVR utility to recover the 2200 disk platter to a diskimage file. Refer to Chapter 13 for instructions on the operation of the 2CBCKP and 2CRCVR utilities.

The NPL utility ATDEVICE can be used to create the diskimage to receive these diskettes. If using a diskimage address other than D10 or D12, set the diskimage up using the ATDEVICE utility.

NOTE: **In some instances, 2CBCKP and 2CRCVR may not be the best method of transferring software from the 2200 (specifically, if there is limited room on the hard drive). That is, this method requires that there be enough room on the drive to hold both source programs and the soon-to-be-created object programs. This is because the compiler cannot compile directly from diskettes created by the General Backup Utility. Diskettes created by the General Backup Utility do not have a traditional catalog. Therefore, one is forced to copy the diskettes to a diskimage on the hard disk. The programs can then be compiled from this diskimage.**

15.3.1 Steps Required for Transfer

To move the utility programs from a system to the 2200 requires that the programs first be decompiled into 2200 format on diskette, after which the diskette may be used directly on the 2200 for the copying procedure. Following are the steps involved in this process:

1. Install the Niakwa NPL Development Package and RunTime Package software on the host system. Refer to Chapter 2 of the appropriate NPL Supplements for details on installing the software.

2. De-compile to transfer the necessary NPL utility programs onto diskette by executing the following compiler command line (and optional display) from the system prompt (MS-DOS assumed):

```
B2C /DISPLAY ON /SRCLOC \BASIC2C\UTILITY.BS2
/OBJLOC \DEV\NUL /LSTLOC A: /LSTFORMAT 2200
2CBCKP 2CRCVR
```

NOTE: Switch characters and directory designations vary from one operating system to another. Refer to Chapter 3 of the appropriate NPL Supplement for details.

This creates a 320K format diskette containing the transfer utility programs 2CBCKP, and 2CRCVR in 2200 atomized code, which may be used directly on the 2200.

NOTE: The diskette must be preformatted in 320K format before beginning the de-compilation procedure. The formatting procedure to use may vary from system-to-system. Please refer to Chapter 5 of the appropriate NPL Supplement for information on the environment being used.

3. Place all programs and data to be transferred onto a single platter on the 2200.

4. On the 2200 system, use the 2CBCKP utility program from the diskette (created in step 2) to transfer this platter to a series of "raw" diskettes.

   A. The user may LOAD/RUN program "2CBCKP" directly from the diskette. The diskette may be removed once the program begins execution. There are no sub-modules overlaid.

   B. Specify full disk backup (if desired, selected files may be transferred using this utility).

   C. Specify the input address as the address of the 2200 platter to be transferred.

   D. Specify the output address as the 2200 address of the diskette drive.

E. Specify the size of the output diskette:

1279 = 320K
1439 = 360K

NOTE: Please refer to Chapter 13 for details on the operation of the 2CBCKP utility.

It is not necessary to preformat diskettes for use by this utility. The 2CBCKP program formats diskettes, if desired. However, it uses 320K format. This is because the Wang 2200 $FORMAT DISK statement only formats in 320K format. If 360K format is desired, the diskettes must be preformatted using the technique described in Section 15.6.

The 2200 disk platter to be transferred has now been backed up to a series of 5-1/4" diskettes.

5. Use the NPL utility "New Diskimage Creation" to create the diskimage to contain the transferred platter on the PC:

   A. Execute the RunTime program specifying UTILITY.OBJ as the boot program at the MS-DOS prompt:

```
CD \BASIC2C    Select \BASIC2C as the current directory.
RTP UTILITY
```

   This executes the RunTime program specifying UTILITY.OBJ as the boot program.

   Refer to Chapter 4 of the appropriate NPL Supplement for details on executing the RunTime program.

   B. This first loads the ATDEVICE utility, which allows the user to specify any new disk addresses/diskimages required. Refer to Chapter 13 for details.

   C. The utilities menu is then loaded.

   D. Select the New Diskimage Creation utility. This allows the user to create the diskimage in the size required. Refer to Chapter 13 for details.

6. Recover the platter from the diskettes created in step 4.

A. From the utilities menu, select the General Recovery Utility (2CRCVR).

B. Specify D10 as the input disk address. D10 is set up as a default by the Run-Time program as a 320K "raw" diskette.

For example:

```
DEVICE(/D10)="A:"
```

C. Specify the output address of the diskimage.

NOTE: At this point, the user has a diskimage containing 2200 source files. To execute these programs, it is necessary to compile them. Refer to Chapter 14 for details on compiling.

Refer to Chapter 13 for details on the use of the NPL General Recovery Utility.

15.4 NPL General File Copy Utility Use

The Niakwa NPL general file copy utility (2CCOPY) provides a method of selectively copying selected groups of files between NPL systems or from the Wang 2200 to an NPL system. The benefit of this utility is that diskettes are created in a standard catalog format. Thus, programs can be compiled directly from diskettes created by 2CCOPY. This utility can also be used to transfer data files.

NOTE: 2CCOPY does not have the capability to span multiple diskettes. As a result, no one file or group of files may be larger than a diskette. If the total size of the file or files to be copied is larger than a diskette, the General Backup Utility (2CBCKP) must be used.

The steps to use the General File Copy Utility to transfer programs and data files are:

1. Install the Niakwa NPL Development Package and RunTime Package software on the host system. Refer to Chapter 2 of the appropriate NPL Supplement for details on installing the software.

2. Decompile the General file copy utility to a "raw" 2200 format 5-1/4 " diskette, as described in section 15.3.1 above, using the name 2CCOPY as the source program name.

3. Copy from the 2200 the data files to be copied and the programs to be compiled (they may be placed on the same diskette(s)) using the 2CCOPY program on the diskette created in step 2 above. Refer to Chapter 13 for details on the operation of the 2CCOPY utility.

4. Use the compiled new diskimage creation utility to create the diskimage to contain the transferred platter:

   A. Follow the steps described in step 5 of Section 15.3.1 above to bring up the utilities menu on the system.

   B. Select the New Diskimage Creation utility. This allows for creation of the diskimage in the size required.

5. Copy files to the platter from the diskettes created in step 3 above.

   A. From the utilities menu, select the General File Copy Utility (2CCOPY).

   B. Specify D10 as the input disk address. D10 is set up as a default by the Run-Time program as "raw" diskette.

      For example:

      ```
      DEVICE(/D10)="A:"
      ```

   C. Specify the output address of the diskimage.

   D. Copy only data files to the diskimage file. Programs may be compiled directly from the "raw" diskette(s). Refer to Chapter 14 for details on compiling programs from "raw" diskettes.

NOTE: Refer to Chapter 13 for details on the 2CCOPY utility.

15.5 User Programs

The two methods stated above work well for transferring application and user data files to most supported NPL systems. However, if there are other special requirements, the above two methods may be awkward or inefficient.

Any other custom third-party NPL utility program can be used to put the files necessary for transferring on "raw" diskettes. If using a custom third party utility or some other method to put source files onto "raw" 320K diskettes, there is no need to go any further. The NPL Compiler can handle the files in that format. Developers wishing to use their own utilities and integrate them with the utilities provided by Niakwa, should follow the steps in the following section.

15.5.1 Transferring Over Your Own Utility Programs

1. Put the utility program(s) on a 320K "raw" diskette.

2. Install the NPL Development Package software on the system.

3. It will be necessary to expand diskimage file UTILITY.BS2 to make room for the compiled program(s). Use the MOVE END statement or the CHANGE DISKIMAGE SIZE utility to do this.

4. Compile the program(s) from the diskette onto disk D35 (diskimage file C:\BA-SIC2C\UTILITY.BS2). From the MS-DOS prompt:

```
CD \BASIC2C (SET THE CURRENT DIRECTORY TO \BASIC2C)
B2C /SRCLOC A: /OBJLOC \BASIC2C\UTILITY.BS2 [program name wildcards]
```

   This executes the compiler and instruct it to access the source programs on a "raw" diskette, and place object programs in diskimage UTILITY.BS2. Refer to Chapter 14 for details on compiling a program.

5. Add the custom program to the utilities menu on D35 by modifying ASCII text file 2CMNDAT1.SRC in \BASIC2C (using a standard text editing program such as EDLIN) and recompiling it onto D35.

NOTE: 2CMNDATA is a simple program containing data statements accessed by the Niakwa NPL 2CMENU program. To compile 2CMNDATA, from the MS-DOS prompt:

```
CD \BASIC2C (Set the current directory to \BASIC2C)
B2C /OBJLOC UTILITY.BS2 2CMNDATA
```

This executes the compiler and instruct it to access source program 2CMNDATA in directory \BASIC2C, and place the object program in diskimage UTILITY.BS2.

Refer to Chapter 14 for details on compiling a program. Refer to Chapter 13 for details on the 2CMENU utility and the organization of the 2CMNDATA menu options file.

6. The custom program is now available for execution on the system.

15.5.2 Executing Your Own Utility Programs

To run a custom utility program from the utilities menu (MS-DOS assumed):

1. Execute the RunTime program specifying UTILITY.OBJ as the boot program. From the MS-DOS prompt:

   ```
   CD \BASIC2C        Select \BASIC2C as the current directory.
   RTP UTILITY
   ```

   This executes the RunTime program with UTILITY.OBJ as the boot program (RTI may also be used).

   Refer to Chapter 4 of the NPL Supplements for details on executing the RunTime program.

2. This first loads the ATDEVICE utility, which allows the user to specify any new disk addresses/diskimages required. Refer to Chapter 13 for details.

3. The utilities menu is then loaded.

4. Select the NEW DISKIMAGE CREATION utility. This allows the user to create the diskimage in the size required. Refer to Chapter 13 for details.

5. Select the custom program from the utilities menu.

15.6 Transfer to and from the Wang 2200

This section discusses the process of transferring application program and data files from a Wang 2200 using "raw" diskettes, to an NPL system which may be unable to recognize the default Wang 2200 320K "raw" diskette format. In addition, the types of files available for porting are also discussed.

15.6.1 Formatting a PC Interchange Disk on the 2200

Due to certain incompatibilities, access to 320K "raw" format diskettes is not supported on all systems. Refer to Appendix G of the Programmer's Guide for information on support of this format for specific NPL hardware operating environments.

There is, however, a feature that allows direct diskette transfer from a Wang 2275 or a Wang DS diskette drive using 360K "raw" diskettes which are supported by the Wang operating system. This feature allows a diskette to be formatted in a 360K disk drive as a 360K "raw" format diskette using 512-byte sectors, as opposed to the 320K format using 256 byte sectors. Diskettes formatted in this manner can be accessed under MS-DOS as "raw" 360K diskettes ($DEVICE(/D10)="A: 360= Y"). This 360K "raw" diskette can also be used as an alternative for porting to systems that do not support the standard Wang 2200 320k format. Refer to Chapter 5 of the appropriate NPL Supplement for more details on "raw" diskettes and associated naming conventions.

To create a "raw" 360K diskette, it must be formatted on either the Wang 2275 or Wang DS diskette drive using a special $GIO statement:

```
$GIO/xxx (0600 0700 70A0 4002 88D0 7040 0130 6A10 6802 4001 8B67,X$)
```

where:

xxx is the appropriate disk address, and
X$ is an alpha-variable dimensioned to be 15 bytes long.

NOTE: This special $GIO statement is for use with Wang 2200s only. It is not compatible with any release of NPL.

Diskettes can be scratched with an end sector value of up to 1439 sectors for 360K diskettes.

Upon execution of the $GIO statement, bytes 1-5 and 9-15 of the alpha-variable are used by the system, while bytes 6-8 are set to the error return. Below is a list of the error return values.

```
HEX(000000) if no errors
HEX(000004) echo error. retry the command.
HEX(010000) if ERR I98
HEX(020000) if ERR I91
HEX(040000) if ERR I94
HEX(000100) if ERR I95
HEX(000200) if ERR I93
HEX(000400) if ERR I96
```

The following is an example of a Basic-2 program used to format a PC Interchange diskette on the Wang 2200 system:

```
10 DIM X$15
20 REM FORMAT THE DISKETTE
30 $GIO/D10 (0600 0700 70A0 4002 88D0 7040 0130 6A10 6802 4001 8B67,X$)
40 IF STR(X$,6,3)=HEX(000000) THEN 60
45 IF STR(X$,6,3)=HEX(000004) THEN 20
50 STOP "Disk Error"
60 REM SCRATCH THE DISKETTE
70 SCRATCH DISK T/D10,LS=10,END=1439<
 : ERROR GOTO 50
80 STOP "DISKETTE FORMATTED AND SCRATCHED"
```

Once the diskettes have been pre-formatted, utility programs, such as Niakwa's General Backup (2CBCKP) and General Recovery (2CRCVR), can be used on the Wang 2200 and supported systems to port software between machines. Access to the diskette is somewhat slow on both systems.

The following is an example of a port to an MS-DOS system using this technique:

1. On the Wang 2200, preformat the diskettes using the special $GIO statement on the Wang 2275 or Wang DS diskette drive.

2. Back up all information needed to diskette. Remember the end sector value can be set to 1439, but does not have to be entirely used.

3. On the system in use, establish the proper device equivalence as required for 360K "raw" diskettes.

   For example:

   ```
   $DEVICE(/D10)="A: 360=Y"
   ```

4. Recover the information into an NPL diskimage on the system, making sure the program accesses the entire contents of the floppy (i.e., the program should not be hard-coded to read just 1280 sectors, if the full 1440 sectors are used).

   Refer to the Wang 2275 Disk Drive User Guide for details on creating 360K diskettes on a 2275 disk drive. Refer to the Wang DS User Guide for details on creating 360K diskettes on the Wang DS.

NOTE: Restrictions that apply when attempting to read diskettes in a 360K disk drive which have been written to in a 1.2MB disk drive still apply.

The Niakwa backup and file copy utilities request entry of the number of sectors for the diskette being accessed. Set this to 1439 for 360K diskettes.

The general recovery utility program (2CRCVR) automatically determines the size of the diskette.

On some hardware versions of NPL, the compiler (B2C) does not have the ability to access 360K "raw" diskettes while the RunTime does. In these cases, the user must copy the files to a diskimage on the hard drive first before compiling. Refer to the diskette compatibility chart in Appendix G of the Programmer's Guide for details on which operating systems are affected.

According to Wang DS documentation, use of the "PC Interchange" format should be supported on a DS equipped with a 1.2MB diskette drive. However, to date, test results on this are inconclusive and 1.2MB compatibility between NPL and the DS 1.2MB drive has not been established.

15.7 File Types for Porting

Three types of files may be transferred from the Wang 2200 to NPL.

- **Programs**. Even if it is not necessary to permanently store source programs on the system, it is necessary to transfer the programs to "raw" diskette(s) so that they can be compiled.

- **Application software data files**. These are files which do not contain user data, but rather contain information required by application software (such as data file parameters). If planning to compile source programs directly from a "raw" diskette and not transfer them to the system's hard disk, it is easier in the future to place all application software data files on separate diskettes from the source files. This is because the source program diskettes are accessed from the compiler, and the application software data files are accessed from utilities. If planning on maintaining source files in a diskimage, it is easier to keep the application data files on the same diskimage.

- **User data files**. These are files which contain actual data, whether test data or live data. Typically, these are the only files which may be larger than a 2275 or a DS diskette. This means that an NPL utility would have to be used, such as the Niakwa General Backup Utility, which allows a file to span diskettes when copying. Refer to Section 13.6 for details on the General Backup Utility.

15.7.1 Methods of Transferring

There are three different methods of transferring files from a 2200 to a system running NPL. In most cases, a combination of these methods is used for transferring the different types of files, as described in Section 15.2. The three methods are:

1. The Niakwa general backup and recovery utilities, 2CBCKP and 2CRCVR. Refer to Section 15.3 for details.

2. The Niakwa general file copy utility, 2CCOPY. Refer to Section 15.4 for details.

3. A user-written program. Refer to Section 15.5 for details.

The only compatible media which may be used between the Wang 2200 and NPL is the 320K or 360K diskettes in raw format (refer to the chart in Appendix G of the Programmer's Guide). These methods require that the NPL utilities be available on the system. The Niakwa 2CBCKP, 2CRCVR, and 2CCOPY utilities are available on disk D35 (diskimage file C:\BASIC2C\UTILITY.BS2) after installing the Development Package Utilities Diskette. These have been provided by Niakwa as part of the NPL Development Package.

If using one's own utility or program to transfer program or data files, it may be necessary to transfer that program individually, compile it, and either place it on the Utilities Menu provided on platter D35 or execute it directly.

Refer to Chapter 13 for details on the use of the NPL Utilities.

CHAPTER 16

MIXED LANGUAGE PROGRAMMING

16.1 Overview

While NPL provides a programming environment with a high degree of portability and reasonable performance capabilities, a number of users have expressed interest in having the ability to perform functions written in other compiled languages. The NPL (formerly Basic-2C) External Subroutine Development Kit (BESDK) provides an interface to external subroutines in response to this requirement.

There are both benefits and drawbacks as a result of using external calls for NPL. The benefits include a potential increase in execution speed for selected processor-intensive functions, and the capacity to access resources and features of a specific environment. The drawbacks include increased memory requirements, limited portability to other NPL environments and a potentially less friendly environment for testing and error diagnosis.

This chapter concerns itself with the general operation of the NPL External (formally Basic-2C) Subroutine Development Kit (BESDK). For a complete discussion of operating system/environment-specific considerations and supported languages, refer to Chapter 11 of the appropriate NPL Supplement and their appropriate NPL Addenda.

Section 16.2 discusses the interface between the NPL application and external subroutines.

Section 16.3 describes the specifications required for the external call interface to DEFFN' type subroutines.

Section 16.4 describes the specifications required for the external call interface to external FUNCTION and PROCEDURE type functions.

Section 16.5 describes the specifications required for the callback interface to native NPL FUNCTIONs and PROCEDUREs.

Section 16.6 discusses the necessity of a user-written mainline routine for linking external subroutines to NPL.

Section 16.7 discusses the RTPEXT subroutine and how to create it.

Section 16.8 discusses the process of creating a test program to verify that the subroutines work and link properly.

Section 16.9 addresses the process of creating and linking the external library.

Section 16.10 provides some cautions and restrictions that should be considered prior to implementing external calls into applications.

16.2 Interface to External Subroutines

The following section discusses the interface between the NPL application and external subroutines.

16.2.1 GOSUB/DEFFN Versus FUNCTION/PROCEDURE Interface

NPL supports three kinds of external interfaces:

- Calls to external subroutines, equivalent to DEFFN' subroutines (with extensions).
- Calls to external functions, equivalent to FUNCTIONs and PROCEDUREs.
- Callbacks from external libraries to FUNCTIONs and PROCEDUREs written in NPL.

The DEFFN' subroutine interface was implemented in earlier releases, and is supported primarily for compatibility reasons, to allow continued use of existing external libraries.

FUNCTIONs and PROCEDUREs were first introduced in Release IV, and the external FUNCTION interface is implemented in order to allow use of the new capabilities.

An external library may allow access to functions using either interface (or both), as shown in the examples in the following sections. For simplicity, the discussions and examples normally deal with these interfaces separately.

Before discussing the details of how external subroutines are implemented, the effect the availability of a set of external subroutines has on the NPL environment must be considered.

16.2.2 Using the GOSUB'/DEFFN' Interface

External subroutines are accessed using the same mechanism as the marked subroutines (GOSUB'x statements). When a GOSUB' statement is executed for which no DEFFN' statement is found in the NPL environment, NPL would normally halt with an error message. If external subroutines are available, however, such calls are directed to the appropriate function instead.

NOTE: Calls are directed to external subroutines ONLY when there is no corresponding DEFFN' statement in NPL. NPL DEFFN' subroutines always take precedence over any external subroutines which may be defined.

Under Release IV or later, a convention for calling external subroutines using names as aliases for numbers is also available. Refer to Section 16.7.5 for details.

The parameter types permitted when calling the external DEFFN’ subroutines are the same as those allowed when calling NPL DEFFN’ subroutines, but with an important difference. Calls to NPL DEFFN’ subroutines always pass the value of the parameters specified in the GOSUB’ statement to the receiving variables in the DEFFN’ statement one at a time (as if a series of LET statements were executed). When the program RETURNs from the subroutine, the only way to return values to the calling routine is by placing the results in variables whose names have been agreed upon as part of the calling convention.

For example:

```
0000 DIM S$124,R$124   :REM dimension to maximum source, result lengths
      : DIM X$100
0100 GOSUB ’100("The quick brown fox jumps over the lazy dog.")
      : X$=R$ :REM assume result comes back in R$
      : X=R :REM contains count of characters that changed
      : END
   0200 DEFFN ’100(S$)
      : R=0
      : L=LEN(STR(S$))
      : FOR I=1 TO MIN(L,LEN(STR(R$)))
      : IF STR(S$,I,1)="a" AND STR(S$,I,1)<="z" THEN DO
      :    STR(R$,I,1)=STR(S$,I,1) SUB HEX(20)
      :    R=R+1
      : ENDDO
      : ELSE STR(R$,I,1)=STR(S$,I,1)
      : NEXT T
      : IF L<LEN(STR(R$)) THEN STR(R$,L+1)=" "
      : RETURN
```

By comparison, external DEFFN’ subroutines may change the values of any of the parameters specified in the GOSUB’ statement. The usual convention is to specify parameters whose value is required by the external routine ("input" parameters) first, followed by the names of variables which are set by the external subroutine as the result of the operation ("output" parameters). In some cases, a variable may be specified which is both used and returned.

For example:

```
0000 DIM X$100
0100 GOSUB ’100("I Want this in upper case",X$,X)
   : REM external subroutine places result into X$ and X
   : END
```

Using the interpretive RTI, it is also possible to display the available external subroutines, using the LIST ’ statement.

Where the display would normally show the line number of a DEFFN’ statement, the LIST ’ statement displays "/EXT" to indicate an external subroutine is available. In addition, a string describing the subroutine is displayed.

For example:

```
:LIST '

/EXT DEFFN'100 (In$, Out$, N) - In$ (upper case)-> Out$, N=#changes
- 0100
```

As with calls to DEFFN' subroutines written in NPL, the number of parameters must be fixed and the types (numeric or string) must match the subroutine being called, or an error occurs at execution time.

The above interface has both advantages and disadvantages. The advantages are that the subroutine calls are simple to use and require no new syntax, and that the effect of a subroutine on an NPL program is relatively straightforward to monitor when testing (external subroutines are not permitted to examine or modify any variables in the NPL environment except those specified as parameters to the routine).

The disadvantages are that:

- Variable numbers and types of parameters are not permitted.
- The class of parameters (input or output, or both) is not enforced by NPL.
- Although passing of alpha-arrays, of elements of alpha-arrays, and of substrings (STR()) is supported, the external routine is not able to infer anything about the number or size of elements in an alpha-array. If this information is required by the external routine, it must be determined by the NPL program and passed as a separate value.

Errors which are detected in an external subroutine may be reported and trapped (if they are in the appropriate range) with the :ERROR clause or the ON ERROR statement. However, the specific location and reason for the error is not available to the NPL program.

16.2.3 Using the FUNCTION/PROCEDURE Interface

Under Release IV or later, external functions can also be accessed using the same mechanism as FUNCTIONs or PROCEDUREs. The FUNCTION/PROCEDURE interface allows the use of new capabilities not available in the DEFFN' interface, such as new parameter types, increased error checking and return values.

NOTE: Whenever the term "function" is used in this chapter, either a FUNCTION or a PROCEDURE is allowed.

The stricter declaration requirements and restrictions enforced when using FUNCTION/PROCEDURE interface should make writing modularized, error-free programs easier.

For example, an NPL program containing a statement "GOSUB'250" when there is no defined "DEFFN' 250" or for which the parameter types were wrong is only flagged as an error when, and if, it is executed. By contrast, an NPL program containing a statement "'Initialize" does not start execution unless a declared "PROCEDURE 'Initialize" precedes it, and all parameter types are correct.

If a program contains a declaration of a function with the /EXTERNAL keyword, this indicates that the named function is not written in NPL, but should be located in an external library. Like a /FORWARD declaration, an /EXTERNAL declaration identifies the parameter names and type used to access the function, and indicates that the body of the function does not follow immediately.

When a program containing any /EXTERNAL function declaration is run, NPL verifies that the named function is present and checks that the number of parameters and parameter types match before execution starts. When a function is called, the call is directed to the external library.

NOTE: Calls are directed to the external functions only if the function declaration is marked as /EXTERNAL. Normally, external libraries which use the FUNCTION/PROCEDURE interface provide a set of compatible declarations as a module which should be INCLUDEd by applications that need them.

For example:

Interface module (provided with external library):

```
0000;Module MYMODULE
    : PUBLIC  :;external functions are public
0100 PROCEDURE 'UpperCase(/POINTER _Instring$,
               /POINTER Output$,
               /POINTER Count)/EXTERNAL
  : END PUBLIC
```

An NPL program that uses the externals:

```
0000 ;Program MYSTART
    : INCLUDE T"MYMODULE"          :;external procedure declarations
    : DIM X$100  :;variable receives the result
    : DIM X      :;variable receives count of characters that changed
    :  'UpperCase(The quick brown fox jumps over the lazy dog.",X$,X)
    : END
```

A local (non-PUBLIC) function declaration in a module can take precedence over a /PUBLIC declaration of an /EXTERNAL function, following the same rules as for other /PUBLIC function declarations. This allows new functions to be added to a library without worrying about conflicts with similarly named non-PUBLIC functions that may already exist in programs that already INCLUDE an older version of the library. To avoid possible conflicts with other functions, identifiers in a library which are not for internal use only, should be prefixed with a unique identifier (e.g., short form of company name).

Parameters of the same type permitted when calling FUNCTIONs or PROCEDUREs written in NPL are allowed when calling the external functions. As with functions written in NPL, non-/POINTER parameters are passed by value, and /POINTER parameters are passed by reference. The same restrictions on arguments passed to functions written in NPL apply to /EXTERNAL functions. In particular:

- The number of parameters must be fixed.
- The types of each parameter must be specified in the declaration.
- Constants and expressions may not be passed to /POINTER parameters, unless the parameters are constant identifiers.

An error occurs before execution starts if a directly-called function's parameters do not match. An error occurs at execution time if an indirectly-called function's parameters do not match.

External functions are not permitted to examine or modify any variables in the NPL environment except those passed as /POINTER parameters with non-constant identifiers.

16.2.4 Callbacks to /PUBLIC FUNCTIONs from the External Library

The external library is permitted to call NPL functions using a similar mechanism to the indirect function call. As is the case with the indirect function call executed by a function written in NPL, only functions which are designated as /PUBLIC may be called in this way, to ensure that the named function is not ambiguous.

Methods are provided to the external functions which permit them to check the parameter types of the named function, and this should always be done in order to ensure that parameters are passed correctly.

16.3 External Subroutine Interface Specifications

NPL external subroutines can be written in a number of languages. Each of these languages may have variations depending on the source vendor, operating system and environment used. For simplicity, the discussion here concerns the use of Microsoft C in an MS-DOS environment. Other languages and environments are discussed in Chapter 11 of the appropriate NPL Supplements and their supporting Addenda.

NOTE: **All Release IV enhancements (i.e., callbacks, etc.) are only supported under C. Previous release features supported by other external languages are fully upwardly compatible with Release IV, but new Release IV features are not downwardly compatible.**

A number of "include" files containing C structure and type definitions are provided with the BESDK. The listings of these files are shown later in this section.

First, consideration will be given to the basic building block of the external subroutine library--a single routine which is called from NPL. The specifications for this subroutine can be divided into four topics: the calling convention, the parameter format, execution of the subroutine and return value.

16.3.1 Calling Conventions

The following section discusses the calling conventions used with NPL mixed language programming.

Near Versus Far Pointers

The segmented architecture of Intel 80x86 class processors requires that a distinction be made between two types of addresses--near and far. Near addresses are normally used only for small programs, whose total program area or data area (including all library information) is less than 64K. Far addresses are used in general systems programming and do not have this restriction.

NPL calls to external subroutines always use far addresses for both code and data.

The easiest way to ensure that far addresses are used in Microsoft C is to specify use of "large model" ("/AL" option) to the compiler.

NOTE: It may also be possible for experienced programmers to use other models, provided all pointers used in connection with NPL are specifically designated as "far". However, use of other models has not been tested and is not recommended.

C Versus Pascal Calling Convention

All Microsoft languages pass parameters on the system stack. Whenever parameters are passed using a stack, a convention must be adopted as to the order in which the parameters are pushed onto the stack.

C programs usually push parameters in the reverse order from the order that they appear in a program because this allows the same interface to be used by subroutines with either a variable or a fixed number of parameters, without requiring information about the number of parameters to be passed with each call. Parameters on the stack are popped by the calling program after the call. Microsoft languages refer to this as the "C calling convention".

Most other languages push parameters in the order that they appear in a program because this simplifies compiler design (parameters can be compiled one at a time as they are parsed, while C convention requires parsing all parameters before producing any code). However, it also means that the interface to a subroutine with a variable number of parameters must be different from that used to access subroutines with a fixed number of parameters, or else all calls must include information about the number of parameters to be passed with each call. (Normally, languages that use this calling convention do not support a variable number of parameters.) Parameters on the stack are popped by the subroutine when it returns from the call. Microsoft languages refer to this as the "Pascal calling convention".

NPL calls to external subroutines use the Pascal calling convention. That is, parameters are pushed on the stack in the order they are specified in the GOSUB' statement. However, the called subroutine is allowed the option of not popping parameters from the stack.

The simplest way to ensure that the Pascal calling convention is used by a subroutine is to use the "Pascal" keyword on subroutine declarations and all external references. This ensures that the correct calling convention (and the recommended procedure for Microsoft C) is adopted.

An alternative is to use the C calling conventions on the subroutine declaration, but to list the parameters in the reverse order from the way in which they would normally be listed (if Pascal calling conventions were used).

NOTE: Be sure to use the same subroutine declaration in all modules. If a subroutine is called using the wrong calling conventions, a program crash is the likely result, unless there are no parameters.

Since not all C compilers support the Microsoft extension keywords "far", "pascal" and "cdecl", the include files provided with the BESDK avoid use of these keywords where possible. The "rtpdeffn.h" include file is normally reserved to provide macro-equivalences for these keywords (using #define directives) and type combinations which depend on these keywords. For Microsoft C, the following equivalences are set:

#define rtpdeffn_ext far pascal
#define rtpdeffn_ptr far pascal

Ensure that the appropriate "rtpdeffn.h" include file for the C compiler is available in the current directory or on one of the alternate search paths before doing compiles which reference other BESDK include files.

16.3.2 Parameter Format

The following section discusses the parameter formats.

Numeric Parameters

Numeric parameters are passed as a (far) pointer to a structure in the following format:

```
typedef unsigned short int word;
typedef short int sword;                        /* (signed) */

struct rtpnum{
    sword rtpnum_exponent;                      /* power of 10 */
    word  rtpnum_low;                           /* LS part of mantissa */
    word  rtpnum_mid;                           /* mid part of mantissa */
    sword rtpnum_high;                          /* MS part of mantissa */
};
typedef struct rtpnum * rtpnum_pointer;
```

The low, mid and high fields of the rtpnum structure form a 6-byte (48-bit) signed (two’s complement) binary integer, which contains the mantissa (M) of the number. The exponent field is an exponent (E) in base 10. The number represented is simply M*(10^E).

This is the hybrid internal format used by NPL, which has the advantage that rounding errors can be handled in the same way as a BCD representation, while the most common arithmetic operations can be quickly performed (provided numbers have the same exponent).

For example:

```
GOSUB'100(A)
```

expects a subroutine header in the format:

```
int rtpdeffn_ext mysub(ptr)
rtpnum_pointer ptr;
```

The disadvantage of this format is that it is not the same as either float or double forms used by C and is not easily manipulated as an integer or long integer unless the exponent is 0 and the precision is less than 32 bits.

NOTE: An exponent of zero cannot be assumed whenever the numeric variable is the result of a calculation or conversion statement. Even use of the INT function does not guarantee an exponent of 0. Only use of non-exponential format numeric-constants or numeric-variables directly assigned from a non-exponential numeric-constant safely assures a zero exponent.

Because of these issues, it is recommended that:

- Use of numeric-parameters be restricted to positive values in the range 0-65535 where the value is a non-exponential constant or is a variable directly assigned as a non-exponential constant. In this case, the rtpnum_low value can be directly accessed by the external routine as an unsigned integer.
- The external routine should still check the exponent of numeric values passed to it to prevent possible errors.
- For numeric values that do not conform to this format, conversion to floating point formats is required. This can be performed by the NPL floating point formats for $PACK/$UNPACK.

NOTE: The $PACK/$UNPACK conversion to and from floating point formats is a relatively slow operation. For calculation-intensive routines where performance is an issue, the NPL program and the external routines should be structured to minimize the number of floating point conversions required. Refer to $PACK/$UNPACK for details.

For example:

```
10 FOR X=1 TO 100
20 GOSUB '100(X)
```

X can safely be treated as an unsigned integer in external routine '100.

```
10 A=X*2
20 GOSUB '100(A)
```

In this case, A may have a non-zero exponent and should not be assumed to conform to integer format by the external routine. This same logic using floating point $PACK would look like:

```
10 A=X*2
20 $PACK(F=HEX(F308)) A$ FROM A: REM Intel double precision floating point
30 GOSUB '100(A$)
40 $UNPACK (F=HEX(F308) A$ TO A
```

String Parameters

String parameters are passed as two C parameters, a (far) pointer to the first byte of the string and a length in bytes (maximum value 64K-1, unsigned).

For example:

```
GOSUB'100(A$)
```

expects a subroutine header in the format:

```
typedef byte unsigned char;
typedef byte * rtpstr_pointer;
typedef unsigned short int rtpstr_length;

    int rtpdeffn_ext mysub(ptr,len)
    rtpstr_pointer ptr;
    rtpstr_length len;
```

NOTE: NPL strings are not in standard C format (C strings normally contain a number of non-HEX(00) bytes terminated by a single HEX(00)), so operations which expect C format may need to make a copy of the string.

16.3.3 Execution of the Subroutine

This area can contain whatever is required, including calls to other subroutines, etc. However, some cautionary notes apply.

It is always a good idea to do some parameter checking in the subroutines. Unlike NPL, most compiled languages stress speed over safety, and features such as subscript checking (especially as applied to strings) are either unavailable or must be specifically enabled. When inserting code to validate parameters, it is faster and more reliable to place this in the subroutine rather than in the calling NPL program.

Checking for potentially fatal errors, such as integer divide by zero, is vital to ensure a safe operating environment for NPL. The subroutines should always return to NPL if possible, even if a fatal condition is detected, since in some environments NPL may have redirected hardware interrupts and must be given a chance to clean up before terminating the process.

Under no circumstances should the subroutine attempt to save addresses of parameters and use them in future calls to the same or other subroutines. All NPL variable and program addresses are dynamic and may only be assumed to be valid for the duration of the subroutine call.

16.3.4 Return Value of the Subroutine

All external subroutines should be declared as a function returning an integer. The value of this integer should be 0 for a successful call (no NPL error required) or else the value of an error to be reported to the NPL environment. The value of the error code should be chosen from the list of standard errors to reflect as closely as possible the cause of the problem.

NOTE: Errors 1-9 are normally fatal and may not be caught even by the general purpose ON ERROR statement. Errors 10 to 59 (excluding 37 & 48) are normally fatal and may not be caught by an :ERROR clause, but may be caught by an ON ERROR statement.

Extended error values in the range 500-999 are reserved for use by external subroutines when no NPL error message is appropriate. The system error message files (ERRORMSG.HLP and ERRORMSG.IDX) may be extended to include descriptions of these errors. Error codes 500-599 are fatal and may not be caught even by an ON ERROR, while those in the 600-799 range may be caught by an :ERROR clause or ON ERROR statement.

16.3.5 Example of External Subroutine

The following C program might be used to implement the example routine, which converts a string to upper case and counts the number of lower case characters changed.

```
/* module name = mysub.c */

#include "rtpall.h"

int rtpdeffn_ext mysub(pIn,szIn,pOut,szOut,pNum)
rtpstr_pointer pIn;
rtpstr_length szIn;
rtpstr_pointer pOut;
rtpstr_length szOut;
rtpnum_pointer pNum;
{
    register rtpstr_length i;
    rtpstr_pointer p;
    rtpstr_pointer q;
    word r;
    byte c;
    p=pIn;

    q=pOut;

    r=0;
    for(i=0;i<szIn && i <szOut;i++){  /* copy and translate to min length */
        c= *p++;
        if( c>='a' && c<='z' ){
           c -= ('a'-'A');
           ++r;
        };
        *q++ = c;
    };
    for(i=szIn;i<szOut;i++){        /* if in string smaller, pad output */
        *q++ = ' ';                       /* with spaces */
    };
    pNum->rtpnum_exponent=0;        /* return number of changes made */
    pNum->rtpnum_low=r;
    pNum->rtpnum_mid=0;
    pNum->rtpnum_high=0;
    return(0);                                    /* no errors */
};
```

16.4 External Function Interface Specifications

The interface routines from the NPL Function Interface to external functions can only be written in the C language. The C language may have variations depending on the source vendor, operating system and environment used. The discussion here concerns the use of Microsoft C, in an MS-DOS environment. Other environments are discussed in Chapter 11 of the appropriate NPL Supplement and in their NPL Addenda.

NOTE: The restriction to C language is a requirement when using Release IV Function Interface. Refer to Section 16.3 for use of the DEFFN' interface found in previous versions of NPL.

A number of "include" files containing C structure and type definitions are provided with the BESDK. The listings of these files are shown later in this Section.

First, consideration must be given to the basic building block of the external function library--a single function to be called from NPL. The specifications for this function can be divided into four topics: the calling convention, the parameter format, execution of the function and return value.

16.4.1 Calling Conventions

The following section discusses calling conventions used in NPL mixed language programming.

Near Versus Far Pointers

The segmented architecture of Intel 80X86 class processors requires that a distinction be made between two types of addresses--near and far. Near addresses are normally used only for small programs, whose total program area or data area (including all library information) is less than 64K. Far addresses are used in general systems programming and do not have this restriction.

NPL calls to external functions always use far addresses for both code and data.

The easiest way to ensure that far addresses are used in Microsoft C is to specify use of "large model" ("/AL" option) to the compiler.

NOTE: It may also be possible for experienced programmers to use other models, provided all pointers used in connection with NPL are specifically designated as "far". However, use of other models has not been tested and is not recommended.

C Versus Pascal Calling Convention

All Microsoft languages pass parameters on the system stack. Whenever parameters are passed using a stack, a convention must be adopted as to the order in which the parameters are pushed on to the stack. NPL calls to external functions are all designed so that only one parameter is used. The called subroutine is allowed the option of not popping this parameter from the stack. Thus, either of the Microsoft calling conventions (cdecl or pascal) may be used by called procedures.

16.4.2 Parameter Format

The following section discusses parameter formats used in NPL mixed language programming.

Base Data Types

The basic data types passed to external functions are either numeric or string.

Numeric Data Type

Numeric scalar parameters passed by value are represented as a structure in the following format:

```
typedef unsigned short int word;
typedef short int sword;          /* signed */

struct rtpnum{
   swrd rtpnum_exponent;   /* power of 10 */
   word rtpnum_low;        /* LS part of mantissa */
   word rtpnum_mid;        /* mid part of mantissa */
   sword rtpnum_high;      /* MS part of mantissa */
};
typedef struct rtpnum * rtpnum_pointer;
```

The low, mid and high fields of the rtpnum structure form a six-byte (48-bit) signed (two’s complement) binary integer, which contains the mantissa (M) of the number. The exponent field is an exponent (E) in base 10. The number represented is simply M*(10^E).

This is the hybrid internal format used by NPL, which has the advantage that rounding errors can be handled in the same way as a BCD representation, while the most common arithmetic operations can be quickly performed (provided numbers have the same exponent).

The disadvantage of this format is that it is not the same as either float or double forms used by C and is not easily manipulated as an integer or long integer unless the exponent is 0 and the precision is less than 32 bits.

NOTE: An exponent of zero cannot be assumed whenever the numeric variable is the result of a calculation or conversion statement. Even use of the INT function does not guarantee an exponent of 0. Only use of non-exponential format numeric-constants or numeric-variables directly assigned from a non-exponential numeric-constant safely assures a zero exponent.

Because of these issues, it is recommended that:

- Use of numeric-parameters should be restricted to positive values in the range 0-65535 where the value is a non-exponential constant or is a variable directly assigned as a non-exponential constant. In this case, the rtpnum_low value can be directly accessed by the external routine as an unsigned integer.
- The external routine should still check the exponent of numeric values passed to it to prevent possible errors.
- For numeric values that do not conform to this format, conversion to floating point formats is required. This can be performed by the NPL floating point formats for $PACK/$UNPACK.

NOTE: The $PACK/$UNPACK conversion to and from floating point formats is a relatively slow operation. For calculation-intensive routines where performance is an issue, the NPL program and the external routines should be structured to minimize the number of floating point conversions required. Refer to $PACK/$UNPACK for details.

For example:

```
10 FOR X=1 TO 100
20 'MyProcedure(A)
```

X can safely be treated as an unsigned integer in external routine 'MyProcedure.

```
10 A=X*2
20 'MyProcedure(A)
```

In this case, A may have a non-zero exponent and should not be assumed to conform to integer format by the external routine. This same logic using floating point $PACK would look like:

```
10 A=X*2
20 $PACK(F=HEX(F308)) A$ FROM A: REM Intel double precision floating point
30 'MyProcedure(A$)
40 $UNPACK (F=HEX(F308) A$ TO A
```

String Data Type

String values are represented as an array of unsigned characters.

NOTE: NPL strings are not in standard C format. C strings normally contain a number of non-HEX(00) bytes terminated by a single HEX(00), so operations which expect C format may need to make a copy of the string.

Arrays of Base Data Types

Arrays of numerics and strings are represented as repeated copies of the base data type, stored in consecutive locations in memory.

Parameters Passed by Reference

A /POINTER parameter is represented by a pointer to a descriptor, whose format depends on the base data type. The descriptor will always contain the address of the referenced parameter, but may also include string length (if a string type) and array size (if an array).

16.4.3 Declaring the Parameter Block

All external functions called from NPL receive a single C parameter. This parameter is the address of a structure which contains:

- An interface area, which is passed containing NULL(0) values and should return with the address and length of the result value of FUNCTIONs.
- A parameter area for each parameter passed to the function.

Part of the C function interface definition should be a parameter block structure declaration, in the form:

```
typedef tag[FUNCTION_NAME]_PARMS{
   RTPFN_RETURN_VALUE(return_value);

   RTP{NUM}[_POINTER][_CONSTANT][_ARRAY](name1[,size][,length]);
      {STR}

   RTP{NUM}[_POINTER][_CONSTANT][_ARRAY](name2[,size][,length]);
      {STR}
   ...
}[FUNCTION_NAME]_PARMS;
```

Where:

"FUNCTION_NAME" is normally derived from the name of the called function.

RTPFN_RETURN_VALUE(x) is a macro which declares a structure member of the appropriate type for the return value/interface area. This member is defined for both FUNCTIONs and PROCEDUREs.

"return_value" is the member name of the return value/interface member.

RTP{NUM/STR}[_POINTER][_CONSTANT][_ARRAY](name,[,size],[length]) is a macro which declares a structure member of the appropriate type for a parameter, giving it the member name "name".

The exact type of the parameter dictates which of the optional bracketed sections should be used for that parameter, and they must appear in the order shown (with no spaces between parts). All numeric parameters use "NUM", and all strings use "STR". All values which are passed by reference (/POINTER) must use "_POINTER". Parameters which are not changed (and are indicated as such in the NPL function prototype by a "_" preceding the identifier) use "_CONSTANT". Arrays use "_ARRAY". An array which is passed by value must specify the "size" parameter'. A string or string array which is passed by value must specify the "length" parameter.

For example, a function with the NPL function declaration:

```
FUNCTION 'UpperCaseCounter (/POINTER _Instr$,Options,/POINTER Outstr$)
```

would require a parameter block declaration such as:

```
typedef tagUPPERCASECOUNT_PARMS{
     RTPFN_RETURN_VALUE(return_value);
     RTPSTR_POINTER_CONSTANT(instr);
     RTPNUM(options);
     RTPSTR_POINTER(outstr);
}UPPERCASECOUNT_PARMS;
```

The macros for defining parameters are located in the rtpall.h include file.

In order to avoid alignment discrepancies between NPL and the C compiler programs, Niakwa recommends that string parameters passed by value be allocated in multiples of four bytes. The total size of string array parameters passed by value should also be allocated so that the total size is a multiple of four bytes.

For example:

```
    RTPSTR(oddlength,13);    /* an odd length is not recommended! */
/* padding may be added after oddlength by NPL, C compiler or both*/
     RTPNUM(nextparm);      /* to ensure proper alignment of nextparm*/
     /* not recommended*/
```

16.4.4 Declaring the Function

A function header for an NPL-callable external function should be declared using the RTPFN_DECLARE_EXTERNAL(function-name,parameter-name) macro.

This declares the function with the appropriate interface, having one parameter with the given name.

16.4.5 Accessing the Parameter Block

The parameter passed to the NPL-callable external functions is not directly usable as a pointer to the parameter block structure.

The macro RTPFN_PARAM_BASE(pparam,parameter-structure-name) must be used to evaluate the base address of the parameter block structure. The macro must be given the name of the parameter passed to the function (pparam), and the type of the parameter block structure; it returns a pointer to the appropriate type.

For example:

```
RTPFN_DECLARE_EXTERNAL(UpperCaseCount,pparam)   /* declare the function */
{
     UPPERCASECOUNT_PARMS *z; /* declare a pointer to param block */
     /* get the base address of the parameter block */
     z=RTPFN_PARAM_BASE(pparam,UPPERCASECOUNT_PARMS);
```

Accessing Parameters Passed by Value

Parameters passed by value may be referenced as structure members, once the base pointer of the parameter block is known.

This base address can then be combined with the member name representing a structure to refer to the parameter.

For example:

```
RTPFN_DECLARE_EXTERNAL(UpperCaseCount,pparam)    /* declare the function */
{
     UPPERCASECOUNT_PARMS *z;  /* declare a pointer to param block */
     /* get the base address of the parameter block */
     z=RTPFN_PARAM_BASE(pparam,UPPERCASECOUNT_PARMS);
     struct rtpnum copy_of_opts;
     copy_of_opts= z->Options;     /* make copy of parameter */
```

Accessing Parameters Passed by Reference (/POINTER)

The pointer, length and/or size information from a parameter passed by reference can be obtained using a macro of the form:

```
     RTP{NUM}_POINTER[_ARRAY]{_ADDRESS}(z->param)
        {STR}                {_LENGTH }
                             {_SIZE   }
```

Where:

- "NUM" is used if the base parameter type is numeric, "STR" is used if the base parameter is string.
- "_ARRAY" must be used if the parameter is any kind of array.
- "_ADDRESS" is used to obtain the address of the parameter.
- "_LENGTH" is used to obtain the length of the string scalar parameter, or the element length of a string array.

- "_SIZE" is used to obtain the size (number of elements) of an array.
- z->param specifies the value of the pointer to the parameter descriptor.

For example:

```
rtpstr_pointer p;
rtpstr_len len;

p=RTPSTR_POINTER_ADDRESS(z->instr);     /* get Instr$ address */
len=RTPSTR_POINTER_LENGTH(z->instr);    /* get Instr$ length */
```

16.4.6 Execution of the Function

This area can contain whatever is required, including calls to other functions, etc. However, some cautionary rules apply.

It is always a good idea to do some parameter checking in the functions. Unlike NPL, most compiled languages stress speed over safety, and features such as subscript checking (especially as applied to strings) are either unavailable or must be specifically enabled. Time and code space expended to ensure validity of parameters will be much faster and more reliable if placed in the function than if placed in the calling NPL program.

Checking for potentially fatal errors (such as integer divide by zero) is vital to ensure a safe operating environment for NPL. The functions should always return to NPL if possible, even if a fatal condition is detected, since in some environments NPL may have redirected hardware interrupts and must be given a chance to clean up before terminating the process.

Under no circumstances should the function attempt to save addresses of parameters and use them in future calls to the same or other functions. All NPL variable and program addresses are dynamic and may only be assumed to be valid for the duration of the function call.

16.4.7 Return Value of the Subroutine

All external functions should be declared as a function returning an integer (the RTPFN_DECLARE_EXTERNAL macro always does this.). The value of this integer should be 0 for a successful call (no NPL error required) or else the value of an error to be reported to the NPL environment. The value of the error code should be chosen from the list of standard errors to reflect as closely as possible the cause of the problem.

NOTE: Errors 1-9 are normally fatal and may not be caught even by the general purpose ON ERROR statement. Errors 10 to 59 (excluding 37 and 48) are normally fatal and may not be caught by an :ERROR clause, but may be caught by an ON ERROR statement.

Extended error values in the range 500-999 are reserved for use by external functions when no NPL error message is appropriate. The system error message files (ERRORMSG.HLP and ERRORMSG.IDX), may be extended to include descriptions of these errors. Error codes 500-599 are fatal and may not be caught even by an ON ERROR, while those in the 600-999 range may be caught by an :ERROR clause or ON ERROR statement.

16.4.8 Example of an External Function

The following C program might be used to implement the example routine, which converts a string to uppercase and returns the number of lowercase characters changed.

```
/* file mymodule.src */

0010 PUBLIC
   : PROCEDURE 'MyProcedure (
        /POINTER _In$,
        /POINTER Out$,
        /POINTER Count)/EXTERNAL
   : END PUBLIC

/* file myproc.h */

RTPFN_DECLARE_EXTERNAL(myprocedure,pparam);

typedef struct tagMYPROCEDURE_PARMS{     /* parameter structure for
myprocedure */
   RTPFN_RETURN_VALUE(myret);      /* return value always first */
   RTPSTR_POINTER_CONSTANT(in);    /* /POINTER _In$ */
   RTPSTR_POINTER(out);          /*  /POINTER Out$ */
   RTPNUM_POINTER(count);       /* /POINTER Count */
}MYPROCEDURE_PARMS;

/*  file myproc.c */
```

```
#include "rtpall.h"
#include "myproc.h"

RTPFN_DECLARE_EXTERNAL(myprocedure,pparam)
{
/* PROCEDURE 'MyProcedure (/POINTER _In$,/POINTER Out$,/POINTER Count)
  : ;uppercase and convert, like mysub
  : DIM Index,Char$1
  : Count=0
  : FOR Index=1 TO MIN(LEN(STR(_In$)),LEN(STR(Out$))) BEGIN
  :  Char$=STR(_In$.Index,1)
  :   IF Char$>="a" AND Char$<="z"
  :    Char$=SUB "a" ADD "A"
  :    Count=Count+1
  :   END IF
  :  STR(Out$,Index,1)=Char$
  : NEXT Index
  : STR(Out$,Index+1)=" "     ;pad with spaces
  : END PROCEDURE
*/
   MYPROCEDURE_PARMS *z;
   register rtpstr_length i;
   rtpstr_pointer p;
   rtpstr_pointer q;
   rtpstr_length szIn;
   rtpstr_length szOut;
   rtpnum_pointer pNum;
   word r;
   byte c;

 /* get pointer to pparams */
   z=RTPFN_PARAM_BASE(pparam,MYPROCEDURE_PARMS);
   p=RTPSTR_POINTER_ADDRESS(z->in);
   szIn=RTPSTR_POINTER_LENGTH(z->in);
   q=RTPSTR_POINTER_ADDRESS(z->out);
   szOut=RTPSTR_POINTER_LENGTH(z->out);

   r=0;
   for(i=0;i<szIn && i <szOut;i++){ /* copy and translate to min
length */
   c= *p++;
   if( c>='a' && c<='z'  ){
        c -= ('a'-'A');
        ++r;
   };
   *q++ = c;
   };
   for(i=szIn;i<szOut;i++){

   /* if in string smaller, pad output with spaces
*/
   *q++ = ' ';
 };
```

```
   pNum=RTPNUM_POINTER_ADDRESS(z->count);
   pNum->rtpnum_exponent=0;  /* return number of changes made */
   pNum->rtpnum_low=r;
   pNum->rtpnum_mid=0;
   pNum->rtpnum_high=0;
/* no return value from procedure */
   return(0);          /* no error in function */
}
```

16.5 NPL Function Callback Interface Specifications

Versions of NPL after Release IV allow external libraries to callback to NPL functions and procedures which have been declared as PUBLIC.

Because the NPL environment can be very dynamic (overlays can cause entire functions to appear or disappear, and program modifications can change the interface to functions). Each call to an NPL function must perform using the following steps.

1. Validate that the function which is to be called exists and determine the parameters.
2. Validate the number and types of parameters.
3. Create a parameter block structure and call the NPL function callback routine.
4. Check for execution errors and extract the result value if any.

HINT: The way the external library calls back to NPL, in many was mirrors the way NPL calls the external library, except that the roles of calling program and called function are switched. If you are learning about the external interface for the first time, it is recommended that this section be skipped until you have thoroughly reviewed Section 16.7.6, "Information Requests about the External functions. This section describes the steps used to call from NPL to externals in detail.

16.5.1 Locating the Function

NPL always locates callback functions using a displayable (ASCII) string and return type. The string must contain the name of an NPL PUBLIC function or procedure of the specified type, and must match exactly (except for case).

The same structure used by NPL to query the external library about external functions (i.e., rtpfn) is used by the externals to query NPL about NPL functions. The external function must supply the name and type information in the rtpfn_name_pointer and rtpfn_name_length and rtpfn_return_type fields of this structure, and pass it to the callback information function (rtpfn_getfunctioninfo()). Refer to Section 16.7.6 for a description of valid values that may be placed in these fields.

The rtpfn_getfunctioninfo routine returns an error code if no such function exists in the workspace, or 0 if the function was successfully located. If the function is found, the function's (pseudo) address is contained in the rtpfn_pointer field, and the parameter types information in the rtpfn_number_params and rtpfn_template fields.

16.5.2 Validating the Function

Provided the rtpfn_getfunctioninfo did not return an error, the next step is to check that the parameters of the NPL function declaration are the same as those expected by the external.

In the same way that the RTPEXT function validates parameters of functions declared /EXTERNAL in NPL, before calling the NPL function the externals should check that the located callback routines' parameters agree with what is required. The same parameter information function (rtpfn_getparminfo()) that is used by RTPEXT is used to validate the parameters of the callback function. Refer to Section 16.7.7 for a discription of valid values that may be placed in these fields.

16.5.3 Calling the Function

The same macros used to define parameter blocks for other functions must be used to define a parameter block structure appropriate for called NPL functions. Parameters passed to the NPL function must be placed in the parameter block structure.

Passing Parameters by Value

If a parameter is passed by value, the value is placed directly in the appropriately named structure member.

Passing Parameters by Reference (/POINTER)

If a parameter is passed by reference (declared as /POINTER), a complete descriptor of the parameter must be allocated (separate from the parameter block structure). The appropriate predefined types defined for the /POINTER descriptors are:

RTPNUM_FRAME	for numeric scalars
RTPSTR_FRAME	for string scalars
RTPNUM_FRAME_ARRAY	for numeric arrays
RTPSTR_FRAME_ARRAY	for string arrays

Values are assigned to the descriptor using the appropriate predefined macro for the type, which is of the form:

RTPNUM_SET_FRAME(id,address)	for numeric scalars
RTPSTR_SET_FRAME(id,address,length)	for string scalars
RTPNUM_SET_FRAME_ARRAY(id,address,size)	for numeric arrays
RTPSTR_SET_FRAME_ARRAY(id,address,size,length)	for string arrays

where in each case:

- "id" is the name of the descriptor frame to be set.
- "address" is the address of the referenced value.
- ",size" must be specified if an array, and is the count of elements.
- ",length" must be specified if the parameter is a string or string array, and specifies the string length or array element length.

It is the address of this descriptor that is then placed in the appropriately named parameter block structure member.

Passing the Address of the Function to be Called

The external routines should place the (pseudo) address of the NPL function (that was returned by the rtpfn_get functioninfo() call in the parameter block, using the macro:

```
RTPFN_SET_CALLBACK_ADDRESS(id,rtpfn)
```

where:

- "id" is the name of the first parameter block structure member (RTPFN_RETURN_VALUE type).
- "rtpfn" is the name of the rtpfn structure passed to rtpfn_getfunctioninfo().

Making the Callback

Call rtpfn_callfunction(x), where x is the address of the first member (return value/interface member) of the parameter block structure. This causes NPL to execute the function and return to the external library.

16.5.4 Checking for Errors and Getting the Return Value

The return code of the NPL callback routine (rtpfn_callfunction()) is 0 if a normal return occurred, or is an NPL error code if a RETURN ERROR exit was taken.

Provided no error occurred, the return value/interface field of the parameter block structure contains the address (and length, if a string) of the result of a FUNCTION (PROCEDUREs have no result). The address and length may be obtained using the macro functions:

RTPNUM_RETURN_VALUE_ADDRESS(x)	address of a numeric result
RTPSTR_RETURN_VALUE_ADDRESS(x)	address of a string result
RTPSTR_RETURN_VALUE_LENGTH(x)	length of a string result

where "x" is the name of the first member (return value/interface member) of the parameter block structure.

NOTE: In general, the return value of NPL functions should always be treated as non-modifiable by the external library, and should never be stored for any duration. Exiting to NPL may cause this address to become invalid.

If the NPL function uses a RETURN value which is not a /STATIC variable, calling any other callback routine may cause the address to become invalid.

16.5.5 Restrictions on Callbacks

In general, NPL permits access to callback routines only while executing in an /EXTERNAL function. Consequently, attempts to call NPL at interrupt time (either to query function availability or to execute a function), return an error code indicating an illegal operation was attempted.

NOTE: Event driven environments such as Windows may also allow callbacks while executing certain NPL statements, such as KEYIN. Refer to Chapter 11 of the appropriate NPL Supplements for details.

16.5.6 Error Handling in Callbacks

External routines that use NPL callbacks should always consider the possibility that the call may return an error, even if the called routine does not contain any RETURN ERROR statements.

Any attempt to perform operations that would invalidate the execution return stack (such as CLEAR V, CLEAR N, CLEAR P, program LOAD or RUN operations, or Immediate Mode program modifications) that occurs while a callback to NPL is in progress is trapped and automatically generates the equivalent of a RETURN ERROR(251) to the external routine.

Where possible, this particular error code should be reflected back to the previous NPL call-out to the externals after any appropriate cleanup operations have been performed, in order to avoid unpredictable behavior.

16.5.7 Example of External Callback to NPL Function

```
/* file mycallbk.h */

/* NPL callback function CallBackKeyin used by mykeyin() */

/* parameter structure for CallBackKeyin */
typedef struct tagCALLBACKKEYIN_PARMS{
     RTPFN_RETURN_VALUE(myret);   /* return value always first */
     RTPSTR_POINTER(key);
}CALLBACKKEYIN_PARMS;

/* file mycallbk.c */

#include "rtpall.h"
#include "mycallbk.h"
/* mykeyin - get keyboard input using NPL callback procedure */
```

```
word keyin_error;
int mykeyin()
{

/* requires callback procedure
   PROCEDURE 'CallBackKeyin (/POINTER Key$)/PUBLIC
   :  STR(Key$,2,1)=HEX(01)   :;'special' key
   : KEYIN Key$,,30
   : STR(Key$,2,1)=HEX(00)   :;'normal' key
30 END PROCEDURE
*/
    byte key_buf[2];
   CALLBACKKEYIN_PARMS cb;     /* parameter block structure */
    char *callbackname="CallBackKeyin";/* callback function name */
    RTPFN rtpfn;          /* used to get NPL function info */
    RTPSTR_FRAME key_frame;/* descriptor for /POINTER Key$ parameter */
    word lookup_error,call_error;
    RTPFN_TEMPLATE tptr;     /* for checking parameter types */
    RTP_PARM_INFO parm;          /* for checking parameter types */

/* make sure CallBackKeyin exists and get its callback address */
    rtpfn.rtpfn_name_pointer=(rtpstr_pointer)callbackname;
    rtpfn.rtpfn_name_length=strlen(callbackname);
    rtpfn.rtpfn_return_type = RTPFN_RETURN_TYPE_VOID; /* PROCEDURE */
    lookup_error=rtpfn_getfunctioninfo(&rtpfn);
    if(lookup_error)return(0);      /* keyin failed? */

/* check interface to CallBackKeyin is correct */
    if (rtpfn.rtpfn_number_params !=1)return(0);  /* 1 parameter */
    /* check paramter types */
    tptr=rtpfn.rtpfn_template;
    rtpfn_getparminfo(&tptr,&parm);

if(parm.parmtype!=(RTPFN_PARAM_RTPSTR|RTPFN_PARAM_POINTER))return(0);
/* looks good! - set up parameters */
     RTPFN_SET_CALLBACK_ADDRESS(cb.myret,rtpfn);
    /* pass key_buf[] as pointer to RTPSTR_FRAME */
    RTPSTR_SET_FRAME(key_frame,key_buf,sizeof(key_buf));
    cb.key= &key_frame;
    call_error=rtpfn_callfunction(&cb.myret);
    if(call_error)return(0);     /* returned with error ? */

    return((key_buf[1]<<8)+key_buf[0]);
}
```

16.6 Writing a Mainline for External Subroutines

Each library of external routines must provide a mainline.

16.6.1 Why a Mainline is Needed

One of the difficulties which arises when an application is written using high-level languages is that each high-level language typically has its own requirements for startup and shutdown procedures.

For example, in Microsoft C the startup procedure saves several values (such as parameters on the command line, and the system's environment), in locations where they can be located by the system library procedures. It also makes determinations about how memory is to be configured for stack, heap and dynamic allocation areas, and otherwise prepares the basic environment which is a prerequisite for operation of the C language routines. It may also determine how floating point operations are to be performed, and what kind of processor is available, in order that routines which take advantage of optional hardware need not make these decisions each time they are called.

It is beyond the scope of this document to discuss these issues in detail, and they are mentioned mainly to explain why linking external subroutines to NPL requires that the developer provide a mainline. If NPL tried to simply call the subroutines without a developer-written mainline and appropriate startup routines, NPL would require detailed knowledge of the startup requirements of each variant of C, Pascal, etc., which is supported. Even for subroutines which do not explicitly call the C library routines, the C compiler may generate implicit calls to library routines to perform operations such as stack allocation checks, shift operations on long integers, floating point operations, etc. By requiring that the user must provide a mainline, NPL allows the user to ensure that the startup and shutdown requirements for the language in use are met.

In addition, the mainline approach allows applications to perform once-only initialization before NPL runs, and to perform cleanup work after NPL exits.

16.6.2 Calling NPL from the Mainline - The RunTime Subroutine

The mainline of the external routines calls NPL (once only) as a subroutine called RTP(). The RTP() subroutine performs a normal session of NPL operations, starting from loading the specified boot program. The RTP() subroutine returns to the mainline only when NPL would normally exit back to the operating system (either because $END is executed, or NPL was "killed" from the HELP subsystem). The RTP() subroutine may not be called twice because the subroutine is not reusable.

The RTP() Subroutine Calling Conventions

The RTP() subroutine uses the same calling conventions as a C mainline.

For Microsoft C, this means C calling conventions.

The RTP() Subroutine Parameters

The RTP() subroutine uses the same parameter conventions as a C mainline.

For Microsoft C, this means with three parameters adopted from the UNIX/Xenix conventions:

argc - the number of "words" in the command line which caused the mainline to run.

argv - a pointer to an array of pointers to C strings, each of which is a "word" from the command line.

envp - a pointer to an array of pointers to C strings, each of which describes the value of a current environment variable using the form "ENVNAME=ENVVALUE".

The parameters passed to the RTP() subroutine are ignored in currently supported (MS-DOS, SuperDOS) environments. However, if the mainline is written in C the recommended procedure is to pass the same arguments given to main() to the RTP() subroutine (without modification), since this is a requirement of other implementations. In this case, the parameters passed to the RTP() subroutine must be EXACTLY the same as those provided to the mainline.

The RTP() Subroutine Return Value

The function value returned by the RTP() subroutine is the exit code normally returned to the operating system.

Example of Mainline

For Microsoft C, a mainline routine must be called "main" and would resemble the following:

```
/* module name = mymain.c */

extern int RTP();

int main(argc,argv,envp)
int argc;
char **argv;
char **envp;
{
    int returncode;
    my_initialization();
    returncode=RTP(argc,argv,envp);
    my_cleanup();
    return(returncode);
};
my_initialization()
{
};
my_cleanup()
{
};
```

The my_initialization() and my_cleanup() routines are optional, and would be used to perform once-only setup work for the external subroutine library, and any cleanup work which might be required before exiting to the operating system.

NOTE: The return code produced by the RTP() is passed back to the operating system in this case (as the result of the main() procedure). This is the normal practice since it allows the exit condition of the RunTime with externals to be tested in batch files in the same way that the exit condition of the RunTime without externals would be used.

16.7 Writing the RTPEXT Subroutine

The following section discusses writing the RTEXT subroutine.

16.7.1 What Is the RTPEXT Subroutine?

The RTPEXT subroutine is called whenever NPL requires information about the external routines. This information includes information on a particular GOSUB' statement parameter count and type checking for external routines and descriptions for LIST' displays.

The RunTime calls RTPEXT under the following conditions:

1. A GOSUB' is executed for a subroutine that is not defined within NPL.

2. A LIST' is executed.

3. When scanning for named aliases of external DEFFN's at startup time.

4. A FUNCTION or PROCEDURE declaration with a /EXTERNAL indicator is resolved (program containing it is RUN or module containing it is INCLUDEd.)

In the case of the GOSUB', the RunTime expects RTPEXT to provide necessary information about the external routine to be used for the ' number specified. Upon return from RTPEXT, the RunTime then uses the information provided to call the correct external routine, perform parameter checking, and determine whether or not to allow for possible memory allocation by the external routine.

In the case of LIST', the RunTime expects RTPEXT to provide information about every defined external routine. This information consists of the existence of the routine, the description to be displayed by LIST', and (optionally) the number of the next external routine to search for.

In the case of the named alias scan, NPL creates a list of aliases and numeric equivalents that are referred to in the event that a named GOSUB' is executed for which a named DEFFN' is not found.

In the case of the FUNCTION or PROCEDURE declaration, the RunTime expects RTPEXT to validate specified information about the single function whose name and return type are provided, including parameter types, and to provide the external function entry point address.

The sections below provide further details on each of these parameters:

Whenever the RunTime calls RTPEXT, the RunTime always passes a request type that indicates whether information is required for FUNCTION type declarations or for external DEFFN' subroutines.

If the request is for external DEFFN' information, the RunTime also passes to RTPEXT the DEFFN' number for which it requires information (as part of the rtpdef version of the rtpreq request structure as described below). RTPEXT must contain logic to check this value and branch to the correct location in RTPEXT where the information about the specific subroutine is contained. In C, for example, this is usually accomplished by use of the "switch" and "case" statements.

If the request is for external FUNCTION or PROCEDURE information, the RunTime also passes to RTPEXT the name of the function for which it requires information (as part of the rtpfn version of the rtpreq request structure as described below). RTPEXT must contain logic to check this value and branch to the correct location in RTPEXT where the information about the specific function is contained. In C, for example, this is usually accomplished by use of "switch" and "case" statements based on return type, each of which does string comparisons looking for exact name matches.

Sections 16.7.9 and 16.7.10 below contain examples of typical RTPEXT routines. Programmers should find these examples particularly useful in understanding what RTPEXT is required to do.

16.7.2 The RTPEXT Calling Conventions

The RTPEXT subroutine may use either the C or Pascal calling conventions.

Since the routine has only one parameter, the calling convention is not important. Also, NPL makes no assumptions about whether parameters were popped by the called subroutine.

The C naming convention is assumed. On some C compilers (e.g., Microsoft) this means the name has an implied "_" prefix. Other languages in the same environment may not follow the same convention (MASM, Pascal). Depending on the environment and language preference, it may be necessary to name the routine "_RTPEXT" instead.

16.7.3 The RTPEXT Parameters

There is only one parameter passed to the RTPEXT subroutine. This is a (far) pointer to a request structure which contains several fields. Refer to the listing of "rtpall.h" which describes this structure for Microsoft C.

The initial fields of the request structure form a "request header" area (for the rtpreq structure, refer to "rtpall.h" include file). These fields are used to distinguish requests for external subroutine (DEFFN’) information and requests for external function (FUNCTION/PROCEDURE) validation, as well as for upward compatibility with future implementations of external calls. They specify the type of request structure which is being passed by the RunTime to RTPEXT on this call. This is done to allow for future expansion, should the need arise to add information to the structure or to request other information not currently required on future releases of NPL.

The final field of the request structure contains the specifications for the information being requested and locations for results to be placed. The contents may vary depending on the values in the request header.

In detail, the contents of the "rtpreq" request structure are:

rtpreq_type:

- An indicator of the type of the request structure. This value should be checked before using any fields in the rtpreq_var part of the request structure.
- If it contains a value of RTPREQ_TYPE_RTPDEF (=0), a request for information about external DEFFN’ subroutines is being made using an rtpdef structure in the rtpreq_var area.
- If it contains a value of RTPREQ_TYPE_RTPFN (=1), a request for validation of an external FUNCTION or PROCEDURE is being made using an rtpfn structure in the rtpreq_var area.
- If any other value is in this field, the RTPEXT routine should set the "RTPREQ_FLAGS_UNKNOWN_TYPE" bit in the rtpreq_flags field to 1 and return immediately.

rtpreq_length:

- An indicator of the length of the request structure. This is for future expansion purposes, should the need arise to extend a particular request structure type. Future implementations may need to check this field before providing values in the rtpreq_var part of the structure if they wish to remain compatible with older versions of NPL.

rtpreq_flags: RTPREQ_FLAGS_UNKNOWN_TYPE bit.

- This field contains a number of bit flags, which may be set to 1 by RTPEXT to indicate an error condition. In particular the RTPREQ_FLAGS_UNKNOWN_TYPE bit should be turned on if a request structure of unknown type is received. The meaning of other bits in this field depends on the request structure type and what are are reserved for future use. All bits are cleared to 0 by the Run-Time prior to any call to RTPEXT.

rtpreq_var:

- This field may contain a different structure depending on the value contained in rtpreq_type. On the current implementation, only two type of structure (named "rtpdef" and "rtpfn" in C) is defined.

16.7.4 Information Requests about External DEFFN’s

If the field rtpreq_type contains a value of RTPREQ_TYPE_RTPDEF (=0), a request for information about external DEFFN’ subroutines is being made using an "rtpdef" structure in the rtpreq_var area. Refer to the listing of "rtpall.h" which describes this structure for Microsoft C. The "rtpdef" structure contains the specifications for the DEFFN’ subroutine for which information is being requested and the locations at which the results should be placed.

Specification fields whose values are defined by NPL are:

rtpdef_number:

- The subroutine number for which information is required.

All information fields (whose values are to be provided by RTPEXT), are cleared to a default value of 0 prior to any call to RTPEXT. They are:

rtpdef_flags: RTPDEF_FLAGS_NO_SUCH bit.

- This bit should be set to 1 as an error indicator, if no subroutine in the library corresponds to the requested number. If this bit is set to 1, values in other fields are ignored, except rtpdef_next_number.

rtpdef_number_params:

- The number of NPL parameters which should appear in the GOSUB’ statement.

rtpdef_param_types:

- A field used to ensure that parameters are of the appropriate type in the GOSUB’ statement. Each bit of this 16-bit field specifies the type of one parameter. The low-order bit specifies the type of the first parameter, and the high-order bit specifies the type of the 16th parameter. A 0 bit means the parameter is numeric. A 1 bit means the parameter is a string. Unused bits should be set to 0’s. The include file ’rtpall.h’ contains equated values for the most commonly used bits of this field. If there are more than 16 parameters, the types information exceeds 16 bits and must be stored in an alternate format in the rtpdef_param_extended_types field.

rtpdef_flags: RTPDEF_FLAGS_EXTENDED_TYPES bit.

- This bit should be set to 1 if types information exceeds 16 bits. If this flag is set by the RTPEXT routine, the information in rtpdef_param types should be left as 0, and type information stored in the rtpdef_param_extended_types field instead.

rtpdef_pointer:

- A (far) pointer to the external subroutine’s entry point.

rtpdef_desc_pointer:

- A (far) pointer to a string which describes the function (for LIST ’).

rtpdef_desc_length:

- The length of the above string.

For compatibility with older releases of NPL, the external routines should check the value of rtpreq_var before setting any of the remaining values, which may not be part of the supplied structure on the older releases.

rtpdef_param_extended_types:

- A field used to ensure that parameters are of the appropriate type in the GOSUB' statement. Normally, this field is used only when types information exceeds the capacity of the rtpdef_param_types field, such as when more than 16 parameters exist. The rtpdef_flags bit RTPDEF_FLAGS_EXTENDED_TYPES must be set to 1 or this field is ignored.

- Each element of this array is an 8-bit field. Each bit of this 8-bit field specifies the type of one parameter. The low-order bit of the first element specifies the type of the first parameter, and the high-order bit specifies the type of the 8th parameter. A 0 bit means the parameter is numeric. A 1 bit means the parameter is a string. Unused bits should be set to 0's. Types for parameters 9-16 are stored in the second element, and so on. A maximum of 255 parameters may be specified. Unused elements should be left as 0's. The include file "rtpall.h" contains equated values for the eight bits of the first element. These equated values may also be used for other elements, keeping in mind that FIRST, SECOND, etc., refer to the parameter number in the set of 8.

rtpdef_flags: RTPDEF_FLAGS_NO_MALLOC bit.

This bit applies only to MS-DOS implementations. It should be set to 1 if it is known that the subroutine (and any routines it calls) DOES NOT require new dynamic memory allocation from MS-DOS. Setting the bit somewhat speeds up calls to the routine in this case. However, if the bit is set to 1, requests to MS-DOS for new memory may return indicating that no memory is available.

rtpdef_next_number:

- RTPEXT should set this field to the next higher subroutine number defined in the external library. If this information is not provided by RTPEXT, NPL must call RTPEXT for each ' subroutine number in the specified range when executing the LIST ' statement. This can require up to 65536 calls, which can cause a considerable delay. If RTPEXT sets this number, it indicates that no ' subroutines are defined between rtpdef_number and rtpdef_next_number, and so NPL can avoid doing many of these calls. This number may also be set for subroutines which are not in the library, especially for function 0. The number set need not actually be a defined subroutine number--in particular, the highest subroutine should set the value to the highest possible number, or 65535.

- If this field is not defined, the only effect is that LIST' may take a few seconds the first time it is executed as the RunTime searches for external subroutines in the range specified (possibly from '0 through '65535). This field is not used in calls to RTPEXT generated by GOSUB'.

- In the following example, rtpdef_next_number is set to 100 in case 0. This tells the RunTime to skip looking for DEFFN' 1 through 99 in the external routine when processing a LIST'. In addition, rtpdef_next_number is set to 65535 in Case 100 (for DEFFN'100). This tells the RunTime to skip looking for DEFFN's above 100 in the external routine when processing a LIST'.

For compatibility with older NPL versions, the following values should only be set if the macro RTPDEF_CAN_SET_POINTER(rtpreq) returns a TRUE value.

rtpdef_name_pointer:

- If the function has a named alias, the RunTime should set this field to point to the name of the alias. Refer to Section 16.7.5 for details.

rtpdef_name_length:

- The length of the string pointer to rtpdef_name_pointer.

16.7.5 Named Aliases of External DEFFN' Routines

As of Release IV, marked DEFFN' subroutines in NPL can be identified by either a number or an identifying name. This capability is extended to external functions in a limited way by means of aliases. The alias extension allows numbered external DEFFN's to also be referenced by an identifying name. This name (or alias) must be a legal NPL identifier (i.e., it must start with a letter and may contain a maximum of 255 characters), selected from letters, digits and underscore ("_") only. To avoid possible conflicts with other functions, identifiers in a library which is not for internal use only should be prefixed with a unique identifier (e.g., short form of company name).

The alias is returned by the RTPEXT in response to a query about a numbered DEFFN’ in the rtpdef_name_pointer and rtpdef_name_length fields (see previous section). If no value is placed in these fields, or a NULL pointer or length is set, the external DEFFN’ is not assigned a named alias. NPL scans for named DEFFN’ aliases once only, at startup time, by querying RTPEXT about external function ’0 an following the rtpdef_next_number chain values for as long as this value is an ascending series of numbers. Consequently, to make effective use of this field, rtpdef_next_number must be correct.

Once external aliases are registered in this way, the functions associated with them become permanently available to all programs, exactly as numbered functions are available.

Because of the way this extension is implemented, a named external DEFFN’ subroutine always has a numbered external DEFFN’ equivalent. Access to numbered functions using aliased external DEFFN’ names only occurs if a named DEFFN’ is called which cannot be found in the workspace, as either a local or (for revision 4) INCLUDEd PUBLIC named DEFFN’. At this point, if the name is an alias for an external DEFFN’ subroutine, the equivalent numbered external DEFFN’ is called.

This may mean that, in rare instances where a named DEFFN’ written in NPL prevents access to an external DEFFN’ with the same name, an external can be called only using its number. Similarly, in rare instances where a numbered DEFFN’ written in NPL prevents access to an external DEFFN’ with the same number, an external can be called only using its named alias. Except in these cases, calling the external by name and by number are equivalent.

Implementing named external DEFFN’s in this way allows new external libraries to be upgraded to provide named access to functions in existing libraries with a minimum of effort, since only the RTPEXT function needs to be updated, while still allowing access to the functions by older application software and versions of NPL.

16.7.6 RTPEXT Requests for Information about External FUNCTIONs

If the field rtpreq_type contains a value of RTPREQ_TYPE_RTPFN (=1), a request for validation of an external FUNCTION or PROCEDURE is being made using an "rtpfn" structure in the rtpreq_var area. Refer to the listing of "rtpall.h" which describes this structure for Microsoft C. The "rtpfn" structure contains the specifications for the FUNCTION or PROCEDURE for which validation is being requested and the locations at which the results should be placed. Specification fields, whose values are defined by NPL include:

rtpfn_name_pointer:

- A pointer to a string containing the identifying name of the subroutine. For the convenience of C, this name is terminated with a null character. The return type of the function is not indicated in this name (no "$" at the end of string-valued functions).

rtpfn_name_length:

- The length of the identifier rtpfn_name_pointer, not including the terminating null.

rtpfn_return_type:

- The type of the return value supplied by the external. This is one of the following predefined values:

RTPFN_RETURN_TYPE_RTPNUM	for numeric FUNCTIONs
RTP_RETURN_TYPE_RTPSTR	for string FUNCTIONs
RTP_RETURN_TYPE_VOID	for PROCEDUREs

rtpfn_template:

This field contains a pointer to a structure which contains information about all parameters of the function as it appears in the NPL declaration. Information about each parameter, in the order specified, can be obtained using this value.

rtpfn_number_params:

This field contains a count of the number of parameters in the function declaration.

All information fields whose values are to be provided by RTPEXT, are cleared to a default value of 0 prior to any call to RTPEXT. They are:

rtpfn_flags: RTPFN_FLAGS_NO_SUCH bit.

- This bit should be set to 1 as an error indicator, if no function or procedure in the library corresponds to the requested name. If this bit is set to 1, values in other fields are ignored.

rtpfn_flags: RTPFN_FLAGS_NO_MALLOC bit.

- This bit applies only to MS-DOS implementations. It should be set to 1 if it is known that the subroutine and any routines it calls will not require new dynamic memory allocation from MS-DOS. Setting the bit somewhat speeds up calls to the routine in this case. However, if the bit is set to 1, requests to DOS for new memory may return indicating that no memory is available.

rtpfn_pointer:

A (far) pointer to the entry point of the external FUNCTION or PROCEDURE.

16.7.7 Validating a FUNCTION/PROCEDURE

The RTPEXT should only return an entry point to the external FUNCTION or PROCEDURE if the name, return type, parameter and parameter types match.

In order to conform to other NPL routines, comparisons of the supplied name and the names known to the RTPEXT routine should ignore case differences. C library functions such as stricmp() are usually available to do this type of comparison. Validation should normally be classified first by return type, using C switch and case statements. Within each class, the name and parameter count of each function should be checked for each function. If the parameter count and name match, each parameter should also be checked for type and dimensions, if passed by value and a string or array.

Information about one parameter at a time can be obtained using the rtpfn_getparminfo(&tptr,&parm) function, where:

- tptr is an RTPFN_TEMPLATE type variable, initially set to the value of the rtpfn_template field to return the first parameter information.
- parm is a RTP_PARM_INFO type variable, which receives information about the parameter.

Each call to rtp_getparminfo returns information about one parameter in the "parm" structure, and updates the value of tptr to point to the next parameter in the template. The information in the RTP_PARM_INFO structure contains the following items:

parmtype:

- The parameter type, constructed as a sum of the following attributes:

RTPFN_PARAM_RTPNUM	numeric base data type
RTPFN_PARAM_RTPSTR	string base data type
RTPFN_PARAM_ARRAY	array parameter
RTPFN_PARAM_CONSTANT	constant type parameter
RTPFN_PARAM_POINTER	/POINTER type parameter

- Any combination may be returned, except that only one of _RTPNUM and RTPSTR is allowed.

parmsize:

- For array parameters passed by value (i.e., not /POINTER types), this value is the number of elements.

parmlength:

- For string parameters passed by value (i.e., not /POINTER types), this value is the length of a string scalar or string array element.

It is important to validate all parameters before indicating that a function exists. Failure to do so could result in the function being called with different parameter types and, consequently, a disagreement between the parameter block structure expected by the external and that passed to it by NPL. Particularly when /POINTER parameters are used, this can be fatal to the program or to the operating system.

Since the RTPEXT routine is usually only called once (when the declaration is RUN or INCLUDEd, not each time the function is referenced), speed is not essential in this routine, but accuracy is.

16.7.8 The RTPEXT Return Code

RTPEXT should return a value of 0 to indicate it has successfully provided the information requested.

Indicating that the subroutine does not exist in the provided field is considered to be a successful operation. Non-zero return codes are reserved for future implementations.

16.7.9 Example RTPEXT Subroutine

The following example is the RTPEXT routine from the MS-DOS "C" examples provided with the BESDK. It performs the following tasks:

1. Defines the name of the external module "mysub" as an external function.

2. Defines external routine '100 and provides the following information about it:

   A. Module "mysub" is the module that contains the actual routine that the RunTime should call.

   B. "Mysub" expects three parameters: parameter 1 and 2 are alpha, and parameter 3 is numeric. This is equivalent to:

   ```
   DEFFN'100(A$,B$,A)
   ```

   C. "Mysub" performs no memory allocation.

3. Defines information for use by LIST':

   A. For "mysub", defines the information line to be displayed by LIST' as "(In$, Out$, N) - In$ (upper case)-> Out$, N=#changes"

   B. Advises the RunTime that no external routines between 0 and 100 exist (in the case 0 section).

   C. Advises the RunTime that no external routines above 100 exist (in the definition of rtpdef_next_number in case 100).

4. Defines the following alias for named DEFFN's:

External DEFFN’100 can also be accessed by name using the alias DEFFN’UpperCase.

NOTE: This information is only supplied if the flag returned by the RTPDEF_CAN_SET_NAME_POINTER macro is true.

5. Validates an external PROCEDURE called "MyProcedure" with three parameters, which are, in order:

   a /POINTER to a constant string scalar (for input)
   a /POINTER to a string scalar (for output)
   a /POINTER to a numeric scalar (for output)

No external numeric or string-valued FUNCTIONs are valid with this library.

```
    /* module name = myrtpext.c */

    #include "rtpall.h"

    /* declaration of DEFFN’ function, so C knows it is a function */
    extern rtpdeffn_ptr mysub();

/* declaration of external PROCEDURE/FUNCTIONs, so C knows */
RTPFN_DECLARE_EXTERNAL(myprocedure,pparam);

int RTPEXT(req_base)
struct rtpreq * req_base;
{
    register struct rtpdef *p;
    register struct rtpfn *pf;
    int setname;
    char *fname;
    int pcount;
    RTPFN_TEMPLATE tptr;
    RTP_PARM_INFO parm;

    switch(req_base->rtpreq_type){

/**********************************************************************/


/* DEFFN’ lookups */

    case RTPREQ_TYPE_RTPDEF:
    p= &req_base->rtpreq_var.rtpreq_var_rtpdef;
/* this is a rtpdef structure */
    setname=RTPDEF_CAN_SET_NAME_POINTER(req_base);
    switch(p->rtpdef_number){
    case 100:p->rtpdef_pointer=mysub;
        p->rtpdef_number_params=3;
        p->rtpdef_param_types=RTPDEF_PARAM_FIRST_IS_STRING+
                                        RTPDEF_PARAM_SECOND_IS_STRING+
                                        RTPDEF_PARAM_THIRD_IS_NUMERIC;
```

```
        p->rtpdef_desc_pointer=(rtpstr_pointer)
           "(In$, Out$, N) - In$ (upper case)-> Out$, N=#changes";
        p->rtpdef_desc_length=strlen(p-rtpdef_desc_pointer);
        p->rtpdef_flags |= RTPDEF_FLAGS_NO_MALLOC;
        p->rtpdef_next_number=65535;             /* for LIST ' scanning */
    if(setname){          /* aliases supported by NPL version ? */
          p->rtpdef_name_pointer="UpperCase";
          p->rtpdef_name_length=strlen(p->rtpdef_name_pointer);
    };
    break;
/*
    other defined subroutines would follow a similar pattern to the
above case
 */
    case 0:
        p->rtpdef_next_number=100;               /* for LIST ' scanning */
    default:
/*
    for all unknown subroutine numbers, set the flag to tell NPL this.
 */
         p->rtpdef_flags |= RTPDEF_FLAGS_NO_SUCH;
    };
    return(0);
/***********************************************************************/


/* FUNCTION declaration certification */

    case RTPREQ_TYPE_RTPFN:
    /* this is a rtpfn structure */
    pf= &req base->rtpreq_var.rtpreq_var_rtpfn;
    fname= pf->rtpfn_name_pointer;
    pcount=pf->rtpfn_number_params;
    switch(pf->rtpfn_return_type){

    case RTPFN_RETURN_TYPE_RTPNUM:
          /* numeric FUNCTIONS */
          /* none! */
          break;
    case RTPFN_RETURN_TYPE_RTPSTR:
          /* string FUNCTIONS */
          /* none! */
          break;
    case RTPFN_RETURN_TYPE_VOID:
          /* PROCEDUREs */
          if((stricmp(fname,"MyProcedure")==0)
          && (pcount==3)
            ){
               tptr=   pf->rtpfn_template;
               rtpfn_getparminfo(&tptr,&parm);
               if(parm.parmtype!=
                   (RTPFN_PARAM_RTPSTR
                   |RTPFN_PARAM_CONSTANT
                   |RTPFN_PARAM_POINTER
                   )
               )break;
            rtpfn_getparminfo(&tptr,&parm);
            if(parm.parmtype!=
                   (RTPFN_PARAM_RTPSTR
```

```
                    |RTPFN_PARAM_POINTER
                     )
            )break;
          rtpfn_getparminfo(&tptr,&parm);
          if(parm.parmtype!=
                     (RTPFN_PARAM_RTPNUM
                     |RTPFN_PARAM_POINTER
                     )
            )break;
          pf->rtpfn_pointer=myprocedure;
            return(0);
          };
         break;
default:
          /*
            for all unknown return types, set the flag to tell NPL
            function is unknown
          */
          break;
};
/*
    for all unknown name types, set the flag to tell NPL function is
    unknown
     */
    pf->rtpfn_flags |= RTPFN_FLAGS_NO_SUCH;
    return(0);

/* for all unknown request types, set the flag to tell NPL request
type is unknown*/

    default:
    req_base->rtpreq_flags |= RTPREQ_FLAGS_UNKNOWN_TYPE;
    return(0);
    };
};
```

16.7.10 Another Example RTPEXT Subroutine

The following example shows an extension to the one above, adding a definition for a second external DEFFN’ subroutine. The new DEFFN’ routine is defined as ’101, with the alias "WindowTitle".

For clarity, only the logic related to DEFFN’s is repeated in this example. In addition, for brevity, all references to the external PROCEDURE are removed.

The name "mysub101" is defined as an external subroutine.

In the logic associated with case 101, ’101 is assigned to the module named "mysub101" and has two parameters, the first numeric and the second alpha.

In addition to adding case 101 to define "mysub101", one change was made to the definition for case 100. This is that the rtpdef_next_number value was changed from 65535 to 101. Without this change LIST' and the named alias registration pass would not find '101 since the rtpdef_next_number of 65535 in '100 would tell the RunTime that no external routines above 100 existed.

Even if rtpdef_next_number were not changed, GOSUB' would still properly locate the external '101, but would be unable to access it using the named alias "WindowTitle". The field rtpdef_next_number is used only by LIST' and by the named alias registration pass.

```
/* module name = myrtpext.c */

#include "rtpall.h"

/* declaration of function, so C knows it is a function */

extern rtpdeffn_ptr mysub();

extern rtpdeffn_ptr mysub101();

int RTPEXT(req_base)
struct rtpreq * req_base;
{

    register struct rtpdef *p;
    int setname;
    switch(req_base->rtpreq_type{

/*********************************************************************/

/* DEFFN' lookups */

    case RTPREQ_TYPE_RTPDEF:
p= &req_base->rtpreq_var.rtpreq_var_rtpdef; /* this is a rtpdef struc-
ture */
    setname=RTPDEF_CAN_SET_NAME_POINTER(req_base);
    switch(p->rtpdef_number){
    case 100:p->rtpdef_pointer=mysub;
        p->rtpdef_number_params=3;
        p->rtpdef_param_types=RTPDEF_PARAM_FIRST_IS_STRING+
                                        RTPDEF_PARAM_SECOND_IS_STRING+
                                        RTPDEF_PARAM_THIRD_IS_NUMERIC;
        p->rtpdef_desc_pointer=(rtpstr_pointer)
            "(In$, Out$, N) - In$ (upper case)-> Out$, N=#changes";
        p->rtpdef_desc_length=strlen(p->rtpdef_desc_pointer);
        p->rtpdef_flags |= RTPDEF_FLAGS_NO_MALLOC;
        p->rtpdef_next_number=101;              /* for LIST ' scanning */
        if(setname){          /* aliases supported by NPL version ? */
          p->rtpdef_name_pointer="UpperCase";
          p->rtpdef_name_length=strlen(p->rtpdef_name_pointer);
   };
break;
```

```
        case 101:p->rtpdef_pointer=mysub101;
            p->rtpdef_number_params=2;
            p->rtpdef_param_types=RTPDEF_PARAM_FIRST_IS_NUMERIC+
                                        RTPDEF_PARAM_SECOND_IS_STRING;
            p->rtpdef_desc_pointer=(rtpstr_pointer)
               "This is '101 - (InN, Out$)";
            p->rtpdef_desc_length=strlen(p->rtpdef_desc_pointer);
            p->rtpdef_flags |= RTPDEF_FLAGS_NO_MALLOC;
            p->rtpdef_next_number=65535;            /* for LIST ' scanning */
            if(setname){      /* aliases supported by NPL version ? */
              p->rtpdef_name_pointer="WindowTitle";
              p->rtpdef_name_length=strlen(p->rtpdef_name_pointer);
       };
        break;
    /*
        other defined subroutines would follow a similar pattern to the
    above case
    */
        case 0:
        p->rtpdef_next_number=100;    /* for LIST ' scanning */
        default:
    /*
        for all unknown subroutine numbers, set the flag to tell NPL this.
     */
             p->rtpdef_flags |= RTPDEF_FLAGS_NO_SUCH;
        };
        return(0);
    };
    /*
       for all unknown request types, set the flag to tell NPL request
        type is unknown
    /*
       default:
         req_base->rtpreq_flags |=RTPREQ_FLAGS_UNKNOWN_TYPE;
       return(0);
       };
    };
```

16.8 Writing a Test Program

Before attempting to call external subroutines from NPL, it is a good idea to write a test program which calls the various subroutines with test values and verifies the results. Having a test program available allows the user to verify that the subroutines work properly and link with the appropriate support libraries before interactions with NPL are considered. In addition, it is generally simpler to compile and link the test program without NPL, and all the normal debugging tools available to the compiled language can be used in the test program to locate problems in the subroutines themselves.

The easiest way to write the test program is to set it up as a replacement for the RunTime subroutine called from the mainline, defined in a separately compiled module.

This technique can also be applied to external FUNCTION and PROCEDURE calls, providing callback functions to NPL are not used. A couple of predefined types and macros are defined specifically to allow testing to use the same call procedure as NPL:

- RTP_PARAM_FRAME is a descriptor type that contains the address of the parameter structure block p in the RTP_PARAM_FRAME variable "frame".
- RTP_PARAM_SET_FRAME(frame,param_frame) places the address of the parameter structure block p into the RTP_PARAM_FRAME variable "frame".

16.8.1 Example Test Program

```
/* module name = myrtp.c */

#include "rtpall.h"

/* declaration of function, so C knows it is a function */
extern rtpdeffn_ext mysub(rtpstr_pointer,rtpstr_length,
                        rtpstr_pointer,rtpstr_length,
                        rtpnum_pointer);


/* NPL 'variables' used for testing */

byte A[]  ="The quick brown fox jumps over the lazy dog.";
byte B[]  ="xxxx.xxxx1xxxx.xxxx2xxxx.xxxx3xxxx.xxxx4xxxx.xxxx5xxxx";
struct rtpnum X = {1,2,3,4};
byte C[] ="The quick brown fox jumps over the lazy dog.";
byte D[] ="xxxx.xxxx1xxxx.xxxx2xxxx.xxxx3xxxx.xxxx4xxxx.xxxx5xxxx";
struct rtpnum Y = {1,2,3,4};
int RTP(argc,argv,envp)
int argc;
char **argv;
char **envp;
{
    int returncode;
    RTP_PARAM_FRAME param_frame;
    RTPSTR_FRAME in_frame;
    RTPSTR_FRAME out_frame;
    RTPNUM_FRAME count_frame;
    MYPROCEDURE_PARM myprocedure_parms;

/* test DEFFN' routine */
    printf("test GOSUB' call\n");
    printf("A$=\"%s\"\n",A);
    printf("B$=\"%s\"\n",B);
    printf("X=%d %d %d E %d\n",X.rtpnum_high,
                                    X.rtpnum_mid,
```

```
                                          X.rtpnum_low,
                                          X.rtpnum_exponent);
     printf("GOSUB'UpperCase(A$,B$,X) ->mysub()\n");
     returncode=mysub(A,sizeof(A) -1,  /* note size not including
     ASCIIZ 0 */
           B,sizeof(B)-1, /* & operator on A, B is implied */
                 &X;
     printf("returncode=%d\n",returncode);
     printf("A$=\"%s\"\n",A);
     printf("B$=\"%s\"\n",B);
     printf("X=%d %d %d E %d\n",X.rtpnum_high,
                           X.rtpnum_mid,
                           X.rtpnum_low,
                           X.rtpnum_exponent);

    /* test PROCEDURE call */
    /* set pointer parameters */
    RTPSTR_SET_FRAME(in_frame,C,sizeof(C)-1);
    myprocedure_parms.in= &in_frame;
    RTPSTR_SET_FRAME(out_frame,D,sizeof(D)-1);
    myprocedure_parms.out= &out_frame;
    RTPNUM_SET_FRAME(count_frame,&Y)
    myprocedure_parms.count=&count_frame;
    /* pass address of parameter block structure */
    RTP_PARAM_SET_FRAME(param_frame,&myprocedure_parms);

    printf("test PROCEDURE call \n");
    printf("C$=\"%s\"\n",C);  printf("D$=\"%s\"\n",D);
    printf("Y=%d %d %d E %d\n",Y.rtpnum_high,
                Y.rtpnum_mid,
             Y.rtpnum_low,
             Y.rtpnum_exponent);
    printf("'MyProcedure(C$,D$,Y) ->myprocedure()\n");
    returncode=myprocedure(&param_frame);

    printf("returncode=%d\n",returncode);
    printf("C$=\"%s\"\n",C);  printf("D$=\"%s\"\n",D);
    printf("Y=%d %d %d E %d\n",Y.rtpnum_high,
                Y.rtpnum_mid,
             Y.rtpnum_low,
             Y.rtpnum_exponent);

    return(0);
};
```

The test program prints the value of variables both before and after the call. The sizes of strings are chosen to ensure that the code for padding strings is tested. A more extensive test program might also check that, if B$ is shorter than A$, the output string is properly truncated.

16.8.2 Example Linkage for Test Program

The following MS-DOS batch file compiles and links the test program, assuming appropriate values for the environment variables INCLUDE (location of C include files), LIB (location of C support libraries), CL (default options for C compiler) and LINK (default options for linker) are set to appropriate values. The include files which are part of the BESDK are assumed to be in the directory ..\include.

```
rem   /* this is the makemain.bat file */
copy ..\include\dos\rtpdeffn.h .
copy ..\include\dos\rtpparm.obj .
cl /AL /c /I..\include mymain.c myrtp.c myrtpext.c mysub.c myproc.c
link /MAP mymain+myrtp+myrtpext+mysub+myproc+rtpparm;
```

Although the test program does not use the RTPEXT subroutine, it is a good idea to include it in the standalone program to ensure that any support routines it may need are available from the support libraries. Part of this support includes the parameter type checking function 'rtpfn_getparminfo' which is located in the 'rtpparm.obj' object file.

16.8.3 Running the Test Program

The resulting executable program may be run as a standalone by typing its name from the DOS command processor.

NOTE: RTPEXT is not tested by this procedure. Therefore, errors in RTPEXT, such as incorrect parameter lists or incorrect module names are not detected. A thorough check of RTPEXT is the best way to detect these kinds of errors in advance.

16.8.4 Running the Example Test Program

Running the example test program (mymain.exe) produces the following output:

```
C>mymain

A$="The quick brown fox jumps over the lazy dog."
B$="xxxx.xxxx1xxxx.xxxx2xxxx.xxxx3xxxx.xxxx4xxxx.xxxx5xxxx"
X=4 3 2 E 1
call mysub(A$,B$,X)
returncode=0
A$="The quick brown fox jumps over the lazy dog."
B$="THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.          "
X=0 0 34 E 0
```

```
Test PROCEDURE call

C$="The quick brown fox jumps over the lazy dog."
D$="xxxx.xxxx1xxxx.xxxx2xxxx.xxxx3xxxx.xxxx4xxxx.xxxx5xxxx"
Y=4 3 2 E 1
'MyProcedure(C$,D$,Y)   -myprocedure()
returncode=0
C$="The quick brown fox jumps over the lazy dog."
D$="xxxx.xxxx1xxxx.xxxx2xxxx.xxxx3xxxx.xxxx4xxxx.xxxx5xxxx"
Y=0 0 34 E 0
```

The programmer might also compile the program with options to keep debug information (i.e., /Zi in "cl" command for Microsoft C) to allow debugging of the program (i.e., using Codeview), especially when running the program for the first time and it is necessary to verify that the subroutines are operating as expected.

16.9 Making the External Library

Once a standalone program has been made that runs acceptably, the next step is to link the modules into a form which is useful to NPL. The format required by NPL for DOS and SuperDOS environments is called a "quick library". Unlike the .EXE files normally produced by the linker, quick libraries may not be executed as standalone programs. They are designed to be used in conjunction with interpretive languages, such as NPL.

The quick library is produced by essentially the same procedure as the standalone test file, except that the test module "myrtp.c" is omitted and a precompiled quick library header file ('qlbhead.obj') is substituted in its place. This quick library header file allows the LINK program to resolve the otherwise undefined reference to the RunTime subroutine, and satisfies some other requirements made by the LINK program and by NPL. The quick library file format contains sufficient symbol and relocation information for NPL to locate the RTPEXT subroutine.

16.9.1 Example Linkage for External Library

The following DOS batch file compiles and links the test external subroutine library, assuming appropriate values for the environment variables INCLUDE (location of C include files), LIB (location of C support libraries), CL (default options for C compiler) and LINK (default options for linker) have already been set to appropriate values. The include files which are part of the BESDK are assumed to be in the directory ..\include.

```
rem   /* this is the makeqlb.bat file */
copy ..\include\dos\rtpdeffn.h .
copy ..\include\dos\rtpparm.obj .
copy ..\include\dos\qlbhead.obj .
copy ..\include\dos\cnullchk.obj .
cl /AL /c /I..\include mymain.c myrtpext.c mysub.c myproc.c mycallbk.c
rem   /* the following line wraps in documentation */
link /MAP/QUICKLIB mymain+qlbhead+myrtpext+mysub+myproc+mycallbk+
rtpparm+cnullchk,mylib.qlb;
```

The "myrtp" module has been replaced in the link step with "qlbhead", which is the (pre-compiled) quick library header module. The /QUICKLIB option is used to instruct the linker to create a quick library format. Quick libraries should always be given the extension ".QLB". The "cnullchk.c" module is used to disable the "null pointer misuse" checking option of the Microsoft C RunTime library, which is a requirement of quick libraries.

While the above command is the simplest way to make the executable file, it does have the disadvantage that it always compiles all the modules. Making even small changes to one of the C programs and remaking the executable file involve considerable excess recompiling. The example programs also contain a "makefile" which sets the proper options for creating the Quick Library for the example external subroutines.

16.9.2 Using an External Library

In order to access the subroutines in a quick library, NPL must be started with an option specifying that an external library of subroutines and functions is required. The format of this option is:

```
RTP /X[quicklib] [bootfile]
```

Where "/X" indicates that an external library must be attached to NPL and "quicklib" is the file name of the quick library module. No spaces should appear between /X and the library name. The /X option must appear before any boot file name specification. The .QLB extension may be omitted and is assumed if the filename has no extension. If the /X option is specified without a quick library name, NPL uses the default filename "RTPXSUBS.QLB". Only one quick library may be specified under the MS-DOS. Refer to the appropriate NPL Supplement’s specific addenda for information on the operating environments.

16.9.3 Using the Example External Library

The example library can be accessed by starting RTI with the option /Xmylib. For test purposes the programmer may wish to write a "boot" program for NPL that puts the external subroutines through some test calls. For this example the following program (mystart.src) was created and compiled using B2C to produce the boot file myboot.obj:

```
boot file MYBOOT.SRC:
0000 $DEVICE(#0)="MYMODULE.BS2"
   : LOAD T"MYSTART"
```

Test program files are located in the diskimage MYMODULE.BS2:

```
0000 REM MYMODULE
   : PUBLIC             :;public external functions for MYLIB library
   : PROCEDURE 'MyProcedure(       ; translate to upper case example
                 /POINTER _In$,            ; source string
       /POINTER Out$,            ; translated output
       /POINTER Count)/EXTERNAL  : ; count of translated characters
   : END PUBLIC
   : COM StayResident : ; a common variable keeps module always loaded
   : ;provide callback procedure for external mykeyin() routine
   : PROCEDURE 'CallBackKeyin (/POINTER Key$)/PUBLIC
   : STR(Key$,2,1)=HEX(01)   :;'special' key
   : KEYIN Key$,,30
   : STR(Key$,2,1)=HEX(00)   :;'normal' key
 30 END PROCEDURE


0000 REM MYSTART
   : INCLUDE T"MYMODULE"
0010 LIST
0020 DIM B$80
   : B$="xxxx.xxxx1xxxx.xxxx2xxxx.xxxx3xxxx.xxxx4xxxx.xxxx5xxxx"
   : X=1.23456789E30
0030 GOSUB '100("The quick brown fox jumps over the lazy dog.",B$,X)
   : PRINT B$: PRINT X
0040 GOSUB 'UpperCase("I can use the equivalent name.",B$,X)
   : PRINT B$: PRINT X
0050 'MyProcedure("I can use the PROCEDURE interface.",B$,X)
   : PRINT B$: PRINT X
```

The following is a sample session with the library attached.

```
C> RTI /Xmylib myboot
-- NPL copyright message display --

READY (NPL) PARTITION 01
0000 REM MYSTART
   : INCLUDE T"MYMODULE"
0010 LIST
0020 DIM B$80
   : B$="xxxx.xxxx1xxxx.xxxx2xxxx.xxxx3xxxx.xxxx4xxxx.xxxx5xxxx"
   : X=1.23456789E30
0030 GOSUB '100("The quick brown fox jumps over the lazy dog.",B$,X)
   : PRINT B$: PRINT X
0040 GOSUB 'UpperCase("I can use the equivalent name.",B$,X)
   : PRINT B$: PRINT X
0050 'MyProcedure("I can use the PROCEDURE interface.",B$,X)
   : PRINT B$: PRINT X
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
 34
I CAN USE THE EQUIVALENT NAME.
23
I CAN USE THE PROCEDURE INTERFACE.
 18
:LIST '
/EXT DEFFN'100 (In$, Out$, N) - In$ (upper case)-> Out$, N=#changes
      - 0030
/EXT DEFFN'UpperCase
      (In$, Out$, N) - In$ (upper case)->Out$, N=#changes
      - 0040
:$END
```

16.10 Cautions and Restrictions

External subroutines represent a useful additional tool for programmers in the NPL environment, and for certain applications can provide a substantial performance benefit. But before investing large amounts of time and effort in programming external call,stop to weigh the advantages and disadvantages of making applications dependent on a particular external library. The following sections mention some areas that might not otherwise be considered.

16.10.1 Compatibility with Other NPL Applications

Currently, most implementations of NPL allow for only one external library to be attached to a running RunTime. The MS-Windows NPL RunTime allows for more than one external library to be attached. When using a version of NPL that only allows for one library, but running an application that requires the use of distinct external libraries, it would be necessary to exit NPL and return to the operating system to switch between libraries. Such a restriction can make putting a user's application on a centralized menu system problematical, if the menu is to be written in NPL.

NOTE: **Refer to the appropriate NPL Supplement's Addenda for more information on loading quick libraries.**

16.10.2 Memory Requirements

Compiled code can be surprisingly expensive in terms of consumed memory, especially if standard library functions are invoked (e.g., printf()) for simple functions. The example quick library in a MS-DOS environment reduces the space available for the RunTime task by about 6K, and this uses only the library functions required by the C startup--adding a single printf() command reduces the available space by a further 7K. Unlike subroutines written in NPL, externals are permanently resident. Since memory used by external routines is deducted from the available NPL user partition, the programmer must consider the effects of this reduced partition size, particularly on systems where the user task size available may be limited.

16.10.3 Isolation from NPL Environment

Access to NPL files, screen, keyboard, etc. is generally not available to the external routines, except through callback functions. Little allowance is made by NPL for applications that access hardware that is used by NPL--such as setting the screen into a graphic mode, for example. Consequently keyboard and display interaction with the user is best left as NPL code if at all possible.

16.10.4 Resource Conflicts

Because the externals and NPL may both make demands on the operating system for limited resources, the demands of either program may inhibit the operation of the other. The most likely areas of contention are memory and open files.

Depending on the configuration, external subroutines which make requests for additional memory from the operating system may either be denied, or if requests are granted, this memory may become permanently unavailable to the NPL environment.

Similarly, NPL typically keeps its files open for a long duration, and this may restrict applications which open many files.

Another possible area of contention concerns resources which can have only a single owner, such as access to math coprocessors and interception of interrupt and abort conditions. In cases like this, the convention is that the programmer "owns" the coprocessor or interrupt only for the duration of the subroutine, and should restore the status of the resource to its original condition, as much as possible, before returning from the subroutine.

16.10.5 Increased Expertise Requirement

Supporting an application which depends on an external subroutine library will require that someone in the programming staff maintain a level of expertise in the area of the language in which the external routines are written, as well as having a knowledge of NPL. This will usually involve a working knowledge of the support tools (compilers, linkers, libraries, debuggers) used when programming in the language for each supported environment (MS-DOS, SuperDOS, etc.), in addition to understanding the language syntax and semantics.

16.10.6 Portability

NPL program code is extremely portable among all operating system/environments supported by NPL. Typically only minor changes to device equivalence statements are required to take an NPL application from one operating environments to another. It is not necessary to even recompile the programs.

External routines are quite a different story. At a minimum, recompilation, relinking, and retesting of the external routines will be required for each operating environment to be supported. In many cases, depending on the actual function of the external routine, potentially extensive coding changes will be required.

The programmer must consider this issue carefully in deciding whether or not to use external routines. If the external routine is to be an essential component of the application, the programmer must either be prepared to develop and support the external routine on all operating environments supported by NPL or else must accept the fact that the application will now be executable only on those operating environments where the external routine has been implemented.

Alternatively, if the programmer wishes to maintain full portability without providing the external routines for all NPL supported operating environments, the external routines must be restricted to optional or nonessential features of the application so that the application may use conditional logic to determine whether or not the functions provided by the external routine are present at a given installation. In other words, the application must be designed to work with or without the external routines.

CHAPTER 17

EXTENDED EXAMPLES

17.1 Overview

This chapter discusses the extended examples of NPL Release IV.

The following extended programming examples demonstrate how some of the Release IV language capabilities of NPL may be used. Each example increases in complexity and it is suggested to be reviewed in order.

All examples are contained in the EXSNPL.BS2 diskimage found on the Development Package Diskettes. Refer to the appropriate hardware supplement for the exact location of this file.

To start the examples, the REL4EXAM.OBJ can be used. This program is located also on the Development Package.

The following command can be issued:

```
RTI REL4EXAM
```

All examples will go to Immediate Mode after executing to allow code examination and re-execution. When ready to access a different example, the command LOAD RUN will re-display the startup menu.

NOTE: The menu program is called START.

The REL4EXAM boot program does the following:

- Sets the color options.
- Sets INSERT mode on for all operations.
- Modifies $KEYBOARD and $SCREEN to make the underline character be displayed as a true underline instead of a back-arrow under MS-DOS and MS-Windows.
- Establishes a $DEVICE equivalence for NPLSYS.BS2. If this diskimage is placed in a directory other than the current directory, this statement must be modified or example number 12 will fail.

Section 17.2 discusses the programming style used in the examples.

Section 17.3 provides the first example: a WHILE/WEND loop.

Section 17.4 contains a second example: a REPEAT/UNTIL loop.

Section 17.5 provides a third example: simple SWITCH/CASE using a FUNCTION.

Section 17.6 is a fourth example, introducing the use of modules (libraries.)

Section 17.7 provides a fifth example: a simple module with a FUNCTION and PUBLIC section.

Section 17.8 provides a sixth example, using the PROCEDURE of Example 1 contained in another module.

Section 17.9 contains a seventh example, which illustrates a recursively called function.

Section 17.10 is an eighth example: the QSORT routine.

Section 17.11 provides a ninth example, that illustrates the logical type of SWITCH/CASE.

Section 17.12 includes a tenth example, using a more complex SWITCH/CASE statements.

Section 17.13 provides an eleventh example, demonstrating the use of RECORDs and FIELDs.

Section 17.14 is example twelve, illustrating more advanced uses of RECORDs and FIELDs.

17.2 Programming Style Used in Examples

In all of the extended examples, every effort was made to incorporate as many Release IV features as possible. The structured IF...ELSE...ENDIF is used, rather than the simple IF/THEN/ELSE. Structured FOR...BEGIN...NEXT was used instead of the simple FOR...NEXT. SWITCH/CASE statements are used instead of ON...GOTO or ON...GOSUB. FUNCTIONs and PROCEDUREs are used instead of DEFFN’ statements.

Where structured constructs are used, statements within the structure have been indented to provide improved readability, clearly showing the flow of logic. This is a recommended technique for programmers using NPL.

NOTE: In these examples, statements are shown on separate line numbers to make it easier to refer to a specific statement in the following discussion. Except where noted, statements need not be on separate lines.

17.3 Example 1: WHILE/WEND

The program "EX1NPL" demonstrates the use of the new structure construct WHILE-WEND.

17.3.1 Program Listing

```
0005 ; EX1NPL - Simple example of WHILE/WEND
0007 PRINT HEX(03)
0010 DIM Counter
0020 Counter=10
0030 WHILE Counter>0
   :   IF Counter=3
   :      BREAK
   :  ; allows exiting the loop with transfer outside of the loop to
        the following command after the LOOP
   :  END IF
   :  Counter=Counter-1
   :  IF Counter=4
   :      LOOP
   :  ; allows skipping execution of remainder of the body of the loop
        with transfer to the WEND
   :   END IF
   :   PRINT Counter
   : WEND
0040 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.3.2 Discussion

This program demonstrates the use of the WHILE/WEND and IF ... ENDIF structures. Line 30 contains the WHILE/WEND loop and also shows the use of LOOP and BREAK statements.

This program never prints Counter= 4 because the LOOP statement transfers program control to the WEND where the loop condition is evaluated (Counter> 0) and the program continues back to the beginning of the loop. When Counter= 3 the program ends due to the BREAK statement that transfers the program control to the statement following the WEND (PRINT in line 40).

NOTE: The statement in line 30 "Counter= Counter-1" could be replaced by the new operator "-= " as follows: "Counter-= ". This change could be made in any code that uses this type of statement.

General Comments

The above example is very simple, but effectively demonstrates how useful the LOOP and BREAK statements can be.

17.4 Example 2: REPEAT/UNTIL

The program "EX2NPL" demonstrates the use of the new structure REPEAT-UNTIL.

17.4.1 Program Listing

```
0005 ; EX2NPL - Simple example of REPEAT/UNTIL
0007 PRINT HEX(07)
0010 DIM Counter
0020 Counter=10
0030 REPEAT
   :   IF Counter=2
   :        BREAK
   :  ; allows exiting with transfer to the statement following the
        UNTIL
   :   END IF
   :  Counter=Counter-1
   :   IF Counter=4
   :        LOOP
   :  ; allows skipping the remainder of the loop body with transfer
        to the UNTIL
   :  END IF
   :  PRINT Counter
   : UNTIL Counter=0
0040 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.4.2 Discussion

This program demonstrates the use of the new REPEAT/UNTIL and IF ... ENDIF structures. Line 30 contains the REPEAT/UNTIL structure and also illustrates the use of the LOOP and BREAK statements.

The program will never print COUNTER= 4 because the LOOP statement transfers program control to the UNTIL where the loop condition (Counter= 0) is evaluated. The program continues back to the beginning of the loop. When Counter= 2 the program ends due to the BREAK statement that transfers the program control to the statement following the UNTIL (PRINT in line 40).

General Comments

The above example is very simple, but effectively demonstrates how useful the LOOP and BREAK statements can be.

17.5 Example 3: Simple SWITCH/CASE

The program "EX3NPL" demonstrates the use of the new structure SWITCH/CASE used in conjunction with FUNCTIONs.

17.5.1 Program Listing

```
0010  ;  EX3NPL   - Simple example of SWITCH/CASE
0015 PRINT HEX(03)
0020 FUNCTION 'myfunc
0030 RETURN (2)
0040 END FUNCTION
0100 DIM SwitchVal,A
0110 A=1
0120 SwitchVal=24
0130 SWITCH 'myfunc
   :  CASE 2
   :     PRINT "first case"
   :  CASE 55
   :     PRINT "second case"
   :  CASE 24
   :     PRINT "third case"
   :  CASE 48+A
   :     PRINT "fourth case"
   :  CASE
   :     PRINT "default case"
   : END SWITCH
0140 PRINT "End of case"
0150 PRINT "Program Complete - LOAD RUN will reload Munu"
```

17.5.2 Discussion

The above program shows the use of a SWITCH/CASE construct that compares the "case-expressions" to the value return by the 'myfunc function. As the function 'myfunc returns a value of 2 (RETURN(2) in line 30), the program executes the statements following CASE 2 and print "first case".

NOTE: CASE can use an expression instead of a value as with CASE 48+ A.

General Comments

SWITCH/CASE constructs, as demonstrated in the above example, are a very useful tool in replacing the ON GOTO and ON GOSUB statements. Examples of SWITCH/CASE constructs using logical conditions are discussed later in the chapter.

17.6 Example 4: Example Module (Library)

The programs "EX4NPL" and "FACT" calculate the factorial of a number entered by the user. This example illustrates a recursive function and the use of external modules (a library).

17.6.1 Program Listing

Figure 1

```
0010 ;EX4NPL
0020 PRINT HEX(03)
0030 INCLUDE T "FACT"
0040 INPUT "Input the value to calculate factorial for ",X
0050 PRINT 'factorial(X)
0060 PRINT "Program Complete - LOAD RUN will reload Menu"
```

Figure 2

```
0010  ; FACT
0020 FUNCTION 'Factorial(Value)/PUBLIC   :; Declare 'Factorial as
                                            Public Function
0030 DIM /RECURSIVE Answer          :; This is a recursive variable
0040 IF Value<0                     :; test the trivial case
   :   RETURN (0)
   : END IF
0050 IF Value<=1                    :; test the second trivial case
   :    RETURN (1)
   : END IF
0060 Answer=Value*'Factorial(Value-1)     :; Recursively calling itself
0070 RETURN (Answer)
0080 END FUNCTION
```

17.6.2 Discussion

Program "EX4NPL" in Figure 1 performs the following tasks:

Line	Commentary
30	Contains an INCLUDE statement. The INCLUDE statement specifies that the program file "FACT" on the currently selected diskimage file is to be loaded as a module. This statement is executed at resolution time.
50	Demonstrates the use of a FUNCTION as a numeric expression. The value entered by the operator is passed to the function as a parameter. The returned value of the FUNCTION ’Factorial is what is printed.

Program "FACT" in Figure 2 illustrates use of a recursive function.

NOTE: The line numbers used in "FACT" overlap with those used in "EX4NPL". Line numbers are local to each module so there is no conflict. Local variables are also supported but are not used in this example.

The module contains only one function, ’Factorial. This function calculates the factorial of the parameter it is passed. Specifically:

Line	Commentary
20	Declares the function ’Factorial. This function has a single numeric parameter named Value. This parameter is passed by value
30	Dimensions the recursive variable Answer. It is not necessary to explicitly dimension all recursive variables.
40	Tests for a valid value. Factorials cannot be calculated for negative numbers.

NOTE: This program uses a new structured form of IF (which does not use the THEN key-word). Two statements follow the IF (the comment and the RETURN statement). Both are executed if the condition is true.

Line	Commentary
50	Tests for the exit condition of the recursion loop. For a factorial calculation, the exit condition occurs when the value reaches one.
60	Performs the actual work in this routine. ’Factorial is called recursively with Value being decremented by one for each call. For each iteration, new copies of Answer and Value are dynamically allocated.
70	Returns the result of the routine to the calling program.
80	The block terminator for the function.

General Comments

The recursion aspect of this example may make it difficult to follow for readers not familiar with recursion. Functions are always recursive, but the recursive capability does not have to be used. Most functions are not called recursively.

17.7 Example 5: Module using PUBLIC Sections

This program demonstrates how a program may be written so that the mainline performs a required procedure (in this case, report from the "SALES" file), while also making the code used to implement the program available to other modules by declaring them as PUBLIC within a PUBLIC section. In this case, the 'Report procedure could also be used by other modules. Refer to Example 2.

17.7.1 Program Listing

```
0100 ;  module EX5NPL :Demonstrate a simple module with a Function
0105 PUBLIC
0110 ; contains 'Report format parameters
0120 DIM Format$32="###,###,###.##--"
   : ;  default number format
0135 DIM Spec$1="_"
   : ; default underlines
0145 PROCEDURE 'Report(File$8,Device,Name_Rpt$50)/FORWARD
0150 END PUBLIC
0155 ; ***************************************************************
0160 ; Report is a typical procedure with no return value
0170 PROCEDURE 'Report(File$8,Device,Name_Rpt$50)/PUBLIC /BEGINS
0180 ; File$8 is the name of the file to read from
0190 ; Device is the device number where the file is located
0200 ; Name_Rpt$ contains the report title
0210 ;
0220 DIM Exit,Amount,Total
   : ;/RECURSIVE variables implied
0230 DIM U$64
0240 ; Exit and Amount are initialized to 0 each time 'Report is called
0250 GOSUB 340
   : ; goto the Initialize routine to prepare report
0260 GOSUB 410 : ; goto the ReadRecord routine to Read the first record
0270 WHILE Exit=0
   : ; repeat loop while Exit is still 0
0280    GOSUB 490
   : ; goto ProcessRecord routine to process record
0290    GOSUB 410
   : ; goto ReadRecord routine to read the next record
0300 WEND
   : ; marks the end of the body of the loop
0310 GOSUB 540
   : ; goto routine Finishup to complete the report
```

```
0320 RETURN
0330 ;
0335 ; ***************************************************************
0337 ;
0340 ; Initialize
0350 ; Print the title of the Report
0360 DATA LOAD DC OPEN T#Device,File$
0365 PRINT HEX(03): PRINT AT(4,1);Name_Rpt$: PRINT : PRINT
0370 Exit=0
   : ; clear the end of file condition
0380 Total=0
0390 RETURN
0400 ;
0405 ; ***************************************************************
0407 ;
0410 ;ReadRecord
0420 ; Read The next Record
0430 DATA LOAD DC #Device,Amount
0440 IF END
0450    Exit=1
   : ; set the end of file condition
0460 END IF
0470 RETURN
0480 ;
0485 ; ***************************************************************
0487 ;
0490 ; ProcessRecord
0500 PRINTUSING Format$,Amount
0510 Total=Total+Amount
0520 RETURN
0530 ;
0535 ; ***************************************************************
0537 ;
0540 ; Finishup
0550 U$=ALL(Spec$)
0560 PRINTUSING Format$,U$
0570 PRINTUSING Format$,Total
0590 DATA SAVE DC CLOSE #Device
0600 RETURN
0610 END PROCEDURE
0620 ;
0625 ; ***************************************************************
0627 ;
0630 ; Mainline
0640 DIM ReportSpec$1,Rpt_Title$50
0660 Rpt_Title$="Sales Report for October 1992"
0670 $DEVICE(/D33)="EXSNPL.BS2"
0680 SELECT #1/D33
0700 'Report("SALES2",1,Rpt_Title$)
0800 PRINT "Program Complete - LOAD RUN will reload Menu"
0810 DATA 22.3,500.40,1200.34,33.5
```

17.7.2 Detailed Discussion

Line	Commentary
0100	Illustrates the use of line format comments (starting with ";"). Line comments may contain ":", and this is not treated as end of statement.
0105	Defines the start of the PUBLIC section for the module. By declaring a PUBLIC section, the items in it which are marked as PUBLIC become usable by other programs.
	Indenting of the statements in a PUBLIC section is a useful way to highlight the contents of the interface, and serves as a reminder that a matching END PUBLIC statement is needed. Here, we have opted not to indent the statements in the interface section.
0120	Defines Format$, with a maximum length of 32 bytes. As the comment indicates, this is a format value used to print numbers. The initial value is a common one.
0135	Defines Spec$ with a maximum length of 1 byte. As the comment indicates, this is the character used to print underlines, prior to a total.
0145	Declares the 'Report PROCEDURE and lists the parameters of the PROCEDURE. Declaring the procedure as PUBLIC allows other modules to call it. The FORWARD keyword indicates that the actual procedure definition will occur later.
0150	Declares the end of the default PUBLIC for this module. The publicly visible elements of the module are the variables Format$ and Spec$ and the procedure 'Report.
0170	Redeclares the 'Report PROCEDURE and lists the parameters of the procedure. The BEGINS keyword indicates that the procedure has been previously defined in a FORWARD reference. The parameters of the procedure are listed in the order that the arguments must be provided. In this example all the parameters are passed by value (no POINTER keyword). The first parameter is a string, and has a maximum length of 8 characters (if a longer string is passed, it would be truncated to 8). The second parameter is a number, and the third is a string with a 50 character maximum (Name_Rpt$). Indenting of the statements in the body of a procedure is a useful way to highlight the body of the procedure, and serves as a reminder that a matching END PROCEDURE statement is needed. In this case, we have opted not to indent the procedure body.

Line	Commentary
0220	Exit, Amount and Total are dimensioned as (implicitly RECURSIVE) numeric variables used by the PROCEDURE. Note that memory for these variables and for the parameters of the procedure is not allocated until the procedure is called, and they may not be referenced outside the body of the PROCEDURE. The memory for the variables is released when the procedure RETURNs.
0230	U$ is dimensioned for use as a temporary (RECURSIVE) variable.
0250	A GOSUB is used to execute a local subroutine at line 340 to perform startup for the report. A statement label could be used at line 340 (i.e., = Initialize), and the subroutine could be referenced with it (i.e., GOSUB Initialize).
0260	A GOSUB is used to execute a local subroutine at line 410 to read the first record from the file. A statement label could be used at line 410 (i.e., = ReadRecord), and the subroutine could be referenced with it (i.e., GOSUB ReadRecord). Implicitly, this subroutine sets the status variable Exit to a non-zero value when end of file is reached.
0270	The WHILE statement indicates the start of a structured loop. The condition Exit= 0 tests to make sure end of file was not detected yet. The body of the loop (up to the matching WEND statement on line 0300) is performed repeatedly as long as the condition remains true.
0280	A GOSUB is used to execute a local subroutine at line 490 to process the last record read. A statement label could be used at line 490 (i.e., = ProcessRecord), and the subroutine could be referenced with it (i.e., GOSUB ProcessRecord). Indenting of the statements in the body of a structured loop is a useful way to highlight the body of the loop, and serves as a reminder that a matching WEND statement is needed.
0290	The next record of the file is read, again calling the subroutine at line 410 (ReadRecord.)
0300	The WEND statement marks the end of the body of the structured loop started by the WHILE statement on line 0270. The condition in the WHILE statement is tested again, and if it is still true the loop body (lines 0280 to 0290) is executed again.
0310	Once end of file is reached, the structured loop is complete and the local subroutine at line 540 is called.
0320	This return statement returns control to the caller of the 'Report procedure. Memory allocated to RECURSIVE variables is released. Since this is a procedure, no return value is required (or permitted) on the RETURN statement.
0340	This statement (a REMark) marks the start of the Initialize subroutine. Here, a statement label could be used to define the start of the subroutine.
0360	The file (name and device number specified by the parameters to the procedure) is opened.
0370	The end of file flag (Exit) is cleared.

Line	Commentary
0380	The report total is (Total) zeroed.
0390	This RETURN statement returns control to the caller of this subroutine (following the GOSUB.)
0410	This statement (a REMark) marks the start of the ReadRecord subroutine. Here, a statement label could be used to define the start of the subroutine.
0430	The next record is read from the file.
0440	The IF statement marks the start of a structured IF...ELSE...END IF block. The system logical variable END is set to a true value by any DC type I/O (here the DATA LOAD statement on line 430) if the end-of-file is reached.
0450	This statement makes up the block of statements executed when the IF statement is true. The end of file indicator is set. Indenting of the statements in the true and false blocks of an IF...ELSE...END IF structure is a useful way to highlight the limits of the conditionally executed blocks, and serves as a reminder that a matching (ELSE or) END IF statement is needed.
0460	The END IF statement marks the end of the structured IF...ELSE...END IF block started on line 440. Here there is no ELSE section in the IF...ELSE...ENDIF structure.
0470	This RETURN statement returns control to the caller of this subroutine.
0490	This statement (a REMark) marks the start of the ProcessRecord subroutine. Here, a statement label could be used to define the start of the subroutine.
0500	The value (Amount) read from the current data record is printed, using a format specified as part of the PUBLIC declaration.
0510	The value from the record is added in to the report total.
0520	This RETURN statement returns control to the caller of this subroutine.
0540	This statement (a REMark) marks the start of the Finishup subroutine. Here, a statement label could be used to define the start of the subroutine.
0550	A temporary value is set to an underline value, using a character (typically -, = or blank) chosen as part of the report format specification.
0560	The underline value is printed, using the same width as the field format specified for the number.
0570	The total is printed, using the same format as the record amounts.
0590	The selected device is closed (to avoid accidental use by other DC statements).
0600	This RETURN statement returns control to the caller of this subroutine.
0610	The END PROCEDURE marks the end of the 'Report procedure body. After this statement, parameters and variables defined within the procedure body (e.g., Device, Total, Amount) may no longer be referenced (references revert to whatever meaning they had before the PROCEDURE declaration).

Line	Commentary
0640	This statement declares the variables ReportSpec$ and Rep_Title$.
0700	The 'Report procedure is called to print the report, using a file called "SALES2" on the current device #1 address (which is previously defined via a SELECT #1 statement in line 680.)

General Comments

This program demonstrates how a program may be written so that the mainline performs a required procedure (in this case, report from the "SALES2" file), while also making the code used to implement the program available to other modules by declaring them as PUBLIC within a PUBLIC section. In this case, the 'Report procedure could also be used by other modules.

The program also demonstrates the use of GOSUB statements in the body of the PROCEDURE. These type of statements may be replaced by 'local' subroutines. Use of local subroutines (defined by GOSUB statements) could also have been defined as PROCEDURES or simply entered as in-line statements in the 'Report procedure body. However by defining them as local subroutines within the procedure body has a couple of benefits:

- Use of the subroutines makes the high-level flow of the PROCEDURE clearer than in-line statements would be. This advantage becomes more obvious as the size and complexity of the component subroutines increases.

- Code need not be duplicated where it is used more than once (e.g., ReadRecord).

- Parameters common to some or all the subroutines (e.g., the Device number, Spec user-type and Amount fields) need not be passed as parameters.

- Where code is indented to highlight the structure of code, indenting levels can be kept within a legible limit by extracting some code as a local subroutine, leaving lots of room on each line for comments.

- The program illustrates the use of a structured WHILE loop.

NOTE: **Because the condition which causes the exit of the loop is set by the last record read, a test for end of file is required before entering the loop in case the file is empty, and an additional read to get the next record before the end of the loop. An alternative loop which would avoid calling ReadRecord could have been (replacing lines 0260-0300):**

```
0260 WHILE TRUE                        : ; loop until BREAK exits
0270   GOSUB 410                       : ; (ReadRecord) - read record to process
0280   IF Exit<>0 THEN BREAK             : ; test for end of file
0290   GOSUB 490                       : ; (ProcessRecord) - process a record
0300 WEND                              : ; end of loop until BREAK exits
```

17.8 Example 6: Module using PROCEDURE

This example assumes that the program of Example 5 is stored on the default (#0) diskimage with the name "EX5NPL".

This program demonstrates how a module may be written so that the mainline performs a required function (in this case, report from the "PURCHASE" file), by using previously written procedures that are available within another module's PUBLIC section. For example, the 'Report procedure was already written, and is suitable for our purpose.

17.8.1 Program Listing

```
0100 ;module EX6NPL
0110 INCLUDE T "EX5NPL"
   : ; defines 'Report and ReportSpec
0115 ; Location of Sales2 file changed but the changes will be made
only here
0120 DIM _Sales2_Device=2
   : ; file number to use
0125 DIM _Sales2_Platter$="D12"
   : ; location of SALES2 file
0128 DIM _Rpt_Title$="Sales for April"
   : ; new title of report
0130 ; Mainline
0155 $DEVICE(<_Sales2_Platter$>)="EXSNPL.BS2"
0160 SELECT #_Sales2_Device/D12
0170 'Report("NEWSALE",_Sales2_Device,_Rpt_Title$)
0180 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.8.2 Detailed Discussion

Line	Commentary
0110	The INCLUDE statement indicates that, to run, this module needs information in PUBLIC sections defined elsewhere. The file with the interface information is located on the default platter address with the file name "EX5NPL". The EX5NPL module is loaded automatically any time the EX6NPL module is run. On the INCLUDE statement, no specific module name is given (no "TO xx" specification) so the module name given to the EX5NPL program is, by default, the same as the file name ("EX5NPL".)
0120	The device number and platter address of the purchases file are defined here as constants on the next few lines. Although the location of the file won't change during execution, according to the comments, this file appears to move on occasion, so the values are assigned to constants rather than written into the code as literals. By defining these as constant values, programs are inhibited from unintentionally changing the values elsewhere.
0125	Define the platter address for the new sales file.
0160	The program ensures that the selected file number maps to the correct platter address.
0170	The public 'Report procedure (defined in EX5NPL's PUBLIC section) is called, with parameters for the purchases file.

General Comments

This program demonstrates how a module may be written so that the mainline performs a required function (in this case, a report from the "NEWSALE" file), by using previously written procedures that have been made available within another module's PUBLIC section. The 'Report procedure was already written, and is suitable for our purpose.

NOTE: A complete understanding of the EX6NPL module depends to some extent on knowing the contents of the interface provided by EX5NPL. As a minimum, the systems documentation for any module which is to be made publicly available should include the names of the available PUBLIC sections, and within each PUBLIC the names of PROCEDUREs which are provided. A description of the parameters and return codes of FUNCTIONs should also be provided. Depending on the complexity of the system, cross references of functions to interface names and module names may also be useful.

The use of constants allows values which are unchanging to be given a name. While the values could also be assigned to variables, the use of constants has a few clear benefits:

- The names of constants may describe their use more clearly than a simple number or string would do, (e.g., instead of pulling the magic number 32767 out of a hat, the name _LARGEST_16_BIT_SIGNED_INTEGER more clearly describes the derivation of the number.)

- Constants, unlike other variables, may not be assigned new values (except where they are defined). Inadvertent changes to the value are avoided.

- Program logic may be clearer to follow if some values are evidently constant, (e.g., in the logical condition IF _LARGEST_ALLOWED_VALUE > X, it is clearly the value in X that is being tested).

- Especially in PUBLIC sections, PUBLIC constants can make special values which must be supplied easier to remember, resulting in less need to refer to interface documentation.

17.9 Example 7: A Recursive Function

This illustrates the use of a recursive function.

For deeply recursive functions, it is quite possible that a lot of excess calculations will be done, depending on how efficiently the function is implemented. In the example, to compute 'Fibonacci(20), the value 'Fibonacci(0) is computed 4181 times and 'Fibonacci(1) is computed 6765 times. In this case, some simple changes to the algorithm (such as "remembering" computed values in an array) could drastically improve performance. In other cases, where the relationship between values is not so simply defined, this may not be an option.

The example could be more simply formulated, eliminating the Term1 and Term2 variables and computing Result= 'Fibonacci(Num-1)+ 'Fibonacci(Num-2) for the non-trivial case.

17.9.1 Program Listing

```
0100 ;module EX7NPL
0102 PRINT HEX(03)
0105 ; this prigram demonstrates the use of a FUNCTION to determine
       the well known Fibonacci number sequence, of which the first few
terms are: 1, 1, 2, 3, 5, 8, 13, ...
0100 FUNCTION 'Fibonacci(Num)
0110 DIM Term1,Term2,Result
   : ; /RECURSIVE is implied
0120 Num=INT(Num)
   : ; Make sure an integer
0130 IF Num<=1
   : ; all well behaved recursions end some time...
0140   Result=1
   : ; definition of Fibonacci(0) and Fibonacci(1)
0150 ELSE          : ; now handle the non-trivial case
0160   Term1='Fibonacci(Num-1)
   : ; get preceding term
   :   Term2='Fibonacci(Num-2)
   : ; and the one before that
0170   Result=Term1+Term2
   : ; add them to get the result
0180 END IF
0185 RETURN(Result)
0190 END FUNCTION 'Fibonacci
0200 ; Mainline
0210 FOR Index=1 TO 17 BEGIN
0220   PRINT Index;'Fibonacci(Index)
0230 NEXT Index
0240 PRINT 'Fibonacci(18)/'Fibonacci(17),(SQR(5)+1)/2
0250 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.9.2 Detailed Discussion

Line	Commentary
0100	Declares a numeric function 'Fibonacci, which takes one parameter. This function computes the well-known Fibonacci number sequence, of which the first few terms are: 1, 1, 2, 3, 5, 8, 13, 21, ... and subsequent terms are generated by adding the previous two numbers in the sequence.
0110	Dimensions numerics Term1 and Term2 to hold intermediate values used by the function. Because these declarations are in a function, they are by default allocated as recursive variables. This is important, since the function calls itself.
0120	As a precaution against a bad (fractional) parameter, we truncate the parameter to an integer.
0130	To end recursion, some values of the function must be defined without the function calling itself! For a number less than or equal to 1.

Line	Commentary
0140	The true block of the IF...ELSE...END IF structure handles predefined values. By definition, the first two terms of the sequence are ’Fibonacci(0)= 1, and ’Fibonacci(1)= 1.
0150	Marks the end of the true block of the IF...ELSE...END IF structure.
0160	The false block of the IF...ELSE...END IF structure handles other values. By definition, for later values ’Fibonacci(X) we must compute the previous two terms, i.e., ’Fibonacci(X-2) and ’Fibonacci(X-1).
0170	The result of the function in this case is the sum of the two previous terms.
0180	Marks the end of the IF...ELSE...END IF structure.
0185	Defines the return value of the function.
0190	Marks the end of the body of the FUNCTION ’Fibonacci.
0210	Mainline of the program - print the first 20 terms. The FOR/BEGIN statement marks the start of a structured loop. The index variable T will take on values from 1 to 20.
0220	Prints the index and the function value for the loop index. Indenting of the statements in the body of a FOR/BEGIN loop is a useful way to highlight the limits of the loop, and serves as a reminder that a matching NEXT statement is needed.
0230	Matching NEXT statement marks the end of the FOR/BEGIN structured loop.
0240	Shows the ratio of the last two computed terms in the sequence, and a magic expression which they should approximate.

General Comments

Much like Example 3, this example is a good illustration of the concept of recursion, which may be difficult to follow for those not familiar with the idea.

17.10 Example 8: QSORT Routine

This example, "QSORT", illustrates a more complex program containing a series of PROCEDUREs and FUNCTIONs. The program sorts 2,000 random numbers.

17.10.1 Program Listing

```
0010 ;--------------------------------------------------------'Swap--
  : ; A utility to swap two numerics X and Y
  : ; Two by-reference parameters are used
  : PROCEDURE 'Swap(/POINTER X,/POINTER Y)
  :    DIM temp
  :    temp=X
  :    X=Y
  :    Y=temp
  : END PROCEDURE 'Swap
0040 ;--------------------------------------------------------stats
  : ;Variables used to keep statistics about how often and
  : ;How deeply recursive 'Qsort is called
  : DIM MaximumRecursionLevel,CallsToQsort
0050 ;------------------------'Qsort
  : ;Quick sort elements LowIndex to HighIndex of array X()
  : ;/PUBLIC decl. is required so the name can be specified indirectly
  : PROCEDURE 'Qsort(/POINTER X(),LowIndex,HighIndex)/PUBLIC
  : ;default declaration is /RECURSIVE, but this is just a reminder...
  : ;DIM /RECURSIVE TopPart
  : ;it turns out these variables could also be statically allocated.
  : ;But if you don't want to worry, just declare them inside the
  : ;function body.
  : DIM PivotIndex,Pivot,ScanIndex
0060 ;This value must be static, to keep it from being reinitialized by
  : ;each call to 'Qsort.
  : DIM /STATIC LevelOfRecursion
  : ;Update statistics and put something on the screen to show how deep
  : SELECT PRINT 005
  : CallsToQsort=CallsToQsort+1
  : PRINT "X";
  : LevelOfRecursion=LevelOfRecursion+1
  : IF LevelOfRecursion>MaximumRecursionLevel
  :    ;We have reached new heights (depths?) of recursion, make a note
  :     MaximumRecursionLevel=LevelOfRecursion
  :  END IF
0070 ;All good recursion must eventually come to a trivial case...
  : DIM Nelements
  : Nelements=HighIndex-LowIndex+1
  : IF Nelements2
  :    ;reduced to trivial cases, (1 or 2 elements)
  :    IF X(LowIndex)>X(HighIndex)
  :       'Swap(X(LowIndex),X(HighIndex))
  :     END IF
  : ELSE
  :    PivotIndex=LowIndex+INT(RND(1)*Nelements)R  :  ;pick a pivot
  :    Pivot=X(PivotIndex)
  :    'Swap(X(PivotIndex),X(HighIndex))
  :  ;put it in the top part
  :    TopPart=HighIndex
  :  ;start index of top part
  :    ScanIndex=LowIndex
  :  ;first element not yet looked at...
0080     REPEAT
  :        IF X(ScanIndex)>Pivot
  :    ;does entry belong in top part?
```

```
  :          TopPart=TopPart-1
  :     ;yes, make the top part bigger
  :         'Swap(X(ScanIndex),X(TopPart))
  :     ;put in top part
  :      ELSE
  :          ScanIndex=ScanIndex+1
  :    ;no, next index
  :        END IF
  :    UNTIL ScanIndex>=TopPart
  :    ;finished when scanned up to top part
  :    IF TopPart=LowIndex
  :     ;unlucky choice of pivot, it was the smallest!
  :     ;swap it to Lo to ensure total size is getting smaller
  :     ;on each step of the recursion.
  :     'Swap(X(LowIndex),X(HighIndex)
  :     TopPart=LowIndex+1
  :    END IF
  :    ;Now have LowIndex... Center-1 all <=Pivot
  :    ;and     Center...HighIndex are all >Pivot
  :    'Qsort(X(),LowIndex,TopPart-1)
  :    'Qsort(X(),TopPart,HighIndex)
  : END IF
  : PRINT HEX(082008);
  : LevelOfRecursion=LevelOfRecursion-1
  : END PROCEDURE
0085 PROCEDURE 'Display(/POINTER _Array(),Nelements,/POINTER _Image$)
  : DIM Index
  : For Index=1 TO Nelements BEGIN
  :   PRINTUSING "-##.##;_Array(Index);
  :   IF MOD(Index,13)=0 THEN PRINT
  : NEXT Index
  : PRINT
  : END PROCEDURE 'Display
0090 ;------------------------------------------------------mainline-
  : DIM _ArraySize=2000, DIM T
  : DIM _IndirectQsortName$="Qsort"
  : DIM NumberList(_ArraySize)
  : T=RND(0)
  :  ;show consistent results...
  : FOR T=1 TO _ArraySize BEGIN
  :    NumberList(T)=ROUND(RND(1)*100,2)
  : NEXT T
  : 'Display(NumberList(),_ArraySize,"+##.##")
  : '<_IndirectQsortName$>(NumberList(),1,_Arraysize)
  : 'Display(NumberList(),_ArraySize,"+##.##")
  : PRINT "Maximum depth of recursive calls";MaximumRecursionLevel
  : PRINT "Calls to Qsort function required";CallsToQsort
  : CallsToQsort,MaximumRecursionLevel=0
  :  ;reset for next call
  : 'Qsort(NumberList(),1,_ArraySize)
  : PRINT "When already sorted";MaximumRecursionLevel;CallsToQsort
  : END
```

17.10.2 Discussion

The program makes use of a whole series of Release IV features like PROCEDUREs, recursive FUNCTIONs, POINTERs, and other structure constructs.

General Comments

The above program makes use of features demonstrated by the previous examples and is well documented.

17.11 Example 9: Logical SWITCH/CASE

Program "EX9NPL" demonstrates the use of the logical form of the SWITCH/CASE construct. The logical SWITCH/CASE allows a complex expression to be specified on CASE statements. The first (and only the first) CASE to be evaluated as TRUE is the CASE that is executed. This form of SWITCH/CASE provides a powerful alternative to complex nested IF statements.

17.11.1 Program Listing

```
0010 ; EX9NPL - Simple example of Logical SWITCH/CASE
0015 PRINT HEX(03)
0020 DIM A,B,C
0030 A=1
   : B=2
   : C=3
0040 SWITCH
   : CASE A>1
   :   PRINT "case 1"
   : CASE B<C
   :   PRINT "case 2"
   : CASE C=A+B
   :   PRINT "case 3"
   : CASE
   :   PRINT "default case"
   : END SWITCH
0050 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.11.2 Discussion

The result of this program is that "case 2" is printed because B is the first case to evaluate as TRUE. "case 3" does not print because only the first CASE to meet the requirements is executed.

General Comments

This is another simple example of using SWITCH/CASE, specifically with logical CASE statements. Example 3 illustrates the use of simpler numeric CASE statements.

17.12 Example 10: Complex SWITCH/CASE

Program "EX10NPL" demonstrates the use of complex nested SWITCH/CASE logic.

17.12.1 Program Listing

```
0010 ;EX10NPL - nested switch/case
0020 ;This example demonstrates a good alternative to fall through CASE
   : ; NPL does not support fall through CASE
0025 DIM Error_Code,Error_Sub_Code
0030 Error_Code=12
   : Error_Sub_Code=3
0040 SWITCH
   : CASE Error_Code<10
   :   PRINT "Case 1"
   : CASE Error_Code>=10 AND Error_Code<20
   :   ; Here is some stuff we want to do for all codes from 10-19
   :   PRINT "case 2; error code>=10 and <20"
   :   SWITCH Error_Code
   :   ; Now we want to do some additional stuff for each specific code
   :       CASE 10
   :           PRINT "case 1, level 2 - error code=10"
   :       CASE 11
   :           PRINT "case 2, level 2 - error-code=11"
0050       CASE 12
   :           PRINT "case 3, level 2 - error-code=12"
   :           SWITCH Error_Sub_Code
   :               ; Now we want a third level based on sub-error-code
   :               CASE 1
   :                   PRINT "case 1, level 3"
   :               CASE 2
   :                   PRINT "case 2, level 3"
   :               CASE 3
   :                   PRINT "case 3, level 3"
   :               CASE
   :                   PRINT "default case, level 3"
   :           END SWITCH
   :       CASE
   :           PRINT "default case, level 2"
   :       END SWITCH
   :       PRINT "Still at case 2, level 1, error-code>=10 and <20"
   :   CASE
   :       PRINT "default case, level 1"
   : END SWITCH
0060 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.12.2 Results

```
case 2; error code >=10 and <20
case 3, level 2 - error-code=12
case 3, level 3
Still at case 2, level 1, error-code >=10 and <20
Program Complete - LOAD RUN will reload Menu
```

17.12.3 Discussion

In this example, the program is required to:

- Execute one group of statements for all Error_Codes within a certain range.
- Execute additional logic for specific Error_Codes.
- Execute further specific logic for Error_Sub_Codes if the Error_Code is a specific value.
- Execute more statements for all Error_Codes within the range.

This example shows how complex logic can be greatly simplified by the use of SWITCH/CASE.

General Comments

This example provides a more comprehensive illustration of using SWITCH/CASE statements in a real-world scenario. It shows how structures can be nested within other structures, providing endless logical combinations, yet remaining well-organized.

17.13 Example 11: RECORDS/FIELDS

Program "EX11NPL" demonstrates the use of the new RECORD/FIELD construct. The following example demonstrates some more sophisticated features of RECORDs/FIELDs.

17.13.1 Program Listing

```
0010 ; EX11NPL - Simple Example of Records/Fields
0020 RECORD cr
   : FIELD name$=HEX(A020)
   : FIELD amount=HEX(5205)
   : FIELD last_pay=HEX(F108)
   : END RECORD cr
0030 DIM cust_record$#RECORDLENGTH(cr),A$1
0040 cust_record$.name$="Harry"
0050 cust_record$.amount=123.45
0060 cust_record$.last_pay=678.91
0070 LIST DIM * cust_record$
0080 KEYIN A$
0090 cust_record$.amount=cust_record$.amount+32.78
0100 PRINT "32.78 added to Amount - new value = ";cust_record$.amount
0110 DIM new_cust_record$#RECORDLENGTH(cr)
0115 KEYIN A$
0120 new_cust_record$.name$="Frank"
0130 new_cust_record$.amount=567.89
0140 new_cust_record$.last_pay=123.45
0150 LIST DIM * new_cust_record$
0160 LIST DIM * cust_record$
0170 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.13.2 Results

```
DIM cust_record$45
        "Harry           " HEX(4861 7272 7920 2020 2020 2020 2020 2020)
STR(17) "                " HEX(2020 2020 2020 2020 2020 2020 2020 2020)
STR(33) "..î4\.....3¢ç"    HEX(0000 1234 5C00 0000 0109 33FF FE)
32.78 added to Amount - new value =  156.23
DIM new_cust_record$45
        "Frank           " HEX(4672 616E 6B20 2020 2020 2020 2020 2020)
STR(17) "                " HEX(2020 2020 2020 2020 2020 2020 2020 2020)
STR(33) "..Vxù....09¢ç"    HEX(0000 5678 9C00 0000 0030 39FF FE)
DIM cust_record$45
        "Harry           " HEX(4861 7272 7920 2020 2020 2020 2020 2020)
STR(17) "                " HEX(2020 2020 2020 2020 2020 2020 2020 2020)
STR(33) "..äb......3¢ç"    HEX(0000 1562 3C00 0000 0109 33FF FE)
Program Complete - LOAD RUN will reload Menu
```

17.13.3 Discussion

Line	Commentary
0020	Defines a logical RECORD with three fields. In this example, the field specification is in the form of $PACK specifications. Name$ is an alpha field of length 32 decimal (20 binary). Amount is a numeric field stored in a packed decimal format with 2 decimal positions. Last_Pay is a numeric field stored in NPL internal floating point format. The record "cr" does not occupy physical storage. Rather this is a logical structure used to access other variables.
0030	The buffer variable Cust_Record$ is defined. The length for Cust_Record$ is established using the #RECORDLENGTH function which returns the number of bytes required to store the specified logical record (in this case "cr").
0040	The logical FIELD Name$ within variable Cust_Record$ is set to a value of "Harry". Based on the definition of logical record "cr", NPL knows that Name$ is the first field in the record and that it is an alpha field with a length of 32. Therefore, this statement is equivalent to: STR(Cust_Record$,1,32)=" Harry"
0050	The logical FIELD Amount within Cust_Record$ is set to a value of 123.45. Based on the definition of logical record "cr", NPL knows that Amount is a packed decimal field of length 5 starting at byte 33. Therefore this statement is equivalent to: $PACK (F= HEX(5205)) STR(Cust_Record$,33,5) FROM 123.45
0060	Equivalent to: $PACK (F= HEX(F108)) STR(Cust_Record$,38,8) FROM 678.91
0070	The LIST DIM * statement shows the contents of Cust_Record$ at this point.
0090	The Amount FIELD within Cust_Record$ is incremented by 32.78. This is equivalent to: $UNPACK (F= HEX(5205)) STR(Cust_Record$,33,5) TO X X=X+ 32.78 $PACK (F= HEX(5205)) STR(Cust_Record$,33,5) FROM X
0100	The results of the last operation are displayed by directly printing the current value of the Amount FIELD from Cust_Record$.
0110	A second record buffer is defined.

The remainder of the program places values into the new record buffer using the same logical RECORD "cr". This illustrates that logical RECORD "cr" is independent of physical storage.

17.14 Example 12: Complex Use of RECORDs

Program "EX12NPL" demonstrates advanced features of the RECORD/FIELD implementation including:

- Use of mnemonic field specifications
- Use of indirect field specifications
- Use of FIELD functions

17.14.1 Program Listing

```
0010 ; EX12NPL
0015 DIM T$32
0020 INCLUDE T/D21,"PCKFIELD"
0030 USES PackFormats
0040 RECORD /PUBLIC cr
   : FIELD name$=HEX(A020)
   : FIELD amount='FieldType$(_PACK_IBM_PACKED_DECIMAL_FORMAT,2,5)
   : FIELD last_pay='FieldType$ (_PACK_FLOATING_POINT_FORMAT,
     _PACK_BASIC2C_INTERNAL_NUMERIC_FORMAT,8)
   : END RECORD cr
0050 DIM cust_record$#RECORDLENGTH(cr),A$1
0060 cust_record$.name$="Harry"
0070 cust_record$.amount=123.45
0080 cust_record$.last_pay=678.91
0090 LIST DIM * cust_record$
0100 KEYIN A$
0110 cust_record$.amount=cust_record$.amount+32.78
0120 PRINT "32.78 added to Amount - new value = ";cust_record$.amount
0130 DIM new_cust_record$#RECORDLENGTH(cr)
0140 KEYIN A$
0145 T$="name$"
0150 new_cust_record$.<T$>$="Frank"
0155 T$="amount"
0160 new_cust_record$.<T$>=567.89
0170 new_cust_record$.last_pay=123.45
0180 LIST DIM * new_cust_record$
0190 KEYIN A$
0200 PRINT ,"Field Start, Field Length, Field Format"
0210 PRINT "    Name$ ",#FIELDSTART(.name$),#FIELDLENGTH(.name$),
    : HEXPRINT $FIELDFORMAT(.name$)
```

```
0220 PRINT "    Amount ",#FIELDSTART(.amount),#FIELDLENGTH(.amount),
    : HEXPRINT $FIELDFORMAT(.amount)
0230 PRINT "    Last_Pay ", #FIELDSTART(.last_pay),
     #FIELDLENGTH(.last_pay),
    : HEXPRINT $FIELDFORMAT(.last_pay)
0240 PRINT "Recordlength = ";#RECORDLENGTH(cr)
0250 KEYIN A$
0260 $UNPACK(F=$FIELDFORMAT(.amount)) STR(cust_record$,
     #FIELDSTART(.amount), #FIELDLENGTH(.amount)) TO A
   : PRINT A
0270 PRINT "Program Complete - LOAD RUN will reload Menu"
```

17.14.2 Results

```
DIM cust_record$45
        "Harry          "  HEX(4861 7272 7920 2020 2020 2020 2020 2020)
STR(17) "               "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
STR(33) "..î4\.....3¢ç"    HEX(0000 1234 5C00 0000 0109 33FF FE)
32.78 added to Amount - new value =  156.23
DIM new_cust_record$45
        "Frank          "  HEX(4672 616E 6B20 2020 2020 2020 2020 2020)
STR(17) "               "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
STR(33) "..Vxù....09¢ç"    HEX(0000 5678 9C00 0000 0030 39FF FE)
              Field Start, Field Length, Field Format
    Name$        1             32            A020
    Amount       33            5             5205
    Last_Pay     38            8             F108
Recordlength =  45
 156.23
Program Complete - LOAD RUN will reload Menu
```

17.14.3 Discussion

This example is based on example 11 above. Points from that discussion that are unchanged are not repeated.

Line	Commentary
0020	Module "PCKFIELD" is INCLUDEd. This module is required to use mnemonic field type specifications.

Line	Commentary
0030	The PUBLIC section PackFormats is specified in the USES statement. This allows this program to use all PUBLIC variables, functions, and procedures declared as PUBLIC in section PackFormats. The module "PCKFIELDS" has two PUBLIC sections. PackFormats is used to provide mnemonic descriptions for $PACK formats. NDMFormats is used to provide mnemonic descriptions using NDM field types. In this example we are using only the $PACK mnemonics.
0040	The RECORD "cr" is declared as PUBLIC. This is necessary to allow indirect FIELD specification for FIELDs defined in this RECORD. The Amount and Last_Pay field specifications are defined using the 'FieldType$ FUNCTION from PCKFIELDS. Parameters to this FUNCTION are mnemonic PUBLIC CONSTANTS established by PCKFIELDS. This function actually returns the $PACK specification in HEX that is used to format the field. The module PCKFIELDS is not scramble protected - it may be examined.
0150	This illustrates the use of indirect FIELD references. The name of the field being referenced is contained in T$ (set at line 145). The $ in "name$" is not required.
0170	Another example of indirect field reference.
0210	This statement illustrates the use of the new FIELD functions. The attributes of the Name$ FIELD are printed.
0220	The attributes of the Amount FIELD are printed. The field format displays as 5205 - not the mnemonic specified in line 40.
0260	This statement $UNPACKs the Amount FIELD using field functions to determine start position, length, and field format. The field name itself could be contained in a variable. This illustrates the high degree of abstraction that is available using RECORDS/FIELDS.

APPENDIX A

UPGRADING OLDER NPL CODE

A.1 Compatibility with Earlier Revisions

Existing applications require no change to run under NPL Release IV. However, developers are encouraged to spend some time exploring the advantages of NPL's new productivity and functionality under Release IV. Once developers become aware of the features of NPL Release IV, and what those features are capable of, they can selectively implement the features that best suit their applications.

When making changes to older NPL code (developed prior to Release IV), it is important to note that Release IV is fully upwardly compatible with Release III code. Release IV is also downwardly compatible with Release III, provided no Release IV specific features are used.

The remainder of this Appendix addresses specific issues NPL developers should consider when working with NPL Release IV.

A.2 Immediate Benefits of Modularization

Immediate benefits can be achieved by moving subroutines written using the GO-SUB'/DEFFN' interface into a separate subroutine module. This requires that developers identify return value variables, and incorporate these as PUBLIC variables within a PUBLIC section or as POINTERS. In addition, the user-callable entry points in the interface section must be identified.

NOTE: If subroutines use numbered DEFFN's as call-backs, these must also be identified as PUBLIC in the calling program.

Benefits:

- Immediately eliminates the problem of line number conflicts with mainline code.
- Reduces the number of potential conflicts with variable names since variables other than those used to return values are private to the function module.
- The mainline module can freely LOAD various applications without concern for certain line number ranges which contain subroutines.
- Program overlays can resolve faster, since the subroutine module does not require re-resolution at each step.
- Modifications to a mainline module are simplified since the program text no longer includes the subroutine code.
- Subroutine lines no longer interfere with operations like LIST [V,#,',text], RENUMBER, etc.
- Debugging efforts can be focused on a specific module, making TRACE and STEP operations more useful since they can ignore operations in previously tested code.

Refer to Chapter 4 for a complete discussion on working with Release IV's programming constructs.

A.3 Immediate Benefits of Long Variable Names

Long Identifier Names (LINs) improve the readability of application code and, in turn, decrease program maintenance. Programs can easily be revised to use long variable names instead of short ones by using the RENAME V command. In addition, descriptive statement labels may be added to the start of subroutines, allowing them to be called by name instead of by number.

Benefits:

- Code becomes more self-documenting.
- Large modules can be easily created without worrying about variable name conflicts.

Disadvantages:

- Some additional overhead in object file size, memory requirements and load time is imposed by long identifiers.

A.4 Upgrading Code Containing GOTO

Applications that make frequent use of the ON/GOTO conditional statements can be adapted to SWITCH/CASE constructs.

For example, if the old code was as follows:

```
0000 ON X GOTO 100,200,300        :REM Lookup type (assume X=1,2,3 only)
0100 INPUT "Name",A$<
  :IF A$=" " THEN 900<
  :GOSUB 'DoName<
  :GOTO 100
0200 INPUT "Address",A$<
  :IF A$=" "THEN 900<
  :GOSUB 'DoAddress<
  :GOTO 200
 0300 INPUT "ZIP",A$<
  :IF A$=" "THEN 900<
  :GOSUB 'Zip<
  :GOTO 300
0900 REM end case
```

This is not the same as the following incorrect code:

```
0000 SWITCH X      :REM Lookup type (assume X=1,2,3 only)
0100 CASE 1<
  :INPUT "Name",A$<
  :IF A$=" " THEN 900<
  :GOSUB 'DoName<
  :GOTO 100
 0200 CASE 2<
  :INPUT "Address",A$<
  :IF A$=" "THEN 900<
  :GOSUB 'DoAddress<
  :GOTO 200
 0300 CASE 3<
  :INPUT "Zip",A$<
  :IF A$=" "THEN 900<
  :GOSUB 'Zip<
  :GOTO 300
 0900 END SWITCH
```

The correct code is:

```
 0000 SWITCH X     :REM Lookup type (assume X=1,2,3 only)<
  :CASE 1
 0100 INPUT "Name",A$<
  :IF A$=" " THEN 900<
  :GOSUB 'DoName<
  :GOTO 100<
  :CASE 2
 0200 INPUT "Address",A$<
  :IF A$=" "THEN 900<
  :GOSUB 'DoAddress<
  :GOTO 200<
  :CASE 3
 0300 INPUT "Zip",A$<
  :IF A$=" "THEN 900<
  :GOSUB 'Zip<
  :GOTO 300
 0900 END SWITCH
```

Refer to Chapter 4 for a complete discussion on working with NPL's structured constructs.

A.5 Useful NPL Enhancements

In addition to the major features of NPL Release IV, which are discussed in Chapter 4, a series of small yet significant enhancements have also been incorporated. These features, which come directly from NPL developer feedback, are categorized into the following areas and will be implemented on all platform versions of NPL.

- LANGUAGE FEATURES
- EDITOR ENHANCEMENTS
- MISCELLANEOUS ENHANCEMENTS

A.5.1 Language Features

Expandable Arrays. MAT REDIM can now be used to increase the size of arrays. In addition, MAT math statements (MAT * for example) can be used to implicitly increase the size of an array.

MAT SEARCH now allows the use of a numeric scalar or a single numeric array element as the locator variable.

The $PACK format specification for alpha fields (A0xx) has been extended to support field lengths up to 4095 bytes (Axxx).

INPUT and LINPUT now support automatic insert mode where characters typed are inserted rather than overstriking existing characters. This feature can be controlled with a new $OPTIONS byte.

NOTE: Refer to the NPL Statements Guide for details on these enhanced statements.

A.5.2 Editor Enhancements

Single line scrolling of long program lines is now supported. Both scroll-down and scroll-up are implemented. Scroll-up capability may not be present on all supported terminals. Refer to Chapter 5 for details.

Syntax error displays for long program lines will now stop on the first screen with an error.

The display of program lines for resolution or execution error display, STEP mode, TRACE * operations, and LIST * operations has been modified to display only the relevant statement preceded by a series of colons indicating how many previous statements are present on the program line. Previous NPL versions displayed the entire program line. For programs with long program lines, this cluttered the display. Refer to Chapter 6 for details.

Insert Mode is now supported in the editor. When insert mode is selected, characters entered are inserted into the program line instead of overwriting existing characters. The INSERT key can be used to toggle between insert mode and overstrike mode. $OPTIONS Byte 44 is used to control the default behavior of this feature. Refer to the NPL Statements Guide for details.

Stacked entry of items with in-line comments is now permitted on FUNCTION and PROCEDURE declarations, DIM, COM, FIELD, and MAT REDIM statements. Refer to Chapter 5 and the NPL Statements Guide for details.

For example:

```
0010 DIM Record_Counter,          ; comment<
     Record_Buffer$512,           ; used for all files<
     Key_Buffer$32
```

A.5.3 Miscellaneous Enhancements

The start-up banner will no longer appear at the start of an NPL RunTime.

The maximum RunTime user level has been increased to 512 concurrent users.

A new Preboot option allows execution of a specified NPL program prior to loading of the specified boot program. This provides a convenient method of establishing system-wide defaults such as device equivalences and $OPTIONS settings in environments where multiple boot programs are required. Refer to Chapter 2 for details.

In addition, there are many platform-specific enhancements. Refer to the appropriate NPL Supplement.

A.6 Additional Considerations

Niakwa provides a variety of modifiable support files with the NPL RunTime and Development package. These files include:

- Keyboard Translation files
- Screen Translation files
- Printer Translation file
- NPL Error Message file
- Native Operating System Level Error files

Developers should be aware that any customization of these files in earlier versions of NPL should be preserved prior to updating a system to Release IV.

NOTE: With Release IV the NPL error message file has grown significantly. If a customized version of an old error message file is in use, these customizations will need to be incorporated into the updated Release IV error files.

APPENDIX B

NPL ERRORS

B.1 Overview

The Niakwa Programming Language provides a method of reporting programming and execution errors. Many errors are recoverable, while others are not. Some errors may occur at program resolution time, while others do not occur until program execution. This appendix outlines the various types of errors, when they may occur, and possible solutions to the errors.

Section B.2 discusses the various error code classes.

Section B.3 presents the RunTime program errors along with a brief description and suggested action.

Section B.4 discusses the RunTime program warnings.

Section B.5 discusses the RunTime error message files.

Section B.6 discusses the native operating system errors that can effect the NPL operation.

B.2 Error Code Classes

There are nine different error code classes, each categorized by the first digit of the error code. The second and third digits identify the specific error condition that occurred. Errors A00-I99 were originally from the Wang 2200 and supported in NPL. Errors in the 200-799 range are errors available only in NPL, and consequently, are not supported on the Wang 2200.

The following table outlines these nine classes of errors:

Error Code Range	Error Class
A00-A09	Miscellaneous Errors
S10-S29	Syntax Errors
P30-P59	Program Errors
C60-C69	Computation Errors
X70-X79	Execution Errors
D80-D89	Disk Errors
I90-I99	I/O Errors
100-199	Reserved
200-499	Extended NPL Errors
500-799	External Errors
800-999	Reserved

B.2.1 RunTime Program Errors and Warnings

When an error occurs, the terminal beeps, the line and statement at fault are displayed, and the letters "ERR" followed by the 3-digit NPL error code and description appear on the screen. Warnings appear with a "Warning:" prefix, but not code. For multi-line statements, only the faulty statement is displayed, preceded by one of more colons. Each colon indicates a statement preceding the statement on the same line where the error occurred.

For example:

```
1000 ::: X=1/Y
             ^
ERR C62: Division by Zero.
```

indicates the statement X= 1/0 is in error, and this is the fourth statement on line 1000.

B.2.2 Recoverable Versus Non-Recoverable Errors

Some NPL errors are recoverable, while others are non-recoverable. When handling errors under program control, recoverable errors can be trapped, allowing execution of the application to continue. Non-recoverable errors are usually the result of some condition that does not allow program execution and immediately halts the program with the appropriate system error message (these types of errors are usually detected at resolution time). Recoverable errors can only occur once the program or module is fully resolved and is executing. The following table lists all error-code ranges and indicates which ones are recoverable:

Error Code Range	Recoverable or Not recoverable
A00-P36	Not recoverable
P37	Recoverable
P38-P47	Not recoverable
P48	Recoverable
P49-P59	Not recoverable
C60-I99	Recoverable
100-199	Reserved
200-299	Not recoverable
300-499	Recoverable
500-599	Not recoverable
600-999	Recoverable

NOTE: There are exceptions: I90-I99 errors or other errors that may occur when loading a module in an INCLUDE statement may not be recoverable, since this occurs at resolution time, rather than execution time.

Refer to Section 8.5 of the Programmer's Guide for more information on the handling of recoverable errors under program control.

NOTE: Error P37 is not recoverable on a Wang 2200/CS or on releases of NPL prior to Revision 3.0.

B.3 RunTime Program Errors

Error codes consist of a letter prefix, indicating an error class, followed by a two digit number which identifies the specific error condition which occurred. The following section displays all of the NPL error codes and messages, with a brief description of the probable cause and possible solution:

B.3.1 Miscellaneous Errors

These include general environment errors relating to memory allocation and usage.

ERR A00 - Not Implemented.

Under normal circumstances, this error will not occur. It may be generated by the compiler for statements whose syntax is acceptable on a Wang 2200 but is not supported by NPL. For example: $INIT "PASSWORD" In some instances, corrupted p-code may cause this error while loading, resolving, or executing a program.

ERR A01 - Memory Overflow (Text < --> Variable Table).

The program has used up all of the memory available in the user partition. Reduce the size of the program or reduce the dimension of larger variables.

ERR A02 - Value Stack Overflow.

The program stack has overflowed. Too many loops or branches have been pushed onto the stack. Check the number of nested FOR/NEXT loops, or possible "broken" loops that have been abnormally exited but left on the stack. Also, check for possible infinite recursion of FUNCTIONs or PROCEDUREs.

ERR A03 - Insufficient Memory for Buffer.

This error cannot occur under NPL. On a Wang 2200, if the user partition is full, this error can occur when certain disk statements that require a buffer are executed. NPL always reserves sufficient memory to perform these statements.

ERR A04 - Operator Stack Overflow.

The operator stack has overflowed. There are too many complex arguments to evaluate. Try simplifying the expression into multiple statements or use intermediate variables.

ERR A05 - Program Line Too Long.

This error cannot occur under NPL. On some releases of Wang Basic-2, programs exceeding a certain maximum (tokenized) line length could be executed but not saved on disk.

ERR A06 - Program Protected.

An attempt was made to list or edit a program which has been scrambled. Load the unscrambled version to make changes. There is no way to unscramble protected programs.

ERR A07 - Illegal Immediate Mode Statement.

An invalid statement was used in immediate mode (e.g., DATA, DEFFN', or FUNCTION). Refer to Section 2.5.3 of the Programmers Guide for information on the use of certain statements in immediate mode.

ERR A08 - Statement Not Legal Here.

A statement was placed such that the Interpreter cannot compile it (e.g., placing an ELSE statement (using non-structured IF/THEN) on a separate line without using DO/ENDDO groups causes this error). Check the rules of the statement at fault in the Statements Guide.

ERR A09 - Program Not Resolved.

A program was EXECuted or CONTINUEd after being HALTed, but changes were made to its resolve status such that it cannot continue. The program must be restarted with RUN or LOAD RUN.

B.3.2 Syntax Errors

With all syntax errors (S10-S29), the message describes specifically what component is missing from the statement. As such, no detailed cause/solution follows these errors. For all syntax errors, refer to the appropriate statement in the NPL Statements Guide for proper syntax use.

ERR S10 - Missing Left Parenthesis "(".

ERR S11 - Missing Right Parenthesis ")".

ERR S12 - Missing Equal Sign "=".

ERR S13 - Missing Comma ",".

ERR S14 - Missing Asterisk "*".

ERR S15 - Missing "" Character.

ERR S16 - Missing Letter.

ERR S17 - Missing Hex Digit (0-9,A-F).

ERR S18 - Missing Relational Operator (=,<,>,<=,>=,<>).

ERR S19 - Missing Required Word.

ERR S20 - Expected End of Statement.

ERR S21 - Missing Line-Number.

ERR S22 - Illegal PLOT Argument.

ERR S23 - Invalid Literal String.

ERR S24 - Illegal Expression or Missing Variable.

ERR S25 - Missing Numeric-Scalar-Variable.

ERR S26 - Missing Array-Variable.

ERR S27 - Missing Numeric-Array.

ERR S28 - Missing Alpha-Array.

ERR S29 - Missing Alpha-Variable.

B.3.3 Program Errors

These errors include errors in the logical organization or information in a program that would otherwise pass without a syntax error.

ERR P32 - Start greater than End.

Certain disk I/O statements such as COPY or DATA SAVE DC OPEN TEMP require a range of disk sector addresses, in which the starting sector must be less than the end sector. Check the values or the parameters being used.

ERR P33 - Line-Number Conflict.

On RENUMBER statements applied to only a portion of a program, the renumbered set of lines must not conflict with any other unchanged lines. Be sure the range specified, taking into account the step interval, will not overlap with any lines that already exist outside of the range.

ERR P34 - Illegal Value.

The value for a parameter was supplied that is out of range. Refer to the Statements Guide for the proper range of values.

ERR P35 - No Program in Memory.

This error cannot occur under NPL. On the Wang 2200, attempting to run an empty program (no program lines) is diagnosed as an error. NPL permits this.

ERR P36 - Undefined Line-Number or CONTINUE Illegal.

A branching reference was found to a non-existent line number. Replace this line with one that exists.

ERR P37 - Undefined Marked Subroutine.

A GOSUB' was found with no matching DEFFN' available. Check the spelling of the subroutine identifier and be sure the DEFFN' is within the scope of the currently executing module.

ERR P38 - Undefined FN Function.

A reference to a one-line function cannot be resolved. Compare the reference with the definition and be sure the FN number matches.

ERR P39 - FN's Nested Too Deep.

One-line FN functions can only be nested five deep. This is designed to trap any recursion, but the limit still applies, even if no recursion takes place. If recursion is desired, convert the FN function to a structured FUNCTION. If not, simplify the nesting by combining FN functions.

ERR P40 - No Corresponding FOR for NEXT Statement.

A NEXT statement was encountered without a matching FOR statement on the program stack. Be sure the FOR statement exists and nested FOR/NEXT loops are complete. Also check for improper transfer of execution into the inside of a FOR/NEXT loop.

ERR P41 - RETURN Without GOSUB.

A RETURN statement was encountered without a GOSUB command on the program stack or a FUNCTION was called and never encountered a RETURN statement. Check for improper transfer of program execution, such as using GOTO instead of GOSUB. Also, be sure a FUNCTION has a RETURN (...) statement within its body.

ERR P42 - Illegal Image.

The line number supplied in a PRINTUSING statement is not in the proper image format. The line must begin with a "%". Check the line number again, and check for overlays that might accidentally overwrite the image line.

ERR P43 - Illegal Matrix Operand.

Certain matrix operations cannot execute correctly if the result is the same as the operand. This is diagnosed as a syntax error by NPL. Be sure the MAT receiver variable is different from the variables used as operands.

ERR P44 - Matrix Not Square.

The source or receiver array is not a square n by n matrix. Both dimensions must be the same size (i.e., DIM Matrix(10,10)).

ERR P45 - Operand Dimensions Not Compatible.

Different arrays in the MAT operation have different dimensions. Some matrix operations require that the dimensions of operands match in some way. For example, to add two matrices, they both must have the same number of rows and columns.

ERR P46 - Illegal Microcommand.

A $GIO statement was encountered with an invalid microcommand. Check the syntax or the microcommand in the NPL Statements Guide, $GIO.

ERR P47 - Missing Buffer Variable.

A $GIO statement was encountered with no buffer variable specified. Specify an alpha variable that is large enough to accommodate the block of data being read or written.

ERR P48 - Illegal Device Specification

The RunTime cannot open the device as specified. This error is usually accompanied with a native operating system error code. Refer to the appropriate NPL Supplement for further information on these error codes.

ERR P49 - Interrupt Table Full.

This error cannot occur under NPL. On the Wang 2200, SELECT ON/OFF statements that are used to allow servicing device interrupts use a fixed size table, which could be exceeded.

ERR P50 - Illegal Array Dimensions or Variable Length.

An operation such as a MAT statement was performed on an array or string with mismatching sizes between the source and receiver variables. Check the dimensions of the variables.

ERR P51 - Variable or Value Too Short.

The specified receiver variable has been declared with a size too small to be used with this statement (e.g., DATA LOAD BA requires that the buffer (receiver) variable be at least 256 bytes in size). Check the dimensions of the variable in question.

ERR P52 - Variable or Value Too Long.

The specified string parameter (or value) in the statement is too large to be used. For example, in a DATA LOAD DC OPEN statement, the filename specified (either literally or indirectly) may not exceed eight characters. Shorten the value to the required maximum length.

ERR P53 - Noncommon Variables Already Defined.

A variable was declared as COM after other variables have been declared with DIM. All COM statements must appear before all DIM statements or any reference to non-common variables.

ERR P54 - Common Variable Required.

The multiple-program LOAD statement requires that the variable that specifies the list of program names must be a COM variable. Change the declaration from DIM to COM.

ERR P55 - Undefined Variable.

A reference was made to a variable that has not been declared with a DIM or COM statement (which is a requirement of certain MAT statements, or if Byte 38 of $OPTIONS has been set to HEX(01)). Add the variable to a DIM or COM statement, or check the value of Byte 38 of $OPTIONS.

ERR P56 - Illegal Subscripts.

A subscript was specified that exceeded the dimensioned size of the array. Check the DIM statement, or the method of calculating the subscript.

ERR P57 - Illegal STR Arguments.

The start or length parameter specified in the STR() function is invalid or out of range. Compare the size of the variable with the values specified. Refer to the STR() Function in the NPL Statements Guide for more details.

ERR P58 - Illegal Field/Delimiter Specification.

When using $PACK/$UNPACK, the pack specification bytes are invalid. Check the field form or delimiter form specification. Refer to $PACK or $UNPACK in the NPL Statements Guide.

ERR P59 - Illegal Redimension.

A variable in a DIM statement was encountered after the variable had already been declared in a previous DIM statement.

B.3.4 Computational Errors

These errors are associated with mathematical calculations or expressions.

ERR C60 - Underflow.

The value has diminished below the minimum allowable by NPL.

ERR C61 - Overflow.

The value has exceeded the maximum allowable value by NPL.

ERR C62 - Division by Zero.

When dividing, the divisor cannot be zero.

ERR C63 - Zero Divided by Zero or Zero ^ Zero.

Special cases where zero values are illegal.

ERR C64 - Zero Raised to Negative Power.

Mathematically the same as dividing by zero.

ERR C65 - Negative Number Raised to Non-integer Power.

Roots (including square root) of negative numbers are imaginary, thus are illegal.

ERR C66 - Square Root of Negative Value.

The square root of a negative number is imaginary, thus illegal.

ERR C67 - LOG of Zero.

Cannot take the logarithm of zero.

ERR C68 - LOG of Negative Value.
Cannot take the logarithm of a negative number.

ERR C69 - Argument Too Large.
Certain math functions can only accept arguments within a defined range. For example, the ARC SIN () function requires an argument in the range -1 to +1. Check the value being passed to the function.

B.3.5 Execution Errors

These errors are the most common run time errors that occur once program execution begins, and usually relates to improper variable handling.

ERR X70 - Insufficient Data.
A READ statement was encountered, but all of the DATA statements have been read. No more data is available to read. Check for sufficient DATA statements or use the RESTORE statement to reset the DATA pointer to a specified value.

ERR X71 - Value Exceeds Format.
The value contained in the variable cannot be stored in the format specified (e.g, in a $PACK statement). Use a different field-form pack specification.

ERR X72 - Singular Matrix.
The source array supplied for a MAT operation is one-dimensional. It must be two-dimensional with equal (square) n by n dimensions. Be sure the correct array is being used, or in certain instances,

ERR X73 - Illegal INPUT Data.
INPUT operations with strings normally don't permit commas to allow multiple answers to be entered on the same line. However, the entered value may also be placed in quotes, to allow a value to contain a comma in it. The there is no matching quote, this error occurs. Be sure an ending quote is entered following the comma to contain it in the string value.

ERR X74 - Wrong Variable Type.
The variable type does not match the field-form pack specification of a $PACK/$UNPACK statement. Check to make sure the variables are being packed or unpacked in the proper order and that the pack specification is correct. Also check the buffer variable being used.

ERR X75 - Illegal Number.

The value that was input or received is not a valid number. Check to make sure alphabetic characters are not being placed into numeric variables.

ERR X76 - Buffer Exceeded.

The buffer variable supplied in the statement is not large enough to accommodate the data placed into it (e.g., a $PACK or $UNPACK statement). Change the dimension of the buffer variable to a larger size, or reduce the amount of data placed into it.

ERR X77 - Invalid Partition Reference.

A reference has been made to a #PART value that is invalid or is not accessible from the current foreground partition. If using background partitions, the specified partition number in $RELEASE TERMINAL is not active.

ERR X79 - Invalid Password.

A modification to the DATE or TIME variables was made with an invalid password. For syntactic compatibility, NPL supports only the password "SYSTEM" as default. Generally, no PASSWORD parameter is necessary to change the date or time from within NPL.

B.3.6 Disk Errors

These errors are associated with disk and catalog operations such as opening and closing data files, and loading and saving of program files.

ERR D80 - File Not Open.

An attempt was made to read data from a file that has not yet been opened. Check the file number in the device table. Use DATA LOAD DC OPEN to open a previously cataloged file for further processing.

ERR D81 - File Full.

The space in the file currently allocated in the catalog has been used. The file must be moved to the end of platter and the old file scratched.

ERR D82 - File Not in Catalog.

An attempt was made to open a file that is not listed in the catalog. Check the spelling of the filename and check the device address or file number being used.

ERR D83 - File Already Cataloged.

An attempt was made to create or save a file that already exists. For programs, use RESAVE instead of SAVE. For data, use DATA LOAD to open a file instead of DATA SAVE and make sure the data file and file number are specified correctly.

ERR D84 - File Not Scratched.

An attempt was made to establish a new data file using an old data file's allocation (the old file has not been scratched). If the "scratch-file" is no longer needed, use SCRATCH T to scratch it, then try again. Otherwise, specify a file that has already been scratched.

ERR D85 - Index Full.

The catalog index for the diskimage or platter is full and cannot accommodate new entries. A new diskimage must be created with a larger index size to accommodate more files.

ERR D86 - Catalog End Error.

A file was attempted to be saved, but the file would go beyond the end of catalog. Try removing older files or move the end of catalog to accommodate the new file.

ERR D87 - No End-of-File.

A DSKIP END statement was encountered, but no valid end-of-file was found. This statement requires that a valid end-of-file mark be saved on the file at the position indicated when the last DATA SAVE DC END statement was done. Check to make sure a DATA SAVE DC statement didn't overwrite the end-of-file, or that a DATA SAVE DC END was executed.

ERR D88 - Wrong Record Type.

When reading data, the format is unreadable or not delimited correctly. When loading programs, the program may be a non-NPL program or the p-code is unreadable by NPL. If the program is in Wang 2200 atomized code format, it must be compiled before it can be loaded. Otherwise the program may be damaged or missing a required password.

ERR D89 - Sector Address Beyond End-of-File.

Access to a file caused access to sectors outside the range of the file. This is likely to occur if the end-of-file mark is corrupted. Check the file, and restore the end-of-file (possibly even data, if corrupted.)

B.3.7 I/O Errors

These errors are rather esoteric, but are likely to be accompanied by a native operating system error code. Refer to the specific hardware supplement for more information on the native error codes.

ERR I90 - Disk Hardware Error.

A disk I/O operation reported to NPL of a failure. This error is usually accompanied with a native operating system error code. Refer to the appropriate NPL Supplement for further information on these error codes.

ERR I91 - Disk Hardware Error.

A disk I/O operation reported to NPL of a failure. This error is usually accompanied with a native operating system error code. Refer to the appropriate NPL Supplement for further information on these error codes.

ERR I92 - Timeout Error.

A disk I/O request was made, but no response was received from the native operating system. Check for possible hardware failures or corrupted files.

ERR I93 - Format Error.

The raw diskette format does not match that specified by its device equivalence or the floppy disk format is incompatible. Check the $DEVICE statement or try reformatting the floppy disk.

ERR I94 - Format Key Engaged.

This error cannot occur under NPL. On the Wang 2200, certain drives had a key-operated format option. While this was active, access to the drive was not permitted.

ERR I95 - Device Error (Write Protect).

The device to be written to is currently write protected. Change the write-protect status (the notch on a diskette) to enable writing.

ERR I96 - Data Error.

This error cannot occur under NPL. CRC data integrity errors would normally be reported by the native operating system.

ERR I97 - Longitudinal Redundancy Check Error.

This error cannot occur under NPL. LRC data integrity errors would normally be reported by the native operating system.

ERR I98 - Illegal Sector Address or Platter Not Mounted.

The specified sector is beyond the end of the physical diskimage or file. Check the value specified against the highest sector available for that file.

ERR I99 - Read-After-Write Error.

An error has occurred while the program or data was being saved with a $ parameter. The verification with the read-after-the-write of the file failed. Check the integrity of the file, the diskimage, or save again so that the file is placed elsewhere on the platter.

B.3.8 NPL Extended Errors

These errors are unique only to NPL, and do not occur on the Wang 2200.

ERR 200 - Handle table full / cannot expand.

NPL's internal handle table could not be dynamically resized due to memory limitations. Extra space can be pre-allocated at startup by using the /H option to specify handle table size. Refer to Section 2.4 for information on the /H option.

ERR 201 - Variable already referenced in program.

A RENAME V/FIELD/PROCEDURE command refers to a new name that is already used in the current module. If the uses of the old variable/identifier and the new are identical, the MERGE option may be used to override this check.

ERR 202 - Variable not referenced in program.

A RENAME V/FIELD/PROCEDURE command refers to an old name that is not used in the current module. An existing identifier name must be specified as the old name.

ERR 203 - Branch into body of a function or procedure.

A GOTO or GOSUB to a line in the middle of a FUNCTION or PROCEDURE is not legal and is flagged as an error at resolution time. The entire function must be called.

ERR 204 - Branch out of body of a function or procedure.

A GOTO or GOSUB within a FUNCTION or PROCEDURE to a line outside of it is not legal and is flagged as an error at resolution time. A function must be exited normally.

ERR 205 - This error code is reserved for future use by NPL.

ERR 206 - FUNCTION/PROCEDURE not matched with END FUNCTION/PROCEDURE.

A FUNCTION or PROCEDURE declaration was found, but no END FUNCTION or END PROCEDURE was found. Functions cannot be nested and the END FUNCTION statement must appear after the declaration. If declaring a function that appears later in the program, use the /FORWARD parameter.

ERR 207 - Nested functions/procedures not permitted.

A FUNCTION or PROCEDURE cannot occur within another FUNCTION or PROCEDURE. All structures must occur at the same level and can call each other as long as the called function or procedure is declared before the calling function or procedure.

ERR 208 - Function/ procedure already defined.

A FUNCTION or PROCEDURE already exists with the same identifier. Use a different identifier or change the scope such that the functions are "private" to each other (i.e., different modules.)

ERR 209 - Function/ procedure differs from /FORWARD declaration.

The declarations in the /FORWARD line does not match with the actual declaration. Check the parameter list and types.

ERR 210 - RETURN(value) not within FUNCTION body.

A RETURN(value) statement was encountered outside of a FUNCTION body. This statement must appear within the FUNCTION body. If a GOSUB was used, use RETURN without a value.

ERR 211 - RETURN(value) incorrect type for FUNCTION.

An attempt was made to return a string value for a numeric function or a numeric value for a string function. Check the RETURN statement and be sure the value type matches the FUNCTION type.

ERR 212 - End Structure name does not agree.

The appropriate structure END statement has an identifier specified, but does not match with the most recent structure declaration (FUNCTION, PROCEDURE, RECORD, etc.). Check the spelling of the identifier.

ERR 213 - Arguments of function/procedure incorrect.

The variables passed to the function do not match with the function declaration either by type or number of parameters. Check the function/procedure declaration for the correct parameters needed.

ERR 214 - No such function or procedure has been declared or included.

A call or reference has been made to a non-existent FUNCTION or PROCEDURE. Check the spelling of the identifier, make sure the module is properly INCLUDEd, and that the subroutine has been declared before the reference. If a function reference must appear before the function body, use the /FORWARD declaration.

ERR 215 - Argument for non-constant /POINTER parameter is not modifiable.

A constant or literal was passed to a /POINTER variable, thus the reference back to the original entry cannot modify its contents. When using /POINTER, avoid passing constants to variables where modification may be done. Use constants in the parameter list when no modifications will be made.

ERR 216 - Not legal after an :ERROR clause.

A statement that defines the start or end of a FUNCTION or PROCEDURE is not permitted after an ERROR statement on the same line, since these statements could result in implied branches into or out of a function body. This does not apply if an ERROR DO statement is used.

ERR 217 - Only simple IF permitted here.

A structured IF statement cannot immediately follow a THEN or ELSE clause of a simple IF statement. Be consistent with using either simple IF or structured IF statements. If a structured IF statement must be nested within a simple IF, use the DO...END DO structure (simple IF/THEN/ELSE statements can be nested within structured IF statements).

ERR 218 - This error code is reserved for future use by NPL.

ERR 219 - This error code is reserved for future use by NPL.

ERR 220 - No enclosing WHILE, REPEAT or FOR/BEGIN loop structure.

A BREAK or LOOP statement was encountered outside of these structured loops. These statements are only valid for these three types of structures. BREAK unconditionally exits the loop, while LOOP skips to the next iteration of the loop.

ERR 221 - This error code is reserved for future use by NPL.

ERR 222 - Structured IF not matched with END IF.

The structured IF or END IF statement was encountered, without any matching IF or END IF statement, or a structure form was mixed in with the simple IF/THEN/ELSE form. If using simple IF/THEN/ELSE, no structured ELSE or END IF is allowed.

ERR 223 - Structured ELSE not matched with END IF.

A structured ELSE clause was encountered, but no END IF statement exists. When using structured IF statements, the structure must end with END IF. Avoid mixing of simple IF and structured IF syntax.

ERR 224 - REPEAT not matched with UNTIL.

A REPEAT statement was encountered with no matching UNTIL expression, or an UNTIL was encountered with no matching REPEAT. The UNTIL expression must follow the REPEAT statement and exist together in the same structure.

ERR 225 - WHILE not matched with WEND.

A WHILE expression was encountered with no matching WEND statement, or a WEND statement was encountered with no matching WHILE expression. The WEND statement must follow the WHILE expression and exist together in the same structure.

ERR 226 - FOR/BEGIN not matched with NEXT.

A structured FOR...BEGIN statement was encountered with no matching NEXT statement found. If nesting loops, be sure equal number of NEXT's exist for the loops. The entire FOR..BEGIN...NEXT loop must exist completely within a structure.

ERR 227 - Recursive/parameter value not legal in declaration.

The expression in a declaration cannot contain recursive variables (e.g., a DIM statement private to a FUNCTION, recursive variables cannot be used to define the variable size or number of elements). Use STATIC or PUBLIC variables in the declaration.

ERR 228 - Variable already declared with different access.

A variable was attempted to be declared, but already exists in the current scope (e.g., a variable cannot be declared STATIC within the function body, if it already exists as RECURSIVE). The static declaration should appear outside the function body, or use a different name.

ERR 229 - Local variable already declared.

A DIM /STATIC variable declaration was encountered, but the variable in question is already declared as STATIC in the current scope. If in the mainline, a variable can be declared STATIC with the same name only if within a function body.

ERR 230 - Local function already declared.

DEF FN() statements within a function body are local to the function. Like any other identifiers, only one declaration of a specific FNx() function is permitted within a function body.

ERR 231 - /BEGINS declaration not preceded by /FORWARD declaration.

An unresolved FUNCTION declaration was encountered with the /BEGINS parameter, but the function must already be resolved with a previous /FORWARD declaration (as required by the /BEGINS parameter). Be sure a /FORWARD declaration exists before the defining declaration. Removing the /BEGINS parameter also avoids this error.

ERR 232 - Structured statements not allowed as THEN or ELSE clause.

A structure (SWITCH, Structured IF, WHILE, REPEAT, etc.) was encountered following the THEN or ELSE clause of the simple IF statement. Convert the simple IF...THEN...ELSE logic to use the structured IF...ELSE...END IF logic. If the simple IF statement must be used, enclose the structure within a DO...END DO structure.

ERR 233 - Recursive arrays cannot be redimensioned.

Arrays that are passed as parameters or declared as local variables within a function cannot be dynamically redimensioned if used recursively. Here, the array should be dimensioned to the size necessary for function use and should be passed as a /POINTER parameter type.

ERR 234 - CASE located after default CASE for this SWITCH.

An additional CASE with expression was found after the default CASE. The default CASE must always be the last CASE within a SWITCH. Since the default CASE signifies "all other values" no further testing of the SWITCH variable would be desired afterward.

ERR 235 - SWITCH not matched with CASE/END SWITCH.

A SWITCH statement was encountered with no matching CASE statement(s) or END SWITCH statement, or a CASE or END SWITCH statement was encountered with no matching SWITCH statement. Be sure the structure is completely within any other structure.

ERR 236 - CASE expression not same type as SWITCH.

The CASE expressions are numeric while the SWITCH is alpha (string) or vise-versa. The expression must match type for any of the CASE expressions to be true.

ERR 237 - FUNCTION/FNx() term not legal in declarations.

The expression containing a function is not legal. For example, a variable declaration in a DIM statement cannot use a FUNCTION value to define the variable size or number of elements unless the FUNCTION is declared in a previously INCLUDEd library module.

ERR 238 - Conflicting options in function declaration.

Options specified after the function declaration do not match those with the /FORWARD declaration. Parameters such as /PUBLIC must appear with both declarations.

ERR 239 - /FORWARD declaration not followed by /BEGINS declaration.

A FUNCTION or PROCEDURE declaration was made with the /FORWARD parameter, but no actual declaration follows it in the program. Check that the subroutine identifier name is spelled correctly and is of matching type (string or numeric.)

ERR 240 - Invalid module name.

No such module is currently loaded. Only INCLUDEd modules can be accessed. Verify the correct spelling of the module name. Use LIST DT to view a list of currently INCLUDEd modules.

ERR 241 - Circular INCLUDE/USE references.

A module or public section is referencing back to the module or public section that originally had the INCLUDE or USES statement. This resulted in an infinite loop.

ERR 242 - Internal Error.

An internal error has occurred in the RunTime, and the integrity of the programs in memory is in question. Do not edit or run the programs anymore. Completely exit and restart NPL.

ERR 243 - No EXTERNAL function is defined with this name and these parameters.

A FUNCTION declaration with the /EXTERNAL parameter was encountered, but it cannot find a matching label or parameter list in the external library. Check to make sure the library is loading properly (if using /X) and check the external routine again to make sure the label name and parameter types match.

ERR 244 - Already defined as Non-public Variable.

A variable was attempted to be declared as PUBLIC, but was already declared in a DIM statement outside of a PUBLIC section or without /PUBLIC within the current scope of the executing module.

ERR 245 - Already defined as PUBLIC function.

A FUNCTION was declared as PUBLIC, but a FUNCTION already exists in an INCLUDEd module, using the same identifier, and is PUBLIC. Use a different identifier to avoid conflicting names. Use LIST PUBLIC FUNCTION to show all current declarations.

ERR 246 - Already defined as PUBLIC section.

A PUBLIC section was defined, but a PUBLIC section already exists in an INCLUDEd module with the same identifier. Use a different PUBLIC section identifier.

ERR 247 - PUBLIC not matched with END PUBLIC.

A PUBLIC section header was encountered with no matching END PUBLIC statement, or and END PUBLIC statement was encountered with no matching PUBLIC statement. The END PUBLIC statement must appear afterward and have a matching public section identifier to the declaration.

ERR 248 - No such PUBLIC section.

A USES statement is referencing an invalid PUBLIC section. Check to make sure the public section identifier is spelled correctly and that the module that contains the public section is properly INCLUDEd and resolved.

ERR 249 - Another procedure is already marked as /MAIN, /EXIT.

A module was found to contain more than one PROCEDURE with one these parameters specified. Only one PROCEDURE declaration per module is allowed for each type of parameter.

ERR 250 - This error code is reserved for future use by NPL.

ERR 251 - Forced error exit to /EXTERNAL from callback.

While executing in a callback from an external function, certain operations (program LOAD or edit operations, CLEAR V) would clear the execution stack and result in never returning to the external library. In such circumstances, attempts to execute these statements result in a forced RETURN ERROR (251) to the pending function, to allow the caller to clean up.

ERR 252 - Callbacks not available.

External routines are not always permitted to call into NPL. In particular, callbacks may not be permitted at interrupt time or in response to Windows messages which occur while an NPL program is active. Any attempts to callback to NPL at inappropriate times are refused with the equivalent of a RETURN ERROR (252) from the callback routine.

ERR 253 - Nested RECORD declarations not permitted.

A new RECORD declaration cannot occur between another RECORD/END RECORD declaration. All RECORDs must be declared at the same level.

ERR 254 - RECORD not matched with END RECORD.

A RECORD declaration was encountered, but no matching END RECORD statement was found. The entire RECORD structure must exist completely within another structure.

ERR 255 - RECORD exceeds maximum allowed length.

ERR 256 - FIELD statement not in RECORD declaration.

A FIELD statement was encountered outside of a RECORD/END RECORD structure. All field declarations must be enclosed within RECORD declarations.

ERR 257 - PUBLIC section is not in an INCLUDEd module.

A USES statement was encountered and referenced a PUBLIC section that is not available. The module containing the PUBLIC section may not be INCLUDEd. Check the INCLUDE statement, and be sure the section identifier is spelled correctly.

ERR 258 - This error code is reserved for future use by NPL.

ERR 259 - Statement label already defined.

There cannot be multiple statement labels with the same label identifier together in a mainline or together in a function body. For example, in the mainline, multiple labels cannot have the same name unless all but one of the identical labels are out of scope (i.e., each in a different function body or module.)

ERR 260 - No such statement label in mainline.

An illegal branch to a non-existent statement label was encountered in the main program. In the mainline, only statement labels declared within the currently executing module can be accessed. Statement labels within FUNCTION or PROCEDURE bodies cannot be accessed. Check the labels with the LIST = command.

ERR 261 - No such statement label in same FUNCTION/PROCEDURE body.

An illegal branch to a non-existent statement label was encountered within a FUNCTION or PROCEDURE. Within the FUNCTION or PROCEDURE body, only those statement labels declared within that same body are accessible. Statement labels in other functions or the mainline cannot be accessed. Check the labels with the LIST = command.

ERR 262 - FUNCTION used by declaration did not complete.

While executing in a library function used in a declaration of a module (during a RUN or LOAD statement,) certain operations (program LOAD or edit operations, CLEAR V) would clear the execution stack and result in never completing the module's resolution. In such circumstances, attempts to do these statements result in a forced RETURN ERROR (262) from the pending function, to allow NPL to clean up a partially resolved module.

ERR 263 - DIM/RECURSIVE must be in a FUNCTION/PROCEDURE body.

A variable was declared with the /RECURSIVE parameter outside of a function body. Explicit /RECURSIVE declaration of variables is only valid inside the body of a FUNCTION or PROCEDURE.

ERR 300 - Incorrect number or types for parameters of GOSUB'.

A DEFFN' with more than 16 parameters was called with the wrong number of parameters or mismatched variable types in the GOSUB' statement. Refer to the DEFFN' statement to determine the proper number of parameters and their variable types.

ERR 301 - Insufficient memory or bad COMSPEC for $SHELL.

The $SHELL statement could not be started. Check the amount of available system memory and that the native shell program (i.e., COMMAND.COM, /bin/sh, etc.) is accessible by the RunTime. Refer to Chapter 8 of the appropriate NPL Supplement for specific requirements of $SHELL.

ERR 302 - No such PUBLIC FIELD name.

An indirect reference was made to a FIELD name that does not exist in a PUBLIC RECORD. Check the variable containing the indirect reference. Be sure all RECORDs that are PUBLIC are accessible via an INCLUDE or USES statement.

ERR 303 - No such PUBLIC PROCEDURE/FUNCTION name.

An indirect reference was made to a FUNCTION or PROCEDURE identifier that does not exist in any PUBLIC section. Check the variable containing the indirect reference. Be sure all functions that are PUBLIC are INCLUDEd and resolved.

ERR 304 - Cannot expand array variable.

There is not enough memory available to expand the allocation of the specified array variable with MAT REDIM. Use the SPACEF variable to examine the amount of free memory. The array may be expandable up to the amount of available memory.

B.4 RunTime Program Warnings

These are generated by the Interpreter only (RTI). When generated, no code is displayed, just the "Warning: ..." message. These are provided for informational purposes. Although these are not errors, the warning should be heeded or unexpected side-effects may occur when executing the program.

WARN 01 - Warning: Programs merged.

A LOAD or LOAD BOOT statement was issued with a program already residing in memory. The new program merges or overlays on top of any existing program, which may or may not be desired. A CLEAR or CLEAR P statement can be used to erase all programs in memory prior to LOADing another.

WARN 02 - Warning: Program moved to end of platter.

A program has been RESAVEd but cannot fit in the space previously cataloged to it. The program is saved at the end of the platter at the next available contiguous open space and the old program is deleted.

WARN 03 - Warning: RUN statement is in progress.

The program has been halted before the last issued RUN or LOAD statement has completed program resolution. This usually occurs if a library function used in a declaration performs a STOP or otherwise is halted before it completes. It can also occur if an INCLUDEd library module has a procedure with the /MAIN parameter and performs a STOP or is halted before it completes. Whenever possible, the halted function or procedure should be allowed to complete normally, so that program resolution can be completed. If this is not possible, an immediate mode RETURN ERROR (x) command should be used to indicate that the function/procedure cannot complete.

B.5 Error Message Files

The NPL Runtime Package supplies two files which contain the literal text for the error codes which are generated. Refer to Section 8.5 for more information.

B.6 Operating System Level Errors

Besides errors that may occur within the RunTime, errors may also occur outside of the RunTime. Native operating system errors can sometimes occur, even causing NPL errors to occur. On most platforms, an error code is generated by the operating system that NPL looks up in an error table (much like one for NPL errors) for the appropriate message. The RTIxERR.HLP file (where "x" is the platform code) contains the literal text for native operating system error codes. The text is displayed by the RunTime in the event of a nonrecoverable error. The associated text is also returned by the $OSERR alpha function. This file must be present in the appropriate operating system directory where the Runtime is installed to generate this text.

NOTE: The file names and associated error messages are unique to each hardware version of NPL (i.e., File Not Found, Access Denied, etc.). Please refer to the specific hardware supplement for the appropriate naming conventions used.

Other external operating system conditions may force NPL to display an "Error Screen". This looks very similar to the NPL Help Screen, but reports the error as text in the middle of the screen. Errors are usually associated with timeout of a hardware device such as a parallel or serial port, or network connection. Options are to "Leave Help" and execution can be attempted to continue, or, depending on system settings, options normally available in the Help Screen can be selected. Refer to Chapter 11 in this manual or Chapter 4 or the appropriate NPL Supplement for the appropriate errors.

APPENDIX C

COMPILER ERRORS

C.1 Compiler Options Errors

The following diagnostic messages are generated by the compiler when compiler options or source program names are incorrectly specified:

Cannot open LSTLOC (or ERRLOC) file

Explanation: Either an invalid path name has been specified, or there is not enough room on the disk to create the specified file, or the directory is full, or the specified filename contains characters which are invalid in a native operating system file name.

Possible Solution: Check the available space on disk, and be sure the path and filename are legal.

Cannot open SRCLOC (or OBJLOC) as a diskimage file

Explanation: An improper specification for SRCLOC (or OBJLOC) has been given. Either the specified pathname does not exist, the diskette specified is of the wrong format ("raw" diskette in drive when directory is specified), or for some other reason the native operating system has returned an error code when the compiler attempted to open the specified file.

Possible Solution: Check the path and filename. If using raw diskettes, be sure to have the correct drive specification.

Extensions not permitted

Explanation: The compiler can only access ASCII text files with a .SRC extension as source files. Any other extension is invalid.

Possible Solution: Rename the ASCII file with the .SRC extension. The extension is not necessary in the compiler parameter.

Fields may not contain embedded spaces

Explanation: Refers to the compiler display only. No field except the Source Programs field may contain embedded spaces.

Possible Solution: Specify filenames without spaces.

Filename too long

Explanation: Filename was specified incorrectly. It can only be a maximum of eight characters.

Possible Solution: Correct the filename specification.

LSTFORMAT option must be one of; .LST, .SRC, 2200, 2200S

Explanation: Parameter was entered incorrectly, only those four are possible.

Possible Solution: Correct the LSTFORMAT specification.

Missing Option after (literal)

Explanation: An option (literal) has been specified, but a required parameter has not been provided.

Possible Solution: Add the appropriate parameter after the option, or remove the option from the command line. Refer to the Programmers Guide for syntax.

No source programs specified

Explanation: No wildcard program names have been specified or no source programs have been found that match the wildcard(s) specified.

Possible Solution: Check the source program location, and be sure the wildcard specification indeed matches the subset of files desired.

Non-Hex digits in option

Explanation: Digits other than 0-9, A-F were specified. The TRANSLATE option requires pairs of hex digits (e.g., 5F20).

Possible Solution: Correct the option with proper HEX digits.

OBJFORMAT must be one of; SCRAMBLED, UNSCRAMBLED

Explanation: One of the two keywords was spelled incorrectly, or some non-keyword was specified.

Possible Solution: Correct the option with only one of the two possible keywords.

Option is not a number

Explanation: Non-numeric characters were specified. The OBJEXTRA option must be digits only.

Possible Solution: Correct the option by entering a number.

Option must be one of; ON, OFF

Explanation: Refers to WARNINGS, LCASE, NUMBERS, REM$, and DISPLAY options. Only the keywords ON or OFF can be used.

Possible Solution: Correct the option by entering either ON or OFF following the option.

Option too long

Explanation: The TRANSLATE option can have a maximum of 30 pairs of hex digits.

Possible Solution: Shorten the HEX string to at most 30 characters.

SRCLOC (or OBJLOC) is not a diskimage file

Explanation: The specified SRCLOC (OBJLOC) has been opened, but the internal format of the file is not that of a 2200 diskimage.

Possible Solution: Make sure the file is an actual diskimage. If the source is ASCII files, specify a pathname rather than a filename.

Too many parameters on command line

Explanation: A maximum of 20 parameters is allowed. Parameters include options, the parameters for options, and wildcards.

Possible Solution: Shorten the parameter list so that no more than 20 appear.

Unknown option on command line (literal)=(parameter)

Explanation: An option has been specified that is not known to the compiler. The option is displayed in the (literal) along with the (parameter).

Possible Solution: Check the spelling of the option keyword.

The following compiler messages are generated by the compiler during execution:

Diskimage index is full

Explanation:	The compiler is attempting to add a newly compiled program to the OBJLOC specified, but found that the diskimage index was full. The program is not compiled.
Possible Solution:	Specify a diskimage that has available index space to accommodate the newly compiled programs.

Filename in catalog is not a program

Explanation:	The compiler has found that the filename being compiled already exists in the OBJLOC but is not a program file. The file is not overwritten. The program is not compiled.
Possible Solution:	The destination file can be COPY'd to a new filename then scratched.

No names found in (location) for wildcard (wildcard)

Explanation:	The compiler found no source files matching the specified wildcard in the SRCLOC specified.
Possible Solution:	Check the pathname and be sure the source files are available, and the wildcard includes the files desired.

Object program was moved to end of catalog in object diskimage

Explanation:	The compiler was attempting to overwrite an existing object file in the OBJLOC diskimage, but found that the newly compiled program did not fit in the old space. The program was successfully relocated to the end of catalog.
Possible Solution:	No action necessary. Message is just informative.

Out of space in Object diskimage

Explanation: The compiler has found insufficient space in the OBJLOC diskimage to place the compiled program. This may result when compiling a new program or when compiling a program which already exists but has expanded. The program specified when this message is issued has NOT been completely compiled and is not executable.

Possible Solution: Use NPL and issue a MOVE END command to expand the size of the diskimage, or specify a diskimage that has enough available space.

C.2 Program Syntax Errors

The compiler displays program syntax errors by displaying the source program line where the error occurred, and on the following line, an up arrow points to the approximate location of the error, followed by a three position NPL error code.

However, the compiler continues syntax checking on the balance of the program. The partial object program on disk is marked as non-executable.

The compiler continues normal compilation of the balance of programs specified in the source programs list.

C.3 Program Syntax Warnings

Compiler WARNINGS are generated for certain program statements which may have RunTime variances under NPL as compared to Wang 2200 BASIC-2. These warnings are not meant to be totally comprehensive. Rather, they are selectively issued for statements which may have to be rethought. There may well be other statements, as documented in the NPL Statements Guide, which have variances which affect the operation of the application.

These warnings may be suppressed by the WARNINGS OFF compiler option. Compiler warnings do not affect the production of object code.

ALL(x) & X$ is illogical syntax, likely an error

Explanation: Refer to the NPL Statements Guide, "ALL" for details.

Cannot do RESTORE LINE / DATA checks due to table overflow

Explanation: In order to check RESTORE LINE statements against DATA statements, the compiler uses a table with a maximum of 200 entries. This message means that more than 200 DATA lines have been found in the module being compiled. This means that the warning "RESTORE LINE does not point to DATA statement in this module" can no longer be issued for this module (refer below).

ELSE clause not preceded by IF statement

Explanation: This also is valid but illogical syntax.

ERROR clause not preceded by statement

Explanation: This is valid, but illogical syntax.

Programs which use SPACE(K) may need adjustment ... under "THIS O/S"

Explanation: Refer to the Section 3.2 of the Programmer's Guide for details.

RESTORE LINE does not point to DATA statement in this module

Explanation: This statement results in a RunTime error when executed unless an overlay module containing a DATA statement with the specified LINE number is loaded.

SAVE DA/DC is not supported at run time

Explanation: Refer to Section 1.5.1 of the Programmer's Guide for details.

Statement requires RTP Rev x.xx.xx or later to execute

Explanation: This indicates that there is a program statement which is new for the revision specified in the warning and does not execute on prior versions of the RunTime. The revision specified in the warning and does not execute on prior versions of the RunTime.

APPENDIX D

NPL DIRECTLY SUPPORTED TERMINAL CHARACTERISTICS

D.1 Overview

This appendix concerns itself with the terminals which are supported under the various versions of NPL. Inclusion of a terminal in this chapter does not guarantee that it operates under all supported NPL environments. Please refer to the appropriate NPL Supplement for details on terminals supported under each operating system.

NOTE: Consoles and monitors are documented in the appropriate Supplement due to their operating environment/hardware dependence.

Section D.2 discusses terminal identification, supported terminals and local printer designation using NPL.

Sections D.3 through D.16 discuss considerations for each NPL supported terminal.

D.2 General Considerations

The following section discusses general considerations of NPL supported terminals.

D.2.1 Determination of Terminal Type

On all versions of NPL, the terminal type is determined by a system variable or disk based table (i.e., BASIC2C_TERM under Xenix, TERMTYPE.TBL under SuperDOS, etc.). Refer to the appropriate NPL Supplement for details of terminal type determination.

D.2.2 Supported Terminals

As of Revision 4.00, the following terminals are directly supported by the NPL RunTime and have their supporting files distributed on the NPL Development Package Utilities diskette. Each supported terminal has a value assigned to it in byte 9 of the $MACHINE variable. This value is consistent across all operating systems that support the terminal. For details on which terminals are supported under a specific operating system, refer to Chapter 6 in the appropriate NPL Supplement.

TERMINAL	$MACHINE BYTE 9 VALUE
Altos III	HEX(02)
Altos V	HEX(08)
Bull HDS 1	HEX(06)
Bull HDS 3	HEX(03)
DEC VT100	HEX(03)
DEC VT200 Series	HEX(03)
IBM 3151	HEX(0D)
NCR 4970	HEX(03)
Spectrix SPX 701	HEX(0C)
Wang 2110a	HEX(01)
Wang 2x36 DE/DW	HEX(07)
Wyse 50	HEX(06)
Wyse 60,150, 160	HEX(05)
Wyse 370	HEX(0B)

D.2.3 Support of Native Operating Systems and Utilities

Each operating system supported by NPL supports specific terminals. Not all supported NPL terminals are supported on all operating systems. Refer to Chapter 6 of the appropriate NPL Supplement for details on specific terminals supported by NPL on that operating system and for details on which terminals are well suited for use with native operating system functions or applications.

D.3 Altos III

The following section details the configuration and use of the Altos III terminal for use with the Niakwa Programming Language.

D.3.1 Screen Character Set

Downloadable Fonts

The Altos III terminal does not support the use of downloadable fonts.

Alternate Character Set (Pixel Graphics)

The Altos III does not fully support the alternate character set .

Non-English Character Sets

Byte 15 of $OPTIONS system variable has been implemented as a "font designator", whereby a specific type of character set may be selected. For Altos III terminals, the most practical choices are:

"0" - (the default) standard graphics.

"A" - this is used to support the UK (British) character set.

"B" - this is used to support the US (ASCII) character set.

Non-English users must choose between using the non-English character set and using pixel graphics. If byte 15 of $OPTIONS is set to "< ", then the non-English characters can be assigned to any hex code desired through the use of the Screen Translation Table. In both cases, the appropriate font file should be downloaded.

Where the required national replacement character set is different, it is possible to choose another appropriate character set through the setting of byte 15 of $OPTIONS to a designated code. Available codes include:

Character Set	**Font Designator**
ASCII	"B"
DEC special graphics	"0"
British	"A"

Character Translation

The default values for the Altos III terminal are contained in the disk file provided with the NPL Development Package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

D.3.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.3.3 Attributes

The Altos III terminal allows only one "hidden" attribute (does not occupy a character position). For the Altos III, the RunTime uses this single attribute for displaying any of the four supported attributes or any combination of attributes. The single attribute to be displayed can be modified by use of the SETUP option on the Altos III terminal. Available attributes are DIM (the default), REVERSE, and UNDERLINE.

NOTE: Testing has shown that the default attribute of DIM is least attractive in NPL application software. Try the different attribute choices with the software being used and determine which is the most applicable.

The Altos III does support multiple "non-hidden" attributes. However, these cannot be used by NPL since they occupy a character position on the screen. Refer to the Altos III User's Guide for further details on the use of attributes on the Altos III terminal.

D.3.4 Cursor Handling

The cursor on the Altos III terminal can be set to on or off under program control.

The type of cursor displayed (blinking or steady, block or underline) is not programmable. This makes it impossible to tell by examining the screen whether or not "EDIT" mode has been invoked.

Refer to the Altos III User's Guide for further details on cursor appearance.

D.3.5 Support for 132-Column Mode

The Altos III terminal supports a 132-column mode output. Refer to Section 7.4.12 for details on the implementation of this mode.

D.3.6 Keyboard Characteristics

The default values for the Altos III terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

Default equivalences for commonly used keys are:

HALT	(Definable by the host operating system). Refer to Chapter 6 of the appropriate NPL Supplement for details.
EXECUTE	LINE FEED
CANCEL	HOME
HELP	HELP

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST and WEST)
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.)
INS/LINE	Inserts a linefeed within a line of text.
DEL/LINE	Deletes characters from the current cursor position to the end of the line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/SCRN	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/SCRN	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer’s Guide for more information.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer’s Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

F5	Moves cursor to end of line being edited.
F6	Moves cursor down one line.
F7	Moves cursor up one line.
F8	Moves cursor to beginning of line being edited.
F9	Erases all text from current position to end of line.
F10	Deletes one character at the current cursor position.
F11	Inserts one space at the current cursor position.
F12	Moves cursor 5 spaces to the right.
F13	Moves cursor one space to the right.
F14	Moves cursor one space to the left.
F15	Moves cursor 5 spaces to the left.
F16	Recalls the current line.

Altos III Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**Altos III Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	LINE FEED
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
FF’A0’XX	UNDERSCORE (DEAD KEY)	?
’00 ... ’09	SF ’0 ... ’9	F1 ... F10
’0A ... ’0F	SF ’10 ... ’15	F11 ... F16
’10 ... ’19	SHIFT SF ’0 ... ’9	SHIFT F1 ... F10
’1A ... ’1F	SHIFT SF’10 ... ’15	SHIFT F11 ... F16
’42	PREV-SCREEN	SHIFT PREV/SCRN
’43	NEXT-SCREEN	NEXT/SCRN
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	DEL CHAR
’4A	INSERT	INS CHAR
’4C	EAST	EAST

Altos III Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**Altos III Key**
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-K
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	SHIFT SOUTH
’56	SHIFT-NORTH	SHIFT NORTH
’59	SHIFT-DEL (LINE DEL)	DEL LINE
’5A	SHIFT-INSERT (LINE INS)	INS LINE
’5C	SHIFT-EAST (EAST-5)	SHIFT EAST
’5D	SHIFT-WEST (WEST-5)	SHIFT WEST
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-Z
’7E	TAB	TAB
’7F	SHIFT-TAB	SHIFT TAB
’E1	HELP	HELP
’F0	EDIT	HOME

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.3.7 Local Printer Support

The Altos III terminal supports a local serial printer under NPL .

NOTE: Output directed to a non-existent or de-selected local serial printer will be lost.

D.3.8 Configuration Requirements

The Altos III terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
baud rate varies by operating system

If a local printer is in use, the AUX port of the terminal should be configured to match the settings of the printer.

Refer to Chapter 6 of the appropriate NPL Supplement for details on operating system configuration requirements.

The ALTOS III can be operated at up to 19200 baud, however, baud rates higher than 9600 may have restricted cable length requirements.

NOTE: The baud rate may also be limited by the operating system the terminal is being used on.

D.3.9 Use as a Remote Terminal

The Altos III terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: Two stop bits will not cause a problem; however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.

D.3.10 Use with Native Operating System Functions and Utilities

The Altos III will operate properly while executing the Niakwa RunTime and with native operating system functions and utilities.

D.4 Altos V

The following section details the configuration and use of the Altos V terminal for use with the Niakwa Programming Language.

D.4.1 Screen Character Set

Downloadable Fonts

The Altos V terminal supports the use of pixel graphics. The Altos V terminal has a "downloadable" character set contained in a disk file provided with the development package for each operating system. Refer to Chapter 6 of the appropriate NPL Supplement for details on the naming and location of these files as well as for specific commands necessary to download this character set. The discussion of this chapter assumes the character set has been downloaded. This font file should not be downloaded to any other type of terminal.

NOTE: There is a black raster line between pixel graphic characters on successive lines of the display.

The NPL utility program EDFONT can be used to modify the downloadable fonts. Refer to Chapter 13 for details on the Font Editor utility program.

Alternate Character Set (Pixel Graphics)

The Altos V terminal fully supports the alternate character set under NPL.

Non-English Character Set

Byte 15 of $OPTIONS system variable has been implemented as a "font designator", whereby a specific type of character set may be selected. For Altos V terminals, the most practical choices are:

"@" - (the default) this supports pixel graphics but no non-English characters.

"< " - (the DEC supplemental FONT) this generates all non-English characters (for all languages) in the range HEX(A1) to HEX(FE), instead of pixel graphics.

Non-English users must choose between using the non-English character set and using pixel graphics. If byte 15 of $OPTIONS is set to "< ", then the non-English characters can be assigned to any hex code desired through the use of the Screen Translation Table. In both cases, the appropriate font file should be downloaded.

Where the required national replacement character set is different, it is possible to choose another appropriate character set through the setting of byte 15 of $OPTIONS to a designated code. Available codes include:

<table>
<tr><th>Character Set</th><th>Font Designator</th></tr>
<tr><td>ASCII</td><td>"B"</td></tr>
<tr><td>DEC supplemental (VT200 only)</td><td>"< "</td></tr>
<tr><td>DEC special graphics</td><td>"0"</td></tr>
<tr><td>Downloaded Fonts</td><td>"@"</td></tr>
<tr><td>Alternate Graphics</td><td>"|"</td></tr>
<tr><td colspan="2">Only one of the following:</td></tr>
<tr><td>British</td><td>"A"</td></tr>
<tr><td>Dutch</td><td>"4"</td></tr>
<tr><td>Finnish</td><td>"C" or "5"</td></tr>
<tr><td>French</td><td>"R"</td></tr>
<tr><td>French Canadian</td><td>"Q"</td></tr>
<tr><td>German</td><td>"K"</td></tr>
<tr><td>Italian</td><td>"Y"</td></tr>
<tr><td>Norwegian/Danish</td><td>"E" or "6"</td></tr>
<tr><td>Spanish</td><td>"Z"</td></tr>
<tr><td>Swedish</td><td>"H" or "7"</td></tr>
<tr><td>Swiss</td><td>"= "</td></tr>
</table>

Character Translation

The default values for the Altos V terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

D.4.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.4.3 Attributes

The Altos V terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE).

Refer to the Altos V User's Guide for further details on the use of attributes on the Altos V terminal.

D.4.4 Cursor Handling

The cursor on the Altos V terminal can be set to on, off or blinking under program control.

The type of cursor displayed (block or underline) is left as a user option to be defined in SETUP.

Refer to the Altos V User's Guide for further details on cursor appearance.

D.4.5 Support for 132-Column Mode

The Altos V terminal supports 132-column mode output. Refer to Section 7.4.12 for details on the implementation of this mode.

D.4.6 Keyboard Characteristics

The default values for the Altos V terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the NPL Supplement for the name and location of the file on the operating system being used.

Default equivalences for commonly used keys are:

HALT	(Definable by the host operating system). Refer to Chapter 6 of the appropriate NPL Supplement for details.
EXECUTE	LINE FEED
CANCEL	HOME
HELP	HELP

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST).
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.
INS/LINE	Inserts a linefeed.
DEL/LINE	Deletes characters from the current cursor position to the end of the line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/SCRN	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/SCRN	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

F5	Moves cursor to end of line being edited
F6	Moves cursor down one line.
F7	Moves cursor up one line.
F8	Moves cursor to beginning of line being edited.
F9	Erases all text from current position to end of line.
F10	Deletes one character at the current cursor position.
F11	Inserts one space at the current cursor position.
F12	Moves cursor 5 spaces to the right.
F13	Moves cursor one space to the right.
F14	Moves cursor one space to the left.
F15	Moves cursor 5 spaces to the left.
F16	Recalls the current line.

Altos V Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**Altos V Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	LINE FEED
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
FF’A0’XX	UNDERSCORE (DEAD KEY)	?
’00 ... ’09	SF ’0 ... ’9	F1 ... F10
’0A ... ’0F	SF ’10 ... ’15	F11 ... F16
’10 ... ’19	SHIFT SF ’0 ... ’9	SHIFT F1 ... F10
’1A ... ’1F	SHIFT SF’10 ... ’15	SHIFT F11 ... F16
’42	PREV-SCREEN	SHIFT PREV/SCRN
’43	NEXT-SCREEN	NEXT/SCRN
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	DEL CHAR
’4A	INSERT	INS CHAR
’4C	EAST	EAST

Altos V Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**Altos V Key**
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-C
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	SHIFT SOUTH
’56	SHIFT-NORTH	SHIFT NORTH
’59	SHIFT-DEL (LINE DEL)	DEL LINE
’5A	SHIFT-INSERT (LINE INS)	INS LINE
’5C	SHIFT-EAST (EAST-5)	SHIFT EAST
’5D	SHIFT-WEST (WEST-5)	SHIFT WEST
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-Z
’7E	TAB	TAB
’7F	SHIFT-TAB	SHIFT TAB
’E1	HELP	HELP
’F0	EDIT	HOME

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.4.7 Local Printer Support

The AltosV terminal supports a local serial printer under NPL .

NOTE: Output directed to a non-existent or de-selected local serial printer will be lost.

D.4.8 Configuration Requirements

The Altos V terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
baud rate varies by operating system

NOTE: **When using the Altos V terminal, be sure to set it to 7 bit mode under the GENERAL 1 setup screen and to 8 bit transfer under the COMMUNICATIONS 1 setup screen. If the terminal is not configured with the above parameters, the numeric keypad and function keys will not work properly.**

If a local printer is in use the AUX port of the terminal should be configured to match the settings of the printer.

Refer to Chapter 6 of the appropriate NPL Supplement for details on operating system configuration requirements.

The Altos V can be operated at up to 38400 baud, however, baud rates higher than 9600 may have restricted cable length requirements.

NOTE: **The baud rate may also be limited by the operating system the terminal is being used on.**

D.4.9 Use as a Remote Terminal

The Altos V terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL. For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: **Two stop bits will not cause a problem, however, 1 stop bit is approximate 10% faster, which may be important over a (slow) modem.**

D.4.10 Use with Native Operating System Functions and Utilities

The Altos V will operate properly while executing the Niakwa RunTime and with native operating system functions and utilities.

D.5 Bull HDS 1

The Honeywell Bull HDS 1 terminal is used in Wyse 50 emulation mode, and operates as follows under NPL.

D.5.1 Screen Character Set

Downloadable Fonts

The HDS 1 terminal does not fully support the use of downloadable fonts.

Alternate Character Set (Pixel Graphics)

The HDS 1 terminal does not fully support the alternate character set.

Non-English Character Sets

Use of non-English character sets through the $OPTIONS Font Designator is not supported on the HDS 1 terminal.

Character Translation

The default values for the HDS 1 terminal are contained in the file SCREEN.wy50.

D.5.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide for details on the use and syntax of the $BOXTABLE system variable to generate character box graphics.

D.5.3 Attributes

The HDS 1 terminal allows only one "hidden" attribute (does not occupy a character position). For the HDS 1, the RunTime Program uses this single attribute for displaying any of the four supported attributes or any combination of attributes. The single attribute to be displayed can be modified by use of the SETUP option on the HDS 1 terminal. Available attributes are DIM (the default) and NORMAL.

NOTE: **Testing has shown that the default attribute of DIM is least attractive in the application software we have tested. Try the different attribute choices with your software and determine the most applicable.**

The HDS 1 does support multiple "non-hidden" attributes. However, these cannot be used by NPL since they occupy a character position on the screen.

Refer to the HDS 1 User's Guide for further details on the use of attributes on the HDS 1 terminal.

D.5.4 Cursor Handling

The cursor on the HDS 1 terminal can be set to on, off, or blink under program control.

The type of cursordisplay is controlled by the RunTime, and always sets the cursor type to line.

Refer to the HDS 1 User's Guide for further details on cursor appearance.

D.5.5 Support For 132-Column Mode

The HDS 1 terminal does not support 132-column mode output.

D.5.6 Keyboard Characteristics

With the HDS 1 running in Wyse 50 mode, the numeric keypad of the HDS 1 takes on the characteristics of the Wyse 50 editing keys. The values for the editing keys is displayed above the keypad.

Default equivalences for commonly used keys are:

HALT	CTRL/C (or other key as defined by stty intr command)
EXECUTE	HOME
CANCEL	BREAK/DEL
HELP	ESC,ESC

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST).
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line will be recalled.)
INS/LINE	Inserts a linefeed within a line of text.
DEL/LINE	Deletes characters from the current cursor position to the end of line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/PAGE	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/PAGE	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for details.

CTRL-N Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for details.

In Edit Mode, SF keys are assigned the following functions:

F5 Move cursor to end of line being edited.
F6 Move cursor down one line.
F7 Move cursor up one line.
F8 Move cursor to beginning of line being edited.
F9 Erase all text from current position to end of line.
F10 Delete one character at the current cursor position.
F11 Not available
F12 Not available
F13 Not available
F14 Not available
F15 Not available
F16 Not available

HDS 1 Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**HDS 1 in Wyse 50 mode Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERLINE	UNDERLINE
81	?	?
82	EXEC	HOME
84	?	?
A1	SHIFT EXEC	CTRL-X
E5	SHIFT ERASE	CTRL-6
FF'A0'xx	UNDERLINE (DEAD KEY)	?
'00 ... '09	SF '0 ... '9	F1 ... F10
'0A ... '0D	SF '10 ...'13	PF1 ... PF4
'0E ... '0F	SF '14 ...'15	CTRL-PF1 ... CTRL-PF2
'10 ... '19	SHIFT SF '0...'9	SHIFT F1 ... F10
'1A ... '1D	SHIFT SF'10...'13	SHIFT PF1 ..SHIFT PF4
'1E ... '1F	SHIFT SF'14...'15	CTL+ SHIFT PF1 ... CTL+ SHIFT PF2

HDS 1 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**HDS 1 in Wyse 50 mode Key**
’42	PREV	CTRL-3
’43	NEXT	CTRL-PERIOD
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-9
’49	DELETE	CTRL-8
’4A	INSERT	CTRL-7
’4C	EAST	EAST
’4D	WEST	BACKSPACE
’4F	RECALL	CTRL-R
’50	SHIFT CANCEL	
’52	SHIFT PREV	CTRL-P
’53	SHIFT NEXT	CTRL-N
’55	SHIFT SOUTH	?
’56	SHIFT NORTH	?
’59	SHIFT DEL (LINE DEL)	CTRL-5
’5A	SHIFT INS (LINE INS)	CTRL-4
’5C	SHIFT EAST (EAST-5)	?
’5D	SHIFT WEST (WEST-5)	?
’5F	D TAB	CTRL-T
’7C	GL	?
’7D	SHIFT GL	CTRL-Z
’7E	TAB	TAB
’7F	SHIFT TAB	SHIFT TAB
’E1	HELP	ESC,ESC
’F0	EDIT	?

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" dash is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.5.7 Configuration Requirements

The HDS 1 terminal must be set for 8 data bits. There are no special requirements for baud rate, stop bits or parity, but terminal setup must match the operating system configuration for the serial port. The XON/XOFF handshake must be used.

The default configuration for HDS 1 terminals in Wyse 50 mode is:

8 data bits
2 stop bits
no parity
9600 baud

The keypad on the HDS 1 must be set to "soft keys" mode. If the key pad is set to "numeric" the editing keys will not work.

The HDS 1 can be operated at up to 38400 baud. Baud rates higher than 9600 may have restricted cable length requirements. See the HDS 1 User's Guide for details on setting baud rates.

D.5.8 Use As Remote Terminals

The HDS 1 terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: **Two stop bits will not cause a problem; however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.**

D.5.9 Use With Native Operating System Functions And Utilities

The HDS 1 will operate properly while executing the Niakwa RumTime and with native operating system functions and utilities.

D.6 Bull HDS 3

The Honeywell Bull HDS 3 terminal is used in VT200 emulation mode with the Niakwa Programming Language.

D.6.1 Screen Character Set

Downloadable Fonts

The Bull HDS 3 terminal supports the use of pixel graphics. The Bull HDS 3 terminal has a "downloadable" character set contained in a disk file provided with the NPL Development Package for each operating system. Refer to Chapter 6 of the appropriate NPL Supplement for details on the naming and location of these files and for specific commands necessary to download this character set. The discussion of this chapter assumes the character set has been downloaded. This font file should not be downloaded to any other type of terminal.

Alternate Character Set (Pixel Graphics)

The Bull HDS 3 terminal fully supports the alternate character set under NPL.

Non-English Character Set

Byte 15 of the $OPTIONS system variable has been implemented as a "font designator", whereby a specific type of character set may be selected. For Bull HDS 3 terminals, the most practical choices are:

> "A" - (the default) this supports pixel graphics but no non-English characters.
>
> "< " - (the DEC supplemental FONT) this generates all non-English characters (for all languages) in the range HEX(A1) to HEX(FE) instead of pixel graphics.

Non-English users must choose between using the non-English character set and using pixel graphics. If byte 15 of $OPTIONS is set to "< ", then the non-English characters can be assigned to any hex code desired through the use of the Screen Translation Table. In both cases, the first should be downloaded. Where the required national replacement character set is different, it is possible to choose another appropriate character set through the setting of byte 15 of $OPTIONS to a designated code. Available codes include:

Character Set	Font Designator
ASCII	"B"
DEC supplemental (VT200 only)	"< "
DEC special graphics	"0"
Only one of the following:	
British	"A"
Dutch	"4"
Finnish	"C" or "5"
French	"R"
French Canadian	"Q"
German	"K"
Italian	"Y"
Norwegian/Danish	"E" or "6"
Spanish	"Z"
Swedish	"H" or "7"
Swiss	"= "

Character Translation

The default values for the Bull HDS 3 terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the NPL Supplement for the name and location of the file on the operating system being used.

D.6.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.6.3 Attributes

The Bull HDS 3 terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE).

Refer to the Bull HDS 3 terminal User's Guide for further details on the use of attributes on the Bull HDS 3 terminal.

D.6.4 Cursor Handling

The cursor on the Bull HDS 3 terminal can be turned on and off under program control. However, it is not possible to distinguish between a blinking and non-blinking cursor.

Refer to the Bull HDS 3 User's Guide for further details on cursor appearance.

D.6.5 Support for 132-Column Mode

The Bull HDS 3 terminal supports 132-column mode output. Refer to Section 7.4.12 of the Programmer's Guide for details on the implementation of this mode.

D.6.6 Keyboard Characteristics

The default values for keyboard translation on the Bull HDS 3 terminal are contained in a disk file provided with the development package. Refer to Chapter 6 of the NPL Supplement for the name and location of the file on the operating system being used.

Uppercase Bull HDS 3 function keys are "programmable", and do not normally generate any key codes when pressed. The NPL Development Package contains a disk file that allows for the downloading of uppercase key definitions. Please refer to the appropriate NPL Supplement for details on operating system specific commands necessary for the implementation of this file.

NOTE: This file should not be downloaded to any other type of terminal.

Default equivalences for commonly used keys are:

HALT	CTRL/C (Definable by the host operating system). Refer to Chapter 6 of the appropriate NPL Supplement for details.
EXECUTE	SELECT
CANCEL	FIND
HELP	HELP

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST)
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.
PF2	Inserts a linefeed within a line of text.
PF4	Deletes characters from the current cursor position to the end of the line.
INS HERE/P F1	Inserts a blank space within a line of text.
REMOVE/PF3	Deletes a character at the current cursor position.
PREV/SCRN	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/SCRN	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or causes cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

F10	Moves cursor to end of line being edited.
F11	Moves cursor down one line.
F12	Moves cursor up one line.
F13	Moves cursor to beginning of line being edited.
F14	Erases all text from current position to end of line.
F17	Deletes one character at the current cursor position.
F18	Inserts one space at the current cursor position.
F19	Moves cursor 5 spaces to the right.
F20	Moves cursor one spaces to the right.
CTRL/[,e	Moves cursor one space to the left.
CTRL/[,r	Moves 5 spaces to the left.

Bull HDS 3 Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**Bull HDS 3 Key**
08	BACKSPACE	CTRL-BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	SELECT
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
FF’A0’XX	UNDERSCORE (DEAD KEY)	?
’00 ... ’09	SF ’0 ... ’9	F6 ... F14, F17
’0A ... ’0C	SF ’10 ... ’12	F18 ... F20
’0D ... ’0F	SF ’13 ... ’15	CTRL-[, (e, r, t)
’10 ... ’19	SHIFT SF ’0 ... ’9	SHIFT F6 ... F14, F17
’1A...’1C	SHIFT SF’10...’12	SHIFT F18 ... F20
’1D...’1F	SHIFT SF’13...’15	CTRL-[,SHIFT, (E, R, T)
’42	PREV-SCREEN	PREV SCRN
’43	NEXT-SCREEN	NEXT SCRN
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	REMOVE / PF3
’4A	INSERT	INS HERE / PF1

Bull HDS 3 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**Bull HDS 3 Key**
’4C	EAST	EAST
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-K
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	CTRL-B
’56	SHIFT-NORTH	CTRL-U
’59	SHIFT-DEL (LINE DEL)	CTRL-Y / PF4
’5A	SHIFT-INSERT (LINE INS)	PF2
’5C	SHIFT-EAST (EAST-5)	CTRL-F
’5D	SHIFT-WEST (WEST-5)	CTRL-D
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-L
’7E	TAB	TAB
’7F	SHIFT-TAB	CTRL-[, TAB]
’E1	HELP	HELP
’F0	EDIT	FIND

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.6.7 Local Printer Support

The Bull HDS 3 terminal supports a local serial printer under NPL.

NOTE: Output directed to a non-existent or de-selected local serial terminal will be lost.

D.6.8 Configuration Requirements

The Bull HDS 3 series terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
baud rate varies by operating system

If a local printer is in use, the AUX port of the terminal should be configured to match the settings of the printer.

Refer to Chapter 6 of the appropriate NPL Supplement for details on operating system configuration requirements.

NOTE: **The Bull HDS 3 supports up to a 38400 baud rate, however, baud rates higher than 9600 may have restricted cable length requirements. The baud rate may also be limited by the operating system the terminal is being used on. Refer to the Bull HDS 3 User's Guide for details on setting baud rates.**

D.6.9 Use as a Remote Terminal

The Bull HDS 3 terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL. For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: **Two stop bits will not cause a problem, however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.**

D.6.10 Use with Native Operating System Functions and Utilities

The Bull HDS 3 operates properly while executing the Niakwa RunTime and with native operating system functions and utilities.

D.7 DEC VT100

The following section details the configuration and use of the DEC VT100 terminal for use with the Niakwa Programming Language.

D.7.1 Screen Character Set

Downloadable Fonts

The DEC VT100 terminal does not support the use of downloadable fonts.

Alternate Character Set (Pixel Graphics)

The DEC VT100 terminal does not support the Alternate Character set under NPL.

Non-English Character Set

Use of non-English character sets through the $OPTIONS Font Designator is supported on the DEC VT100 terminal, but alternate fonts are limited.

Character Translation

The default values for the DEC VT100 terminal are contained in a disk file provided with the development package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

D.7.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.7.3 Attributes

The DEC VT100 terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE).

Refer to the DEC VT100 terminal User's Guide for further details on the use of attributes on the DEC VT100 terminal.

D.7.4 Cursor Handling

Cursor control on the DEC VT100 is not programmable.

Refer to the DEC VT100 series User's Guide for further details on cursor appearance.

D.7.5 Support for 132-Column Mode

The DEC VT100 terminal supports 132-column mode output. Refer to Section 7.4.12 of the Programmer's Guide for details on the implementation of this mode.

D.7.6 Keyboard Characteristics

The default values for the DEC VT100 terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 f the NPL Supplement for the name and location of the file on the operating system being used.

Default equivalences for commonly used keys are:

HALT	(Definable by the host operating system). Refer to Chapter 6 of the NPL Supplement for details.
EXECUTE	LINE FEED
CANCEL	ESC, c (pressing ESC then lower case c)
HELP	ESC, h (pressing ESC then lower case h)

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST and WEST)
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.)
F2	Inserts a linefeed within a line of text.

F4 Deletes characters from the current cursor position to the end of line.

F1 Inserts a blank space within a line of text.

F3 Deletes a character at the current cursor position.

In Edit Mode, SF keys are assigned the following functions:

ESC, 5 Moves cursor to end of line being edited.
ESC, 6 Moves cursor down one line.
ESC, 7 Moves cursor up one line.
ESC, 8 Moves cursor to beginning of line being edited.
ESC, 9 Erases all text from current position to end of line.
ESC, p Deletes one character at the current cursor position.
ESC, q Inserts one space at the current cursor position.
ESC, w Moves cursor 5 spaces to the right.
ESC, e Moves cursor one space to the right.
ESC, r Moves cursor one space to the left.
ESC, t Moves cursor 5 spaces to the left.
ESC,) Recalls the current line.

DEC VT100 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**DEC VT100 Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	LINE FEED
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
FF’A0’XX	UNDERSCORE (DEAD KEY)	?
’00 ... ’09	SF ’0 ... ’9	ESC, (0 ... 9)
’0A ... ’0C	SF ’10 ... ’12	ESC, (p, q, w)
’0D ... ’0F	SF ’13 ... ’15	ESC, (e, r, t)
’10 ... ’19	SHIFT SF ’0 ... ’9	ESC, SHIFT, (0 ... 9)

DEC VT100 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**DEC VT100 Key**
’1A ... ’1C	SHIFT SF’10 ... ’12	ESC, (P, Q, W)
’1D ... ’1F	SHIFT SF’13 ... ’15	ESC, (E, R, T)
’42	PREV-SCREEN	ESC, s
’43	NEXT-SCREEN	ESC, n
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	F3
’4A	INSERT	F1
’4C	EAST	EAST
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-C
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	CTRL-B
’56	SHIFT-NORTH	CTRL-U
’59	SHIFT-DEL (LINE DEL)	F4
’5A	SHIFT-INSERT (LINE INS)	F2
’5C	SHIFT-EAST (EAST-5)	CTRL-F
’5D	SHIFT-WEST (WEST-5)	CTRL-D
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-L
’7E	TAB	TAB
’7F	SHIFT-TAB	ESC, TAB
’E1	HELP	ESC, h
’F0	EDIT	ESC, c

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" dash is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.7.7 Local Printer Support

The DEC VT100 terminal does not support a terminal printer. Some similar models, e.g., VT102 and emulator packages, do support a terminal printer. The RunTime allows for a VT100 terminal to be configured with a local printer. If a printer is not attached, output will be lost.

D.7.8 Configuration Requirements

The DEC VT100 terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
baud rate varies by operating system

If a local printer is in use, the AUX port of the terminal should be configured to match the settings of the printer.

Refer to Chapter 6 of the appropriate NPL Supplement for details on operating system configuration requirements.

The DEC VT100 can be operated at up to 9600 baud.

NOTE: The baud rate may also be limited by the operating system, the terminal is being used on. Refer to the DEC VT100 User's Guide for details on setting baud rates.

D.7.9 Use as a Remote Terminal

The DEC VT100 terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: Two stop bits will not cause a problem, however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.

D.7.10 Use with Native Operating System Functions and Utilities

The DEC VT100 will operate properly while executing the Niakwa RunTime and with native operating system functions and utilities.

D.8 DEC VT200 Series

The following section details the configuration and use of the DEC VT200 terminal for use with the Niakwa Programming Language.

D.8.1 Screen Character Set

Downloadable Fonts

The DEC VT200 series terminal supports the use of pixel graphics. The DEC VT200 series terminal has a "downloadable" character set contained in a disk file provided with the NPL Development Package for each operating system. Refer to Chapter 6 of the appropriate NPL Supplement for details on the naming and location of these files and for specific commands necessary to download this character set. The discussion of this chapter assumes the character set has been downloaded. This font file should not be downloaded to any other type of terminal.

Alternate Character Set (Pixel Graphics)

The DEC VT200 series terminal fully supports the alternate character set under NPL.

Non-English Character Set

Byte 15 of the $OPTIONS system variable has been implemented as a "font designator", whereby a specific type of character set may be selected. For VT200 series terminals, the most practical choices are:

"A" - (the default) this supports pixel graphics but no non-English characters.

"< " - (the DEC supplemental FONT) this generates all non-English characters (for all languages) in the range HEX(A1) to HEX(FE) instead of pixel graphics.

Non-English users must choose between using the non-English character set and using pixel graphics. If byte 15 of $OPTIONS is set to "< ", then the non-English characters can be assigned to any hex code desired through the use of the Screen Translation Table. In both cases, the first should be downloaded. Where the required national replacement character set is different, it is possible to choose another appropriate character set through the setting of byte 15 of $OPTIONS to a designated code. Available codes include:

Character Set	Font Designator
ASCII	"B"
DEC supplemental (VT200 only)	"< "
DEC special graphics	"0"
Only one of the following:	
British	"A"
Dutch	"4"
Finnish	"C" or "5"
French	"R"
French Canadian	"Q"
German	"K"
Italian	"Y"
Norwegian/Danish	"E" or "6"
Spanish	"Z"
Swedish	"H" or "7"
Swiss	"= "

Character Translation

The default values for the DEC VT200 series terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the NPL Supplement for the name and location of the file on the operating system being used.

D.8.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.8.3 Attributes

The DEC VT200 series terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE).

Refer to the DEC VT200 series terminal User's Guide for further details on the use of attributes on the DEC VT200 series terminal.

D.8.4 Cursor Handling

The cursor on the DEC VT200 series terminal can be turned on and off under program control. However, it is not possible to distinguish between a blinking and non-blinking cursor.

Refer to the DEC VT200 series User's Guide for further details on cursor appearance.

D.8.5 Support for 132-Column Mode

The DEC VT200 series terminal supports 132-column mode output. Refer to Section 7.4.12 of the Programmer's Guide for details on the implementation of this mode.

D.8.6 Keyboard Characteristics

The default values for keyboard translation on the DEC VT200 series terminal are contained in a disk file provided with the development package. Refer to Chapter 6 of the NPL Supplement for the name and location of the file on the operating system being used.

Uppercase VT200 series function keys are "programmable", and do not normally generate any key codes when pressed. The NPL Development Package contains a disk file that allows for the downloading of uppercase key definitions. Please refer to the appropriate NPL Supplement for details on operating system specific commands necessary for the implementation of this file.

NOTE: This file should not be downloaded to any other type of terminal.

Default equivalences for commonly used keys are:

HALT	(Definable by the host operating system). Refer to Chapter 6 of the appropriate NPL Supplement for details.
EXECUTE	SELECT
CANCEL	FIND
HELP	HELP

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST)
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.
F2	Inserts a linefeed within a line of text.
F4	Deletes characters from the current cursor position to the end of the line.
INSERT or F1	Inserts a blank space within a line of text.
DELETE or F3	Deletes a character at the current cursor position.
PREV/SCRN	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/SCRN	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or causes cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

CTRL-N Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

F10	Moves cursor to end of line being edited.
F11	Moves cursor down one line.
F12	Moves cursor up one line.
F13	Moves cursor to beginning of line being edited.
F14	Erases all text from current position to end of line.
F17	Deletes one character at the current cursor position.
F18	Inserts one space at the current cursor position.
F19	Moves cursor 5 spaces to the right.
F20	Moves cursor one spaces to the right.
CTRL/[,e	Moves cursor one space to the left.
CTRL/[,r	Moves 5 spaces to the left.

DEC VT200 Series Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**DEC VT200 Key**
08	BACKSPACE	CTRL-BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	SELECT
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
FF'A0'XX	UNDERSCORE (DEAD KEY)	?
'00 ... '09	SF '0 ... '9	F6 ... F14, F17
'0A ... '0C	SF '10 ... '12	F18 ... F20
'0D ... '0F	SF '13 ... '15	CTRL-[, (e, r, t)
'10 ... '19	SHIFT SF '0 ... '9	SHIFT F6 ... F14, F17
'1A...'1C	SHIFT SF'10...'12	SHIFT F18 ... F20
'1D...'1F	SHIFT SF'13...'15	CTRL-[,SHIFT, (E, R, T)
'42	PREV-SCREEN	PREV SCRN
'43	NEXT-SCREEN	NEXT SCRN
'45	SOUTH	SOUTH

DEC VT200 Series Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**DEC VT200 Key**
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	REMOVE / PF3
’4A	INSERT	INS HERE / PF1
’4C	EAST	EAST
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-K
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	CTRL-B
’56	SHIFT-NORTH	CTRL-U
’59	SHIFT-DEL (LINE DEL)	CTRL-Y / PF4
’5A	SHIFT-INSERT (LINE INS)	PF2
’5C	SHIFT-EAST (EAST-5)	CTRL-F
’5D	SHIFT-WEST (WEST-5)	CTRL-D
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-L
’7E	TAB	TAB
’7F	SHIFT-TAB	CTRL-[, TAB]
’E1	HELP	HELP
’F0	EDIT	FIND

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.8.7 Local Printer Support

The DEC VT200 series terminal supports a local serial printer under NPL.

NOTE: Output directed to a non-existent or de-selected local serial terminal will be lost.

D.8.8 Configuration Requirements

The DEC VT200 series terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
baud rate varies by operating system

If a local printer is in use, the AUX port of the terminal should be configured to match the settings of the printer.

Refer to Chapter 6 of the appropriate NPL Supplement for details on operating system configuration requirements.

NOTE: **The DEC VT200 series supports up to a 19200 baud rate, however, baud rates higher than 9600 may have restricted cable length requirements. The baud rate may also be limited by the operating system the terminal is being used on. Refer to the DEC VT200 User's Guide for details on setting baud rates.**

D.8.9 Use as a Remote Terminal

The DEC VT200 series terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL. For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: **Two stop bits will not cause a problem, however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.**

D.8.10 Use with Native Operating System Functions and Utilities

The DEC VT200 operates properly while executing the Niakwa RunTime and with native operating system functions and utilities.

D.9 IBM 3151

The following section details the configuring and use of the IBM 3151 terminal for use with the Niakwa Programming Language.

D.9.1 Screen Character Set

Downloadable Fonts

The IBM 3151 console terminal does not support downloadable fonts.

Alternate Character Set (Pixel Graphics)

The IBM 3151 terminal does not support use of the alternate character set.

Non-English Character Sets

Use of non-English character sets through the $OPTIONS Font Designator is not supported on the IBM 3151 terminal.

Character Translation

The default values for both screen and keyboard translation for the IBM 3151 terminal are not built into the Niakwa RunTime. The downloadable files must be present and are used by the Niakwa RunTime instead of the built-in defaults. Refer to the appropriate NPL Supplement for the use of these files.

D.9.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Niakwa Programming Language Statements Guide for details on the use and syntax of the $BOXTABLE system variable to generate character box graphics.

D.9.3 Attributes

The IBM 3151 terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE).

Refer to the IBM 3151 User's Guide for further details on the use of attributes on the IBM 3151 terminal.

D.9.4 Cursor Handling

The IBM 3151 cannot display a "steady" cursor. A steady cursor appears the same as a blinking cursor.

NOTE: There is no control available for the cursor's appearance on the IBM 3151 terminal.

D.9.5 Support for 132-Column Mode

The IBM 3151 terminal supports 132-column mode output. Refer to Section 7.4.12 for details on the implementation of this mode.

D.9.6 Keyboard Characteristics

The IBM 3151 terminal should be configured to use the appropriate NPL keyboard file. Refer to Chapter 6 of the appropriate NPL Supplement for details.

Default equivalences for commonly used keys are:

HALT	CTRL-C (configurable by the stty intr command)
EXECUTE	HOME
CANCEL	BACK TAB (under HOME key)
HELP	ESC, ESC

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST).
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.)
?	Inserts a linefeed within a line of text.
?	Deletes characters from the current cursor position to the end of line.

INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/PAGE	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/PAGE	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the NPL Programmer's Guide for more information.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the NPL Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

F5	Moves cursor to end of line being edited.
F6	Moves cursor down one line.
F7	Moves cursor up one line.
F8	Moves cursor to beginning of line being edited.
F9	Erases all text from current position to end of line.
F10	Deletes one character at the current cursor position.
F11	Insert one space at the current cursor position.
F12	Moves cursor 5 spaces to the right.
SHIFT-CTRL-F1	Moves cursor one space to the right.
SHIFT-CTRL-F2	Moves cursor one space to the left.
SHIFT-CTRL-F3	Moves cursor 5 spaces to the left.
SHIFT-CTRL-F4	Recalls the current line.

IBM 3151 Default Keyboard Equivalences Table		
NPL Code	**NPL Virtual Key**	**IBM 3151 KEY**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN ENTER
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXEC	HOME CTRL-J CTRL-RETURN
84	CONTINUE (LOAD)	?
FF’A0’xx KEY)	UNDERLINE(DEAD KEY)	?
’00 ... ’0B	SF ’0 ... ’11	F1 ... F12
’0C ... ’0F	SF ’12 ...’15	CTRL-SHIFT F1 ... F4
’10 ... ’1B	SHIFT-SF ’0...’11	SHIFT-F1 ... F12
’1C ... ’1F	SHIFT-SF’12...’15	CTRL-SHIFT F5 ... F8
’42	PREV-SCREEN	CLEAR
’43	NEXT-SCREEN	ERASE
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’49	DELETE	DELETE
’4A	INSERT	INSERT
’4C	EAST	EAST
’4D	WEST	WEST
’50	SHIFT-CANCEL	CTRL-HOME
’52	SHIFT-PREV-SCREEN	CTRL-CLEAR
’53	SHIFT-NEXT-SCREEN	CTRL-ERASE
’59	SHIFT-DEL (Line DEL)	CTRL-DELETE
’5A	SHIFT-INS (LINE INS)	CTRL-INSERT
’7E	TAB	TAB
’7F	SHIFT- TAB	SHIFT-TAB BACKTAB
’E1	HELP	ESC, ESC
’F0	EDIT (CANCEL)	BACK TAB

The column "IBM 3151 Key" represents default values under AIX.

A question mark (?) indicates that no key is assigned to the Niakwa Programming Language code. When a "-" (dash) is used in a key sequence it indicates that the keys should be pressed simultaneous. When a "," (comma) is used in a key sequence it indicates that the keys should be pressed in sequential order.

D.9.7 Local Printer Support

The IBM 3151 terminal supports a local serial printer under NPL. Output directed to a non-existent or de-selected local serial printer is lost.

D.9.8 Configuration Requirements

The IBM 3151 terminal should be set to:

Machine Mode	IBM 3151
Row and Column	25 x 80
Auto LF	OFF
Word Length	8
Turnaround Character	CR
Insert Character	SPACE

NOTE: The IBM 3151 terminal must be configured in the above manner for proper operation with the NPL.

If a local printer is in use the AUX port of the terminal should be configured to match the settings of the printer.

D.9.9 Use as Remote Terminals

The IBM 3151 terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

D.9.10 Use with Native Operating System Functions and Utilities

The IBM 3151 terminal is supported for use with NPL and for native operating system functions and utilities.

D.10 NCR 4970

The NCR 4970 terminal is used in VT200 emulation mode. The following section details the use of this terminal under NPL.

D.10.1 Screen Character Set

Downloadable Fonts

The NCR 4970 terminals have a "downloadable" character set. Refer to Chapter 6 of the appropriate NPL Supplement for details.

NOTE: This file should not be downloaded to any other type of terminal.

The NPL utility program EDFONT can be used to modify the downloadable fonts. Refer to Chapter 13 of the Programmer's Guide, Compiler Utilities, for details on the Font Editor utility program.

Alternate Character Set (Pixel Graphics)

The NCR 4970 terminal fully supports the Alternate Character set under NPL.

Non-English Character Set

Byte 15 of $OPTIONS system variable has been implemented as a "font designator", whereby a specific type of character set may be selected. For NCR 4970 terminals, the most practical choices are:

"A" (the default) this supports pixel graphics but no non-English characters.

"< " (the supplemental FONT) this generates all non-English characters (for all languages) in the range HEX(A1) to HEX(FE) instead of pixel graphics.

Non-English users must choose between using the non-English character set and using pixel graphics. If byte 15 of $OPTIONS is set to "< ", then the non-English characters can be assigned to any hex code desired through the use of the Screen Translation Table. In both cases, the appropriate file should be downloaded.

Where the required national replacement character set is different, it is possible to choose another appropriate character set through the setting of byte 15 of $OPTIONS to a designated code. Available codes include:

Character Set	Font Designator
ASCII	"B"
DEC supplemental (VT200 only)	"< "
DEC special graphics	"0"
Only one of the following:	
British	"A"
Dutch	"4"
Finnish	"C" or "5"
French	"R"
German	"K"
Italian	"Y"
Norwegian/Danish	"E" or "6"
Spanish	"Z"
Swedish	"H" or "7"
Swiss	"= "

Character Translation

The default values for the NCR 4970 terminal are contained in a downloadable file. Refer to Chapter 6 of the appropriate NPL Supplement for more details.

D.10.2 Box Graphics

"True" box graphics are not supported for this terminal.

Character box graphics are supported on this terminal. Refer to the Statements Guide for details on the use and syntax of the $BOXTABLE system variable to generate character box graphics.

D.10.3 Attributes

The NCR 4970 terminal supports all of the video attributes in any combination (BRIGHT,BLINK, UNDERLINE, REVERSE).

Refer to the NCR 4970 User's Guide for further details on the use of attributes on the NCR 4970 terminal.

D.10.4 Cursor Handling

The cursor control on the NCR 4970 can be set to on and off under program control. However, blink or steady is not programmable.

Refer to the NCR 4970 User's Guide for further details on cursor appearance.

D.10.5 Support For 132-Column Mode

The NCR 4970 terminal supports 132-column mode output. Refer to Section 7.4.12 of the Programmer's Guide for details.

D.10.6 Keyboard Characteristics

The uppercase NCR 4970 function keys are "programmable", and do not normally generate any key codes when pressed. A downloadable file is used for support of the NCR 4970 keyboard. Refer to Chapter 6 of the appropriate NPL Supplement for details.

NOTE: The downloadable file should not be downloaded to any other type of terminal.

Default equivalences for commonly used keys are:

HALT	CTRL/C (or key as defined by the stty intr command)
EXECUTE	SELECT
CANCEL	FIND
HELP	HELP

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST).
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.)
PF2	Inserts a linefeed within a line of text.

PF4	Deletes characters from the current cursor position to the end of line.
INS HERE/PF1	Inserts a blank space within a line of text.
REMOVE/PF3	Deletes a character at the current cursor position.
PREV/SCRN	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/SCRN	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or causes cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for details.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for details.

In Edit Mode, SF keys are assigned the following functions:

F10	Moves the cursor to end of line being edited.
F11	Moves the cursor down one line.
F12	Moves the cursor up one line.
F13	Moves the cursor to beginning of line being edited.
F14	Erases all text from current position to end of line.
F17	Deletes one character at the current cursor position.
F18	Inserts one space at the current cursor position.
F19	Moves the cursor 5 spaces to the right.
F20	Moves the cursor one space to the right.
CTRL-[,e	Moves the cursor one space to the left.
CTRL-[,r	Moves the cursor 5 spaces to the left.

NCR 4970 Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**NCR 4970 Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERLINE	UNDERLINE
81	?	?
82	EXEC	SELECT
84	?	?
A1	SHIFT EXEC	CTRL-X
E5	SHIFT ERASE	CTRL-W
FF'A0'xx	UNDERLINE(DEAD KEY)	?
'00 ... '09	SF '0 ... '9	F6 ... F14,F17
'0A ... '0C	SF '10 ...'12	F18 ... F20
'0D ... '0F	SF '13 ...'15	CTRL-[,(e,r,t)*
'10 ... '19	SHIFT SF '0...'9	SHIFT F6 ...F14,F17
'1A ... '1C	SHIFT SF'10...'12	SHIFT F18 ... F20
'1D ... '1F	SHIFT SF'13...'15	CTRL-[,SHIFT,(E,R,T)*
'42	PREV	PREV SCRN
'43	NEXT	NEXT SCRN
'45	SOUTH	SOUTH
'46	NORTH	NORTH
'48	ERASE	CTRL-E
'49	DELETE	REMOVE,PF3
'4A	INSERT	INS HERE,PF1
'4C	EAST	EAST
'4D	WEST	WEST
'4F	RECALL	CTRL-R
'50	SHIFT CANCEL	CTRL-K
'52	SHIFT PREV	CTRL-P
'53	SHIFT NEXT	CTRL-N
'55	SHIFT SOUTH	CTRL-B
'56	SHIFT NORTH	CTRL-U
'59	SHIFT DEL (LINE DEL)	PF4
'5A	SHIFT INS (LINE INS)	PF2
'5C	SHIFT EAST(EAST-5)	CTRL-F
'5D	SHIFT WEST (WEST-5)	CTRL-D

NCR 4970 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**NCR 4970 Key**
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT GL	?
’7E	TAB	TAB
’7F	SHIFT TAB	CTRL-[,TAB*
’E1	HELP	HELP
’F0	EDIT	FIND

A question mark (?) indicates that no key is assigned to the NPL code.

CTRL and [must be pressed simultaneously, then released before the other key sequences are pressed.

When a "-" (dash) is used in a key sequence it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence it indicates that the keys should be pressed in sequential order.

D.10.7 Local Printer Support

The NCR 4970 terminal supports a local serial printer under NPL.

NOTE: Output directed to a non-existent or de-selected local serial printer will be lost.

D.10.8 Configuration Requirements

The NCR 4970 terminal must be configured as follows:

GENERAL FEATURES

VT200 Mode, 7 Controls
User Defined Keys Unlocked
User Features Unlocked
Save User Defined Keys
Application Keypad
Normal Cursor Keys

ENHANCEMENT FEATURES MENU

F-key Priority: Coded Keys Prime
VT100 Keys: F1 to F20

Terminal setup must match the system configuration for the serial port.

The NCR 4970 terminals is should be set up as follows:

Baud rate optimal
1 stop bits
Even parity
XON/XOFF handshake

The NCR 4970 can be operated at up to 19200 baud. Baud rates higher than 9600 may have restricted cable length requirements. See the NCR 4970 User's Guide for details on setting baud rates.

D.10.9 Use As Remote Terminals

The NCR 4970 terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: Two stop bits will not cause a problem, however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.

D.10.10 Use With Native Operating System Functions And Utilities

The NCR 4970 is supported under NPL and well suited for native operating system functions and utilities.

D.11 Spectrix SPX 701

The Spectrix 701 terminal is supported in its native Spectrix mode by NPL. This provides functionality that is very similar to a Wang 2x36 terminal.

The NPL RunTime will automatically place the Spectrix 701 terminal into native mode at start up time and place it in VT100 mode when the RunTime is exited, provided the terminal is set up correctly for NPL use. Refer to the appropriate NPL Supplement for more information.

This feature allows the Spectrix 701 terminal to be configured as a VT100 terminal for use with host operating system functions or non-NPL applications (refer to the DEC VT200 discussion in Section D.7).

D.11.1 Screen Character Set

Downloadable Fonts

The Spectrix SPX701 terminals fully supports the standard and alternate character sets without "downloadable" fonts.

Alternate Character Set (Pixel Graphics)

The Spectrix SPX701 terminal fully supports the Alternate Character set under NPL.

Non-English Character Set

Use of non-English character sets through the $OPTIONS Font Designator is not supported on the Spectrix SPX701 terminal.

Character Translation

The default values for the Spectrix SPX701 terminal are contained in a downloadable file. Refer to the appropriate NPL Supplement for more details.

D.11.2 Box Graphics

"True" box graphics are supported for this terminal. Refer to the Statements Guide for details on the use and syntax of the $BOXTABLE system variable to generate "true" box graphics. Character box graphics are not supported on this terminal.

D.11.3 Attributes

The Spectrix SPX701 terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE).

NOTE: The BLINK attribute does not function properly unless the status line is enabled.

Refer to the Spectrix SPX701 User's Guide for further details on the use of attributes on the Spectrix SPX701 terminal.

D.11.4 Cursor Handling

The cursor on the Spectrix SPX701 can be set to on, off, or blink under program control.

The cursor appearance is controlled by byte 32 of $OPTIONS (00 for default, 01 for line and 02 for block).

Refer to the Spectrix SPX701 User's guide for further details on cursor appearance.

D.11.5 Support For 132-Column Mode

The Spectrix SPX701 terminal supports 132-column mode output. Refer to section 7.4.12 of the Release III Programmer's Guide for details.

D.11.6 Keyboard Characteristics

Default values for keyboard translation on the Spectrix SPX701 are contained in down-loadable files. Refer to the appropriate NPL Supplement for details.

Default equivalences for commonly used keys are:

HALT	Definable by operating system (see note below).
EXECUTE	EXEC
CANCEL	SHIFT-CANCEL
HELP	SHIFT-CONTROL-RESET

NOTE: Due to the Spectrix SPX701's handling of XON/XOFF handshaking, and the codes transmitted by the various function keys, the HALT key on the keyboard cannot be used for the HALT function. The only keys which can be used without affecting the operation of the other keys are "@", "‘", and "{". Of these three choices, we recommend that the "{" key be used as the HALT key. Information on setting the HALT key under the native operating environment is given in Chapter 6 of the appropriate NPL Supplement. The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST).
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line will be recalled.
INS/LINE	Inserts a linefeed within a line of text.
DEL/LINE	Deletes characters from the current cursor position to the end of line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/PAGE	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or causes cursor movement to the first line of text.
NEXT/PAGE	Causes the screen display to shift to the next screen (if more than 23 line of text exist) or causes cursor movement to the last line of text.

In Edit Mode, SF keys are assigned the following functions:

SF’4	Moves the cursor to end of line being edited.
SF’5	Moves the cursor down one line.
SF’6	Moves the cursor up one line.
SF’7	Moves the cursor to beginning of line being edited.
SF’8	Erases all text from current position to end of line.
SF’9	Deletes one character at the current cursor position.
SF’10	Inserts one space at the current cursor position.
SF’11	Moves the cursor 5 spaces to the right.
SF’12	Moves the cursor one space to the right.
SF’13	Moves the cursor one space to the left.
SF’14	Moves the cursor 5 spaces to the left.
SF’15	Recalls the current line.

Spectrix SPX701 Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**Spectrix SPX701 Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	?
81	CLEAR	CLEAR
82	EXECUTE	EXEC
84	CONTINUE	CONT
A1	LOAD	SHIFT EXEC
E5	SHIFT ERASE	SHIFT ERASE
FF’A0’xx	UNDERLINE(DEAD KEY)	?
’00 ... ’0F	SF ’0 ... ’15	SF ’0 ... ’15
’10 ... ’1F	SHIFT SF ’0...’15	SHIFT SF ’0...’15
’42	PREV SCREEN	PREV/SCRN
’43	NEXT SCREEN	NEXT/SCRN
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	ERASE
’49	DELETE	REMOVE
’4A	INSERT	INSERT
’4C	EAST	EAST
’4D	WEST	WEST

Spectrix SPX701 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**Spectrix SPX701 Key**
’4F	RECALL	RECALL
’50	CANCEL	SHIFT CANCEL
’52	SHIFT PREV SCREEN	SHIFT PREV/SCRN
’53	SHIFT NEXT SCREEN	SHIFT NEXT/SCRN
’55	SHIFT SOUTH	SHIFT SOUTH
’56	SHIFT NORTH	SHIFT NORTH
’59	SHIFT DELETE	SHIFT REMOVE
’5A	SHIFT INSERT	SHIFT INS (LINE INS)
’5C	SHIFT EAST	SHIFT EAST (EAST-5)
’5D	SHIFT WEST	SHIFT WEST (WEST-5)
’5F	D TAB	SHIFT RECALL
’7C	GL	GL
’7D	SHIFT GL	SHIFT GL
’7E	TAB	TAB
’7F	SHIFT TAB	SHIFT TAB
’E1	HELP	SHIFT CONTROL RESET
’F0	EDIT	CANCEL

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.11.7 Local Printer Support

The Spectrix SPX701 terminal supports a local serial or parallel printer under NPL.

NOTE: Output directed to a non-existent or de-selected local printer will be lost.

D.11.8 Configuration Requirements

The Spectrix SPX701 terminal is supported by NPL with byte 9 of $MACHINE set to HEX(0C) if the RunTime recognizes that a Spectrix SPX701 terminal is in use.

The Spectrix SPX701 terminal has several emulation modes, but for use with NPL it must be set to VT100.

Refer to the appropriate NPL Supplements for more details on using this terminal with NPL files.

The Spectrix SPX701 terminal must be configured as:

VT100 Mode
Status line enabled
DTR flow control disabled
RTS flow control disabled
XON/XOFF handshaking

The default configuration for Spectrix SPX701 terminals is:

8 data bits
1 stop bits
No parity
9600 baud

There are no special requirements for stop bits, or parity. Terminal setup must match the operating system configuration for the serial port. The XON/XOFF handshake must be used with the DTR and RTS flow control disabled.

The Spectrix SPX701 can be operated at up to 19200 baud. Baud rates higher than 9600 may have restricted cable length requirements. See the Spectrix SPX701 User's Guide for details on setting baud rates.

D.11.9 Use As Remote Terminals

The Spectrix SPX701 terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: Two stop bits will not cause a problem, however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.

D.11.10 Use With Native Operating System Functions And Utilities

The Spectrix SPX701 is supported under NPL for native operating system functions and utilities.

D.12 Wang 2110A

The following section details the configuration and use of the Wang 2110a terminal for use with the NPL.

D.12.1 Screen Character Set

Downloadable Fonts

The Wang 2110A terminal does not support downloadable fonts.

Alternate Character Set (Pixel Graphics)

The Wang 2110A terminal does not support the Alternate Character Set under NPL.

Non-English Character Set

All characters available on the Wang PC in remote mode (i.e., WISCII) are supported by NPL on this terminal.

Character Translation

The default values for the Wang 2110A terminal are contained in a disk file provided with the development package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

D.12.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.12.3 Attributes

The Wang 2110A terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE).

NOTE: BLINK and BRIGHT display the same--always between normal and bright mode.

D.12.4 Cursor Handling

The cursor on the Wang 2110A terminal can be set to on, off or blink under program control.

D.12.5 Support for 132-Column Mode

The Wang 2110A terminal supports 132-column mode output. Refer to Section 7.4.12 for details on the implementation of this mode.

D.12.6 Keyboard Characteristics

The default values for the Wang 2110A terminal are contained in a disk file provided with the development package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used. The default keyboard file includes additional key codes to match upper case function keys on the WANG 2210A which by default do not generate any key codes for NPL. To use these upper case function keys, the following values must be programmed into the "Keyboard Configuration" section of the terminal's built-in SETUP program.

SETUP Key Name	Program Value for NPL	**NPL Key**
PF21	^[[50~	SF'20
PF22	^[[51~	SF'21
PF23	^[[52~	SF '22
PF24	^[[53~	SF '23
PF25	ALREADY DEFINED	SF '24
PF26	^[[54~	SF '25
PF27	^[[55~	SF '26
PF28	^[[56~	SF '27
PF29	^[[57~	SF '28
PF30	^[[58~	SF '29
PF31	^[[59~	SF '30

NOTE: In the "Program value" column, the "^[" is the notation used for the ESCAPE control code (HEX(1B)), and the "~" is a HEX(7E).

Default equivalences for commonly used keys are:

HALT	(Definable by the host operating system). Refer to Chapter 6 of the NPL Supplement for details.
EXECUTE	EXEC
CANCEL	CANCEL
HELP	HELP

The following editing keys are available:

Keypad arrows	NORTH, SOUTH, EAST, and WEST
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

SF’5	Moves cursor to end of line being edited.
SF’6	Moves cursor down one line.
SF’7	Moves cursor up one line.
SF’8	Moves cursor to beginning of line being edited.
SF’10	Deletes one character at the current cursor position.
SF’11	Inserts one space at the current cursor position.
SF’12	Moves cursor 5 spaces to the right.
SF’13	Moves cursor one space to the right.
SF’14	Moves cursor one space to the left.
SF’15	Moves cursor 5 spaces to the left.
SF’16	Recalls the current line.

Wang 2110A Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**WANG 2110A Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	EXEC
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
FF’A0’XX	UNDERSCORE (DEAD KEY)	?
’00 ... ’09	SF ’0 ... ’9	F1 ... F10
’0A ... ’0F	SF ’10 ... ’15	F11 ... F16
’10 ... ’19	SHIFT SF ’0 ... ’9	SHIFT F1 ... F10
’1A ... ’1F	SHIFT SF’10 ... ’15	SHIFT F11 ... F16
’42	PREV-SCREEN	PREV
’43	NEXT-SCREEN	NEXT
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	DELETE
’4A	INSERT	INSERT
’4C	EAST	EAST
’4D	WEST	WEST

Wang 2110A Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**WANG 2110A Key**
'4F	RECALL	CTRL-R
'50	SHIFT-CANCEL	CTRL-K
'52	SHIFT-PREV-SCREEN	CTRL-P
'53	SHIFT-NEXT-SCREEN	CTRL-N
'55	SHIFT-SOUTH	CTRL-B
'56	SHIFT-NORTH	CTRL-U
'59	SHIFT-DEL (LINE DEL)	CTRL-Y
'5A	SHIFT-INS (LINE INS)	CTRL-O
'5C	SHIFT-EAST (EAST-5)	CTRL-F
'5D	SHIFT-WEST (WEST-5)	CTRL-D
'5F	D TAB	CTRL-T
'7C	GL	CTRL-G
'7D	SHIFT-GL	CTRL-L
'7E	TAB	TAB
'7F	SHIFT-TAB	CTRL-[, TAB
'E1	HELP	HELP
'F0	EDIT	Cancel

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.12.7 Local Printer Support

The Wang 2110A terminal supports a local parallel printer under NPL.

NOTE: On the 2110A, the terminal hangs if attempting to print to a non-existent or de-selected local printer.

For the 2110A terminal to use a local printer, the section titled "Host Access" under the Aux Port setup screen of the terminal must be set to "YES".

D.12.8 Configuration Requirements

The Wang 2110A terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
25 line display
baud rate varies by operating system

If a local printer is in use, the AUX port of the terminal should be configured to watch the settings of the printer. Refer to Chapter 6 of the appropriate NPL Supplement for details on operating system configuration requirements. The default setting for the "LOCK" key causes all keys to produce upper shifted keys. This can be changed by the build-in SETUP program of the Wang 2110A. The default is set to "SHIFT LOCK". Changing this to "CAPS LOCK" only produces upper case letters.

NOTE: **The Wang 2110A terminal supports up to a 19200 baud rate, however, baud rates higher than 9600 may have restricted cable length requirements. Baud rate may also be limited by the operating system the terminal is being used on. Refer to the Wang 2110A Terminal User's Guide for details on setting baud rates.**

D.12.9 Use as a Remote Terminal

The Wang 2110A terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit and no parity would be appropriate.

D.12.10 Use with Native Operating System Functions and Utilities

The Wang 2110a does not support native operating system functions or utilities.

D.13 Wang 2x36 DE/DW

The following section details the configuration and use of the Wang 2x36 terminals for use with the NPL.

D.13.1 Screen Character Set

Downloadable Fonts

The NPL character set is fully supported on the Wang 2x36 DE/DW terminal without the use of downloadable fonts.

Alternate Character Set (Pixel Graphics)

The Wang 2x36 DE/DW terminal fully supports the Alternate Character Set under NPL.

Non-English Character Set

All characters available on the Wang 2200 are supported by NPL on this terminal.

Character Translation

The default values for the Wang 2x36 DE/DW terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

D.13.2 Box Graphics

"True" box graphics are supported for this terminal. Character box graphics are not supported on this terminal. "True" box graphics are always generated even if character boxes are specified.

D.13.3 Attributes

The Wang 2x36 DE/DW terminal supports all of the video attributes in any combination (BRIGHT, BLANK, UNDERLINE, REVERSE).

NOTE: BLINK and BRIGHT BLINK display the same--always between normal and bright mode.

D.13.4 Cursor Handling

The cursor on the Wang 2x36 DE/DW terminal can be set to on, off or blink under program control.

D.13.5 Support for 132-Column Mode

The Wang 2x36 DE/DW terminal does not support 132-column mode output.

D.13.6 Keyboard Characteristics

The default values for the Wang 2x36 DE/DW terminal are contained in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

Default equivalences for commonly used keys are:

HALT	HALT (not modifiable by keyboard translation)
EXECUTE	RUN
CANCEL	EDIT
HELP	RESET

NOTE: Unlike on the Wang 2200, HALT does not perform single step operation. HALT simply invokes Immediate Mode upon completion of the currently executing instruction. To single step requires that a STEP instruction be executed from Immediate Mode or the HELP processor. RUN (EXECUTE) is then used to single step through the program.

The Wang 2.36 DE/DW keyboard must be in upper case only mode to use HELP (RESET) or HALT.

The following editing keys are available:

Keypad arrows	NORTH, SOUTH, EAST and WEST (available on 2x36 DW only).
RECALL	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.)
SHIFT INSERT	Inserts a linefeed within a line of text.
SHIFT DELETE	Delete characters from the current cursor position to the end of line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/SCRN	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text (available on 2x36 DW only).
NEXT/SCRN	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or causes cursor movement to the last line of text (available on 2x36 DW only).
SHIFT NEXT/SCREEN	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.
SHIFT PREV/SCREEN	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

SF'4	Moves cursor to end of line being edited.
SF'5	Moves cursor down one line.
SF'6	Moves cursor up one line.
SF'7	Moves cursor to beginning of line being edited.
SF'8	Erases all text from current position to end of line.
SF'9	Deletes one character at the current cursor position.
SF'10	Inserts one space at the current cursor position.
SF'11	Moves cursor 5 spaces to the right.
SF'12	Moves cursor one space to the right.
SF'13	Moves cursor one space to the left.
SF'14	Moves cursor 5 spaces to the left.
SF'15	Recalls the current line.

Wang 2x36 DE/DW Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**Wang 2x36DE Key**	**Wang 2x36DW Key**
08	BACKSPACE	BACKSPACE	BACKSPACE
0D	RETURN	RETURN	RETURN
5F	UNDERSCORE	?	?
81	CLEAR	CLEAR	?
82	EXECUTE	RUN	EXEC
84	CONTINUE	CONTINUE	?
A1	LOAD	LOAD	SHIFT EXEC
E5	SHIFT-ERASE	ERASE	ERASE
FF'A0'XX	UNDERSCORE (DEAD KEY)	?	?
'00 ... '09	SF '0 ... '9	SF '0 ... '9	SF '0 ... '9
'0A ... '0F	SF '10 ... '15	SF '10 ... '15	SF '10 ... '15
'10 ... '19	SHIFT SF '0 ... '9	SHIFT SF '0 ... '9	SHIFT SF '0 ... '9
'1A ... '1F	SHIFT SF'10 ... '15	SHIFT SF'10 ... '15	SHIFT SF'10 ... '15
'42	PREV-SCREEN	-	PREV/SCRN
'43	NEXT-SCREEN	-	NEXT/SCRN
'45	SOUTH	-	SOUTH
'46	NORTH	-	NORTH
'48	ERASE	-	ERASE

Wang 2x36 DE/DW Default Keyboard Equivalences Table			
NPL Code	NPL Virtual Key	**Wang 2x36DE Key**	**Wang 2x36DW Key**
’49	DELETE	-	DELETE
’4A	INSERT	-	INSERT
’4C	EAST	-	EAST
’4D	WEST	-	WEST
’4F	RECALL	-	RECALL
’50	SHIFT-CANCEL	-	SHIFT-CANCEL
’52	SHIFT-PREV-SCREEN	-	SHIFT PREV/SCRN
’53	SHIFT-NEXT-SCREEN	-	SHIFT NEXT/SCRN
’55	SHIFT-SOUTH	-	SHIFT SOUTH
’56	SHIFT-NORTH	-	SHIFT-NORTH
’59	SHIFT-DELETE	-	SHIFT DEL (LINE DEL)
’5A	SHIFT-INSERT	-	SHIFT INS (LINE INS)
’5C	SHIFT-EAST	-	SHIFT EAST (EAST-5)
’5D	SHIFT-WEST	-	SHIFT WEST (WEST-5)
’5F	D TAB	-	D TAB
’7C	GL	-	GL
’7D	SHIFT-GL	-	SHIFT GL
’7E	TAB	FN	TAB
’7F	SHIFT-TAB	SHIFT FN	SHIFT TAB
’E1	HELP	SHIFT RESET	SHIFT RESET
’F0	EDIT	EDIT	EDIT

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.13.7 Local Printer Support

The Wang 2x36 DE/DW terminal supports a local parallel printer under NPL.

NOTE: On the Wang 2x36 DE/DW, the terminal hangs if attempting to print to a non-existent or de-selected local printer.

D.13.8 Configuration Requirements

The Wang 2x36 DE/DW terminal should be set to:

8 data bits
1 stop bit
odd parity
XON/XOFF handshake
baud rate varies by operating system

Since the local printer port on the Wang 2x36 DE/DW is a parallel port, there are no configuration requirements.

NOTE: The Wang 2x36 DE/DW requires execution of a special program. This program is contained on a disk provided with the development package. This program sets up proper XON/XOFF codes for use with the Wang 2x36 DE/DW terminal. If this program is not executed, flow control problems will occur and the terminal will not be usable. Please refer to Chapter 6 of the appropriate NPL Supplement for more information on this file.

D.13.9 Use as a Remote Terminal

The Wang 2x36 DE/DW terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

D.13.10 Use with Native Operating System Functions and Utilities

The Wang 2x36 does not support native operating system functions or utilities.

D.14 Wyse 50

The Wyse 50 terminal is supported by the Niakwa Programming Language as discussed below.

D.14.1 Screen Character Set

Downloadable Fonts

The Wyse 50 terminal does not support the use of downloadable fonts.

Alternate Character Set (Pixel Graphics)

Use of the alternate character set is not well supported by the Wyse 50.

Non-English Character Sets

Use of non-English character sets through the $OPTIONS Font Designator is not supported on the Wyse 50 terminal.

Character Translation

The default values for the Wyse 50 terminal are continued in a disk file provided with the NPL Development Package. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of the file on the operating system being used.

D.14.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.14.3 Attributes

The Wyse 50 terminal allows only one "hidden" attribute (does not occupy a character position). For the Wyse 50, the RunTime Program uses this single attribute for displaying any of the four supported attributes or any combination of attributes. The single attribute to be displayed can be modified by use of the SETUP option on the Wyse 50 terminal. Available attributes are DIM (the default), REVERSE, and NORMAL.

NOTE: Testing has shown that the default attribute of DIM is the least attractive in the application software we have tested. Try the different attribute choices with the software being used and determine which is the most applicable.

The Wyse 50 does support multiple "non-hidden" attributes. However, these cannot be used by NPL since they occupy a character position on the screen. Refer to the Wyse 50 terminal User's Guide for further details on the use of attributes on the Wyse 50 terminal.

D.14.4 Cursor Handling

The cursor on the Wyse 50 terminal can be set to on, off or blink under program control.

The cursor appearance is controlled by the RunTime Program, and always set the cursor type to line.

Refer to the Wyse 50 User's Guide for further details on cursor appearance.

D.14.5 Support for 132-Column Mode

The Wyse 50 terminal supports 132-column mode output. Refer to Section 7.4.12 of the Programmer's Guide for details on the implementation of this mode.

D.14.6 Keyboard Characteristics

The default values for the Wyse 50 terminal are contained in a disk file provided with the development package. Refer to Chapter 6 of the NPL Supplement for the name and location of the file on the operating system being used.

Default equivalences for commonly used keys are:

HALT	(Definable by the host operating system). Refer to Chapter 6 of the NPL Supplement for details.
EXECUTE	SEND
CANCEL	DEL
HELP	INS/REPLACE

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST and WEST)
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line is recalled.)
INS/LINE	Inserts a line fee within a line of text.
DEL/LINE	Deletes characters from the current cursor position to the end of the line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/PAGE	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/PAGE	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

F5	Moves cursor to end of line being edited.
F6	Moves cursor down one line.
F7	Moves cursor up one line.
F8	Moves cursor to beginning of line being edited.
F9	Erases all text from current position to end of line.
F10	Deletes one character at the current cursor position.
F11	Inserts one space at the current cursor position.
F12	Moves cursor 5 spaces to the right.
F13	Moves cursor one space to the right.
F14	Moves cursor one space to the left.
F15	Moves cursor 5 spaces to the left.
F16	Recalls the current line.

Wyse 50 Default Keyboard Equivalences Table

NPL Code	NPL Virtual Key	**Wyse 50 Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	SEND
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
FF'A0'XX	UNDERSCORE (DEAD KEY)	?
'00 ... '09	SF '0 ... '9	F1 ... F10
'0A ... '0F	SF '10 ... '15	F11 ... F16
'10 ... '19	SHIFT SF '0 ... '9	SHIFT F1 ... F10
'1A ... '1F	SHIFT SF'10 ... '15	SHIFT F11 ... F16
'42	PREV-SCREEN	PREV
'43	NEXT-SCREEN	NEXT
'45	SOUTH	SOUTH
'46	NORTH	NORTH
'48	ERASE	CTRL-E
'49	DELETE	DELETE
'4A	INSERT	INSERT
'4C	EAST	EAST

Wyse 50 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**Wyse 50 Key**
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-K
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	SHIFT SOUTH
’56	SHIFT-NORTH	SHIFT NORTH
’59	SHIFT-DELETE (LINE DEL)	SHIFT DEL (LINE DEL)
’5A	SHIFT-INSERT (LINE INS)	SHIFT INS (LINE INS)
’5C	SHIFT-EAST (EAST-5)	SHIFT EAST
’5D	SHIFT-WEST (WEST-5)	SHIFT WEST
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-Z
’7E	TAB	TAB
’7F	SHIFT-TAB	SHIFT TAB
’E1	HELP	REPL
’F0	EDIT	DEL

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.14.7 Local Printer Support

The Wyse 50 terminal supports a local serial printer under NPL.

NOTE: Output directed to a non-existent or de-selected local serial printer will be lost.

D.14.8 Configuration Requirements

The Wyse 50 terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
baud rate varies by operating system

If a local printer is in use the AUX port of the terminal should be configured to match the settings of the printer.

Refer to Chapter 6 of the appropriate NPL Supplement for details on operating system configuration requirements.

The Wyse 50 can be operated at up to 38400 baud, however, baud rates higher than 9600 may have restricted cable length requirements.

NOTE: Baud rate may also be limited by the operating system the terminal is being used on.

D.14.9 Use as a Remote Terminal

The Wyse 50 terminal may be connected as a remote workstation on operating systems which support it. Use as a remote workstation does not affect the operation of NPL. For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. In this case, a setting of 8 data bits, 1 stop bit, and no parity would be appropriate.

NOTE: Two stop bits will not cause a problem, however, 1 stop bit is approximately 10% faster, which may be important over a (slow) modem.

D.14.10 Use with Native Operating System Functions and Utilities

The Wyse 50 will operate properly while executing the Niakwa RunTime and with native operating system functions and utilities.

D.15 Wyse 60, 150, and 160

The Wyse 60, 150 and 160 terminals are supported by the Niakwa Programming Language as discussed below.

D.15.1 Screen Character Set

Downloadable Fonts

The character set for the Wyse 60, 150, and 160 terminal supports the use of pixel graphics. For use with the NPL, the Wyse 60, 150, and 160 terminals require a "downloadable" character set. Refer to Chapter 6 of appropriate NPL Supplement for details on the naming and location of this file.

NOTE: This font file should not be downloaded to terminals other than the Wyse 60, 150, and 160.

The Niakwa utility program EDFONT can be used to modify the downloadable fonts. Refer to Chapter 13 for details on the Font Editor utility program.

Alternate Character Set (Pixel Graphics)

The Wyse 60, 150, and 160 terminals fully support the alternate character set under the NPL.

Non-English Character Set

Use of non-English character sets through the $OPTIONS Font Designator is not supported on the Wyse 60, 150, or 160 terminals.

Character Translation

The default values for the Wyse 60, 150, and 160 terminals are contained in a disk file. Refer to Chapter 6 in the appropriate NPL Supplement for the name and location of this file.

D.15.2 Box Graphics

"True" box graphics are not supported for the Wyse 60, 150, or 160 terminals. Character box graphics are supported on these terminals. Refer to the Statements Guide, $BOX-TABLE for details on the syntax to generate character box graphics.

D.15.3 Attributes

The Wyse 60, 150, and 160 terminals support all video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE), however support for the BRIGHT attribute is slightly different from other terminals. Refer to the Wyse 60, 150, or 160 User's Guide for further details on the use of attributes on the Wyse 60 terminal. The Wyse 60, 150, and 160 only support DIM and NORMAL attributes. The NPL BRIGHT attribute can be handled as:

NORMAL= DIM and BRIGHT= NORMAL (default)

or

NORMAL= NORMAL and BRIGHT= DIM

Byte 21 of $OPTIONS controls this feature (00 for default, 01 for alternate method).

D.15.4 Cursor Handling

The cursor on the Wyse 60, 150, and 160 terminals can be set to on, off, or blink under program control. This appearance is controlled by byte 32 of $OPTIONS (00 for default, 01 for line, and 02 for block). Refer to the Wyse 60, 150, or 160 User's Guide for further details on cursor appearance.

D.15.5 Support For 132 Column Mode

The Wyse 60, 150, and 160 terminals support 132 column mode output. Refer to Section 7.4.12 for details on the implementation of this mode.

D.15.6 Keyboard Characteristics

The default values for the Wyse 60, 150, and 160 terminals are contained in a disk file. Refer to Chapter 6 of the appropriate NPL Supplement for the name and location of this file.

NOTE: The Wyse 150 and 160 do not support application key mode like the Wyse 60. As such, the file WYKEYS.ON must be downloaded on both the Wyse 150 and 160 prior to starting the RunTime. To return the keyboard back to its standard mode, the file WYKEYS.OFF must be downloaded on both the Wyse 150 and 160 prior to returning to the operating system.

For detailed information on downloading files, refer to Chapter 6 of the appropriate NPL Supplement.

With the Wyse 150 and 160 terminal keyboards "reprogrammed" for use with the NPL, leaving the Niakwa RunTime (SHELL, END, etc.) does not change the keyboard mode back to native operating system mode until the WYKEYS.OFF file is downloaded. This is true even if the terminal is turned off. To avoid conflicts with other native operating system programs, make sure the WYKEYS.OFF file is automatically downloaded when users who are using other native operating system applications leave the Niakwa Run-Time.

The Wyse 60, 150, and 160 support several different keyboards. Niakwa provides sup-port for the ASCII and PC styles only.

Default equivalences for commonly used keys are:

HALT	CTRL-C (ASCII keyboard) CTRL-ALT (PC Style keyboard)
EXECUTE	SEND or HOME (ASCII keyboard) HOME (PC Style keyboard)
CANCEL	DEL (ASCII keyboard) END (PC Style keyboard)
HELP	INS/REPLACE (ASCII keyboard) ESC (PC Style keyboard)

The following editing keys are available:

Keypad arrows	(NORTH,SOUTH,EAST, and WEST)
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line will be executed.
INS/LINE	Inserts a linefeed within a line of text.
DEL/LINE	Deletes characters from the current cursor position to the end of the line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/PAGE	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
NEXT/PAGE	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to chapter 5 for more information.
CTRL-N	Recalls the next command from the "multi-command" keyboard buffer. Refer to chapter 5 for more information.

In Edit Mode, SF keys are assigned the following function:

F5	Moves cursor to end of line being edited.
F6	Moves cursor down one line.
F7	Moves cursor up one line.
F8	Moves cursor to beginning of line being edited.
F9	Erases all text from current position to end of line.
F10	Deletes one character at the current cursor position.
F11	Inserts one space at the current cursor position.
F12	Moves cursor 5 space to the right.
F13	Moves cursor one space to the right.
F14	Moves cursor one space to the left.
F15	Moves cursor 5 spaces to the left.
F16	Recalls the current line.

NOTE: Default PC Style Keyboard Equivalences Table for the PC style keyboard, the Wyse 60, 150, and 160 terminals' default keyboard equivalences tables is as shown below followed by the ASCII keyboard equivalences table.

Wyse 60,150,and 160 Default Keyboard Equivalences Table - PC Style Keyboard		
Niakwa Code	**Niakwa Virtual Key**	**PC Style Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERLINE	UNDERLINE
81	CLEAR	?
82	EXECUTE	HOME CTRL-J
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W
'00...'09	SF '0...'9	F1...F10
'0A...'0F	SF '10...'15	ESC-0...ESC-5
'10...'19	SHIFT SF '0...'9	SHIFT F1...F10
'1A...'1F	SHIFT SF '10...'15	ESC-SHIFT-0 ... ESC-SHIFT-5
'42	PREV-SCREEN	PG UP
'43	NEXT-SCREEN	PG DN

Wyse 60,150,and 160 Default Keyboard Equivalences Table - PC Style Keyboard		
Niakwa Code	**Niakwa Virtual Key**	**PC Style Key**
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	DELETE
’4A	INSERT	INSERT
’4C	EAST	EAST
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	SHIFT-END CTRL-K
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	SHIFT-SOUTH
’56	SHIFT-NORTH	SHIFT-NORTH
’59	SHIFT-DEL (LINE DEL)	SHIFT-DELETE
’5A	SHIFT-INS (LINE INS)	SHIFT-INSERT
’5C	SHIFT-EAST	SHIFT-EAST
’5D	SHIFT-WEST	SHIFT-WEST
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-Z
’7E	TAB	TAB
’7F	SHIFT-TAB	SHIFT-TAB
’E1	HELP	ESC,ESC
’F0	EDIT	END

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

Wyse 60,150,and 160 Default Keyboard Equivalences Table - ASCII Keyboard		
Niakwa Code	Niakwa Virtual Key	**Wyse 60 Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERLINE	UNDERLINE
81	CLEAR	?
82	EXECUTE	SEND
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W SHIFT CLR
FF’A0’xx	UNDERSCORE (DEAD KEY)	?
’00...’09	SF ’0...’9	F1...F10
’0A...’OF	SF ’10...15	F11...F16
’10...’19	SHIFT SF ’0...’9	SHIFT F1...F10
’1A...’1F	SHIFT SF ’10...’15	SHIFT F11...F16
’42	PREV-SCREEN	PREV
’43	NEXT-SCREEN	NEXT
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	DELETE
’4A	INSERT	INSERT
’4C	EAST	EAST
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-K
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	SHIFT SOUTH
’56	SHIFT-NORTH	SHIFT NORTH
’59	SHIFT-DEL (LINE DEL)	SHIFT DEL (LINE DEL)
’5A	SHIFT-INS (LINE INS)	SHIFT INS (LINE INS)
’5C	SHIFT-EAST (EAST-5)	SHIFT EAST
’5D	SHIFT-WEST (WEST-5)	SHIFT WEST
’5F	D TAB	CTRL-T

Wyse 60,150,and 160 Default Keyboard Equivalences Table - ASCII Keyboard		
Niakwa Code	Niakwa Virtual Key	**Wyse 60 Key**
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-Z
’7E	TAB	TAB
’7F	SHIFT-TAB	SHIFT TAB
’E1	HELP	REPL
’F0	EDIT	DEL

A question mark (?) indicates that no key is assigned to the NPL code. When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneously. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.15.7 Local Printer Support

The Wyse 60 terminal supports a local serial printer under NPL.

NOTE: **Output directed to a non-existent or de-selected local serial printer will be lost.**

The Wyse 150 supports a local parallel printer under NPL. The Wyse 160 terminal supports a local parallel or a serial printer under NPL.

WARNING--The Wyse 150 terminal is configured with a parallel port, not a serial port as the Wyse 60 is. This port is labeled Aux1 on the Wyse 150. Do not attempt to run a serial device from this port or damage may result.

NOTE: **The COM2 port of the Wyse 160 terminal is supported by Niakwa for local serial printing. To use the COM2 port, it is necessary to use the Wyse 160 Setup to modify the setting of the LOCAL PRINTER value, from parallel to serial. Refer to the Wyse 160 terminal Guide for details on setting this option.**

D.15.8 Configuration Requirements

The Wyse 60, 150, and 160 terminals should be set to:

Native personality
8 data bits
1 stop bit
No parity
XON/XOFF handshake
10 x 16 character cells
25 line display

If a local serial printer is in use on the Wyse 60 terminal, the AUX port of the terminal should be configured to match the settings of the serial printer.

NOTE: **This does not apply to the Wyse 150 terminal since only parallel printing is supported locally, but does apply to the Wyse 160 terminal if serial printing has been selected through the Wyse 160 Setup.**

The Wyse 60, 150, and 160 can all operate at up to 38400 baud, however, baud rates slighter than 9600 may be restricted by cable length requirements.

D.15.9 Use As Remote Terminal

The Wyse 60, 150, and 160 terminals may be connected as a remote workstation. Use as a remote workstation does not affect the operation of the NPL.

For remote operations, it may be desirable to configure the terminal for use with modems which support only 10 bit data. Here, a setting of 8 data bits, 1 stop bit , and no parity would be appropriate.

NOTE: **Two stop bits will not cause a problem, however 1 stop bit is approximately 10% faster, which may be important over a modem (slow baud rate).**

D.15.10 Use with Native Operating System Functions and Utilities

The Wyse 60, 150, and 160 will all operate properly while executing the Niakwa Run-Time and with native operating system functions and utilities.

D.16 Wyse 370

The NPL RunTime supports the Wyse 370 color terminal. This terminal fully supports the NPL color conventions as documented in Chapter 7 of the NPL Programmer's Guide. The following sections detail configuring this terminal for use with NPL.

D.16.1 Screen Character Set

Downloadable Fonts

The character set for the Wyse 370 terminal supports the use of pixel graphics. The Wyse 370 terminal has a "downloadable" character set contained on 2 files provided on the Terminal Support Files Diskette provided with the NPL Development Package. Both of these files must be downloaded to insure correct operation of the Wyse 370 under NPL. Refer to Chapter 6 of the appropriate NPL Supplement for more information.

The NPL utility program EDFONT can be used to modify the downloadable fonts. Refer to Chapter 13 for details on the Font Editor utility program.

Alternate Character Set (Pixel Graphics)

The Wyse 370 terminal fully supports the Alternate Character set under NPL. However, it should be noted that the pixel graphic characters on this terminal are represented slightly smaller than on other terminals. As a result of this, these characters do not join together as they would on other terminals. A small gap will be present between the bottom of one character and the top of the character below.

Non-English Character Set

Use of non-English character sets through the $OPTIONS Font Designator is supported on the Wyse 370 terminal, but alternate fonts are limited.

Byte 15 of $OPTIONS system variable has been implemented as a "font designator", whereby a specific type of character set may be selected. For Wyse 370 terminals, the most practical choices are:

Non-English users must choose between using the non-English character set and using pixel graphics. If byte 15 of $OPTIONS is set to "< ", then the non-English characters can be assigned to any hex code desired through the use of the Screen Translation Table. In both cases, the appropriate font file should be downloaded.

Where the required national replacement character set is different, it is possible to choose another appropriate character set through the setting of byte 15 of $OPTIONS to a designated code. Available codes include:

Character Set	Font Designator
ASCII	"A"
DEC supplemental	"< "
Downloaded Fonts	"@"
Only one of the following:	
British	"A"
Dutch	"4"
Finnish	"C" or "5"
French	"R"
French Canadian	"Q"
German	"K"
Italian	"Y"
Norwegian/Danish	"E" or "6"
Spanish	"Z"
Swedish	"H" or "7"
Swiss	"= "

Character Translation

The default values for the Wyse 370 terminal are contained in a file on the Terminal Support Files Diskette provided with the NPL Development Package. Refer to the appropriate NPL Supplement for more information.

D.16.2 Box Graphics

"True" box graphics are not supported for this terminal. Character box graphics are supported on this terminal. Refer to the Statements Guide, $BOXTABLE for details on the syntax to generate character box graphics.

D.16.3 Attributes

The Wyse 370 terminal supports all of the video attributes in any combination (BRIGHT, BLINK, UNDERLINE, REVERSE). Refer to the Wyse 370 User's Guide for further details on the use of attributes on the Wyse 370 terminal.

D.16.4 Cursor Handling

The cursor on the Wyse 370 terminal can be set to on or off under program control. The cursor appearance is controlled by byte 32 of $OPTIONS (00 for default, 01 for line and 02 for block).

See the Wyse 370 User's Guide for further details on cursor appearance.

D.16.5 Support For 132-Column Mode

The Wyse 370 terminal supports 132-column mode output. Refer to Section 7.4.12 of the Programmer's Guide for details on the implementation of this mode.

D.16.6 Keyboard Characteristics

The default values for the Wyse 370 terminal are contained in a file on the Terminal Support Files Diskette provided with the NPL Development Package.

Many of the Wyse 370 function keys are "programmable" and do not normally generate any key codes when pressed. The NPL Development Package contains a file on the Terminal Support diskette containing definitions to these keys. This file must be downloaded to insure correct operation of the Wyse 370 under NPL. Refer to the appropriate NPL Supplement for more details.

The Wyse 370 supports several different keyboards. The recommended keyboard is the Wyse 60 style, ASCII keyboard. Other keyboard styles may be used, but default keyboard translation values provided by Niakwa would require modification by the developer. Note that other keyboard styles have not been tested by Niakwa.

Default equivalences for commonly used keys are:

HALT	(Definable by the host operating system). Refer to the appropriate the NPL Supplement for details.
EXECUTE	SEND
CANCEL	DEL
HELP	INS/REPLACE

The following editing keys are available:

Keypad arrows	(NORTH, SOUTH, EAST, and WEST).
Right arrow	Recalls the previous command entry. If a line number is entered before pressing the right arrow key, that specific program line will be recalled.)
INS/LINE	Inserts a linefeed within a line of text.
DEL/LINE	Deletes characters from the current cursor position to the end of line.
INSERT	Inserts a blank space within a line of text.
DELETE	Deletes a character at the current cursor position.
PREV/PAGE	Causes the screen display to shift to the previous screen (if more than 23 lines of text exist) or cause cursor movement to the first line of text.
NEXT/PAGE	Causes the screen display to shift to the next screen (if more than 23 lines of text exist) or cause cursor movement to the last line of text.
CTRL-P	Recalls the previous command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

CTRL-N Recalls the next command from the "multi-command" keyboard buffer. Refer to Chapter 5 of the Programmer's Guide for more information.

In Edit Mode, SF keys are assigned the following functions:

F5 Move cursor to end of line being edited.
F6 Move cursor down one line.
F7 Move cursor up one line.
F8 Move cursor to beginning of line being edited.
F9 Erase all text from current position to end of line.
F10 Delete one character at the current cursor position.
F11 Insert one space at the current cursor position.
F12 Move cursor 5 spaces to the right.
F13 Move cursor one space to the right.
F14 Move cursor one space to the left.
F15 Move cursor 5 spaces to the left.
F16 Recall the current line.

Wyse 370 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**Wyse 370 Key**
08	BACKSPACE	BACKSPACE
0D	RETURN	RETURN
5F	UNDERSCORE	UNDERLINE
81	CLEAR	?
82	EXECUTE	SEND
84	CONTINUE (LOAD)	?
A1	LOAD	CTRL-X
E5	SHIFT-ERASE	CTRL-W, SHIFT CLR
FF'A0'	xxUNDERSCORE(DEAD KEY	?
'00 ... '09	SF '0 ... '9	F1 ... F10
'0A ... '0F	SF '10 ...'15	F11 ... F16
'00 ... '19	SHIFT SF '0...'9	SHIFT F1 ... F10
'1A ... '1F	SHIFT SF'10...'15	SHIFT F11 ... F16
'42	PREV-SCREEN	PREV
'43	NEXT-SCREEN	NEXT

Wyse 370 Default Keyboard Equivalences Table		
NPL Code	NPL Virtual Key	**Wyse 370 Key**
’45	SOUTH	SOUTH
’46	NORTH	NORTH
’48	ERASE	CTRL-E
’49	DELETE	DELETE
’4A	INSERT	INSERT
’4C	EAST	EAST
’4D	WEST	WEST
’4F	RECALL	CTRL-R
’50	SHIFT-CANCEL	CTRL-K
’52	SHIFT-PREV-SCREEN	CTRL-P
’53	SHIFT-NEXT-SCREEN	CTRL-N
’55	SHIFT-SOUTH	SHIFT-SOUTH
’56	SHIFT-NORTH	SHIFT-NORTH
’59	SHIFT-DEL (LINE DEL)	SHIFT DEL(LINE DEL)
’5A	SHIFT-INS (LINE INS)	SHIFT DEL(LINE INS)
’5A	SHIFT-INS (LINE INS)	SHIFT INS (LINE INS)
’5C	SHIFT-EAST (EAST-5)	SHIFT EAST
’5D	SHIFT-WEST (WEST-5)	SHIFT WEST
’5F	D TAB	CTRL-T
’7C	GL	CTRL-G
’7D	SHIFT-GL	CTRL-Z
’7E	TAB	TAB
’7F	SHIFT-TAB	SHIFT TAB
’E1	HELP	REPL
’F0	EDIT	DEL

A question mark (?) indicates that no key is assigned to the NPL code.

When a "-" (dash) is used in a key sequence, it indicates that the keys should be pressed simultaneous. When a "," (comma) is used in a key sequence, it indicates that the keys should be pressed in sequential order.

D.16.7 Local Printer Support

The Wyse 370 terminal supports a local serial printer under NPL.

NOTE: Output directed to a non-existent or de-selected local serial printer will be lost.

D.16.8 Configuration Requirements

The Wyse 370 terminal should be set to:

8 data bits
1 stop bit
no parity
XON/XOFF handshake
24 line display
baud rate will vary by operating system

This terminal should be defined as a VT100 to the host environment.

If a local printer is in use the AUX port of the terminal should be configured to match the settings of the printer.

Refer to Chapter 5 of the appropriate NPL Supplement for details on operating system configuration requirements.

The Wyse 370 terminal can be operated at up to 38400 baud. However, baud rates higher than 9600 may have restricted cable length requirements. Please note that baud rate may also be limited by the operating system the terminal is being used on.

D.16.9 Color Support

The Wyse 370 terminal supports the use of color when configured with a WYSE 370 personality

D.16.10 Use As Remote Terminals

The Wyse 370 terminal may be connected as a remote workstation. Use as a remote workstation does not affect the operation of NPL.

D.16.11 Use With Native Operating System Functions And Utilities

The Wyse 370 terminal is not supported or well suited for use with native operating system functions and utilities.

APPENDIX E

KEYBOARD KEYWORDS

If the keyboard translation table causes a key to return a code in the range HEX(80) through HEX(FF), the Wang 2200 equivalent to this key appears, except in the case of the following two exceptions:

HEX(82) - (RUN/EXECUTE)	Used when program stepping (STEP mode is on) or to continue a halted program (STEP mode is off).
HEX(E5) -	ERASE entire line.

Keyboard keywords are supported only during command mode (not LINPUT or INPUT), and only by the interpretive RunTime (RTI, not RTP). Keys may be defined according to individual programming preferences and needs. The following list summarizes the Wang 2200 atomization scheme.

NOTE: Due to compatibility limitations, the items with asterisks (*) after them are never available as keyboard keywords (RUN and HEX).

Refer to Appendix D for information on default keys used to generate these key codes on specific terminals.

KEYWORD ATOMIZATION SCHEME							
HEX	Keyword	HEX	Keyword	HEX	Keyword	HEX	Keyword
(80)	’LIST ’	(A0)	’PRINT ’	(C0)	’FN’	(E0)	’LS=’
(81)	’CLEAR’	(A1)	’LOAD ’	(C1)	’ABS(’	(E1)	’ALL’
(82)	’RUN ’*	(A2)	’REM ’	(C2)	’SQR(’	(E2)	’PACK’
(83)	’RENUMBER ’	(A3)	’RESTORE ’	(C3)	’COS(’	(E3)	’CLOSE’
(84)	’CONTINUE’	(A4)	’PLOT ’	(C4)	’EXP(’	(E4)	’INIT’
(85)	’SAVE ’	(A5)	’SELECT ’	(C5)	’INT(’	(E5)	’HEX’*
(86)	’LIMITS ’	(A6)	’COM ’	(C6)	’LOG(’	(E6)	’UNPACK’
(87)	’COPY ’	(A7)	’PRINTUSING ’(C7)	’SIN(’	(E7)	’BOOL’	
(88)	’KEYIN ’	(A8)	’MAT ’	(C8)	’SGN(’	(E8)	’ADD’
(89)	’DSKIP ’	(A9)	’REWIND ’	(C9)	’RND(’	(E9)	’ROTATE’
(8A)	’AND ’	(AA)	’SKIP ’	(CA)	’TAN(’	(EA)	’$’
(8B)	’OR ’	(AB)	’BACKSPACE ’	(CB)	’ARC’	(EB)	’ERROR’
(8C)	’XOR ’	(AC)	’SCRATCH ’	(CC)	’#PI’	(EC)	’ERR’
(8D)	’TEMP ’	(AD)	’MOVE ’	(CD)	’TAB(’	(ED)	’DAC’
(8E)	’DISK ’	(AE)	’CONVERT ’	(CE)	’DEFFN’	(EE)	’DSC’
(8F)	’TAPE ’	(AF)	’PLOT ’	(CF)	’TAN(’	(EF)	’SUB’
(90)	’TRACE ’	(B0)	’STEP ’	(D0)	’SIN(’	(F0)	’LINPUT ’
(91)	’LET ’	(B1)	’THEN ’	(D1)	’COS(’	(F1)	’VER(’
(92)	’FIX(’	(B2)	’TO ’	(D2)	’HEX(’	(F2)	’ELSE ’
(93)	’DIM ’	(B3)	’BEG ’	(D3)	’STR(’	(F3)	’SPACE’
(94)	’ON ’	(B4)	’OPEN ’	(D4)	’ATN(’	(F4)	’ROUND’
(95)	’STOP ’	(B5)	’CI ’	(D5)	’LEN(’	(F5)	’AT(’
(96)	’END ’	(B6)	’R’	(D6)	’RE’	(F6)	’HEXOF(’
(97)	’DATA ’	(B7)	’D’	(D7)	’#’	(F7)	’MAX(’
(98)	’READ ’	(B8)	’CO ’	(D8)	’%’	(F8)	’MIN(’
(99)	’INPUT ’	(B9)	’LGT(’	(D9)	’P’	(F9)	’MOD(’
(9A)	’GOSUB ’	(BA)	’OFF’	(DA)	’BT’	(FA)	’DATE’
(9B)	’RETURN ’	(BB)	’DBACKSPACE ’(DB)	’G’	(FB)	’TIME’	
(9C)	’GOTO ’	(BC)	’VERIFY ’	(DC)	’VAL(’		
(9D)	’NEXT ’	(BD)	’DA ’	(DD)	’NUM(’		
(9E)	’FOR ’	(BE)	’BA ’	(DE)	’BIN(’		
(9F)	’IF ’	(BF)	’DC ’	(DF)	’POS(’		

APPENDIX F

DECOMPILER COMPATIBILITY

F.1 Overview

NPL includes provisions to "decompile" previously compiled programs. This capability is present both in the interpretive RunTime (RTI) and the compiler (B2C). Using the interpretive version of the RunTime Program, RTI, all existing NPL compiled code executes directly without conversion of any kind, with full interpretive capabilities enabled. RTI decompiles program code "on the fly" as required.

As each individual program line is "requested" during a HALT/STEP, LIST, EDIT/RECALL or other debugging or program editing commands, it is decompiled and displayed on the screen. Whenever a specific program line is EDITed, it is immediately recompiled. In essence, the RunTime Program decompiles and recompiles each line in the program as it is required for the particular interpretive operation.

Chapter 14 of the NPL Programmer's Guide explains that the "2200" option of the LSTFORMAT compiler option generates 2200 atomized code, which can then be ported directly to a Wang 2200 for operation.

This procedure for decompiling object code into source code is not without certain unexpected results, however. Reconstruction of some specific statements and syntactic elements of a program produces functionally equivalent code, but somewhat different in appearance. This is an important consideration. The remainder of this Appendix discusses these known differences, along with specific examples.

F.2 Differences In Source Code Decompilation

The following are expeced differences in source code decomplilation.

Spacing of syntactic elements is not preserved:

```
                A = B + C - D * E
decompiles as   A=B+C-D*E
```

Some statements in 2200T format:

```
                BIN(A$)=1
decompiles as   STR(A$,,1)=BIN(1)

                OR(A$,B$)
decompiles as   A$=OR B$

                XOR(A$,B$)
decompiles as   A$=XOR B$

                AND(A$,B$)
decompiles as   A$=AND B$

                BOOL4(A$,B$)
decompiles as   A$=BOOL4 B$

                MAT CONVERT A() TO B$()
decompiles as   MAT MOVE A() TO B$()
```

Some structures of IF statements:

	IF X=1 THEN GOTO 1000
decompiles as	IF X=1 THEN 1000
	IF X=1 THEN IF Y=1 THEN 1000
decompiles as	IF X=1 AND Y=1 THEN 1000
	IF X=1 THEN 1000: ELSE X=2
decompiles as	IF X=1 THEN 1000: X=2

The TRANslate instruction:

	$TRAN(A$,B$)FF
decompiles as	$TRAN(A$,B$)

Some logical operations using BOOLx:

	A$=BOOLE B$
decompiles as	A$=OR B$
	A$=BOOL6 B$
decompiles as	A$=XOR B$
	A$=BOOL8 B$
decompiles as	A$=AND B$

Statements with optional commas in syntax:

	LINPUT "PRINT" A$
decompiles as	LINPUT "PRINT",A$
	$GIO #2(A$)
decompiles as	$GIO#2,(A$)

STR functions with a first argument of "1":

	STR(A$,1,10)
decompiles as	STR(A$,,10)

Some formats of numerics:

	00001
decompiles as	1
	0.01
decompiles as	.01
	100E-4
decompiles as	.0100

FOR/TO statement using STEP value of 1:

	FOR I=1 TO 10 STEP 1
decompiles as	FOR I=1 TO 10

Explicit images used for PACK/UNPACK containing editing characters and sign indicators:

	PACK($#,###.##-)A$ FROM A
decompiles as	PACK(+####.##) A$ FROM A

STR functions which have no numeric arguments:

	STR(A$,2)=STR(A$)
decompiles as	STR(A$,2)=A$

NOTE: This may cause problems if ported back to the 2200. Refer to the NPL Statements Guide, STR() function, for a discussion on the effects of string ravel.

MAT SEARCH statement using STEP value of 1:

	MAT SEARCH A$,=B$ TO C$ STEP 1
decompiles as	MAT SEARCH A$,=B$ TO C$

MAX/MIN functions generates combinations of simpler MAX/MIN functions:

	MAX(A,B(),C,D)
decompiles as	MAX(A,MAX(B()),C,D)

Math expressions with extraneous parentheses:

```
                A+(B*C)
decompiles as   A+B*C

                IF (A+B+C+D)>100 THEN 1000
decompile s as  IF A+B+C+D>100 THEN 1000
```

Use of an alpha-variable in ON ... GOTO/GOSUB/SELECT statements:

```
                ON A$ GOTO 10,20,30
decompiles as   ON VAL(A$) GOTO 10,20,30
```

Use of an alpha-variable in alpha-argument type expressions:

```
                $TRAN(A$<,L$>,B$)
decompiles as   $TRAN(A$<,VAL(L$,2)>,B$)
```

Syntax which allows for hex digits instead of a HEX() literal:

```
                POS(A$=HEX(1F))
decompiles as   POS(A$=1F)

                ALL(HEX(99))
decompiles as   ALL(99)

                INIT(HEX(01))Z$
decompiles as   INIT(01)Z$
```

Use of the RESTORE statement with "1" parameter:

```
                RESTORE 1
decompiles as   RESTORE

                RESTORE LINE 100,1
decompiles as   RESTORE LINE 100
```

Certain 2200T-syntax of DISK statements:

```
                COPY FR(0,X)
decompiles as   COPY #0,FR(0,X)

                DATASAVE DC OPEN T A$,B$
decompiles as   DATASAVE DC OPEN T (A$)B$

                SELECT #1310,PRINT /215,#0 D32
decompiles as   SELECT #1/310,PRINT 215,DISK/D32
```

DEFFN' literals separated by semicolons:

```
                DEFFN'10"Press ";"RETURN ";"key"
decompiles as   DEFFN'10"Press RETURN key"
```

DC disk type accesses which do not specify a file number:

```
                DATALOAD DC A$,B$
decompiles as   DATA LOAD DC #0,A$,B$

                LOAD T#0,"PROGRAM"
decompiles as   LOAD T "PROGRAM"
```

Optional DC keyword:

```
                LOAD DC T "PROGRAM"
decompiles as   LOAD T "PROGRAM"
```

Use of $FORMAT same as simple STR assignment:

```
                $FORMAT A$=A10,P+5.2
decompiles as   STR(A$,,4)=HEX(A00A5205)
```

ELSE clauses after ON GOTO are redundant:

```
                ON X GOTO 10,20,30: ELSE PRINT
decompiles as   ON X GOTO 10,20,30: PRINT
```

Use of the HEXOF statement:

```
                  PRINT HEXOF(A$,3,2)
decompiles as     PRINT HEXOF(STR(A$,3,2))
```

Use of STR function on MAT COPY and MAT SEARCH:

```
                  MAT COPY STR(A$,S,L) TO B$
decompiles as     MAT COPY A$<S,L >TO B$
```

NOTE: This could cause a problem if the string length equates to zero and the code is ported back to the 2200, since the first form would generate an error, but the second would not.

DIM/COM statements not indicating the default 16 byte string length become explicitly stated:

```
                  DIM A$,B$(2)
decompiles as     DIM A$16,B$(2)16
```

Default line number ranges in a LOAD statement:

```
                  LOAD T Z$ 0,9999 BEG 0
decompiles as     LOAD T Z$ 0

                  LOAD T Z$ ,9999 BEG 0
decompiles as     LOAD T Z$ 0
```

LOAD RUN default same as explicit specification:

```
                  LOAD RUN
decompiles as     LOAD RUN T "START"
```

SELECT addresses optionally preceded by a slash:

```
                  SELECT PRINT /215,DISK/D31,#1/310,LIST/215
decompiles as     SELECT PRINT 215,DISK/D31,#1/310,LIST 215
```

Use of the $BREAK ! statement:

```
                $BREAK !
decompiles as   $END
```

STOP statements with comments ending in a 4-digit line number which is the same as the current line number:

```
                1000 STOP "ERROR ON LINE 1000"
decompiles as   1000 STOP "ERROR ON LINE "#
```

SAVE protected options are treated the same:

```
                SAVE T P N$
decompiles as   SAVE T !N$
```

Uses of the STR(STR()) argument:

```
                B$=STR(STR(A$,2),,L)
decompiles as   B$=STR(A$,2,L)
```

The following examples indicate how certain code decompiles if the compiler options KEEPREMS and NUMBERS are set to OFF, or, under the Interpreter, $KEEPREMS=BIN(0) and $NUMBERS=BIN(0).

Implicit redimensioning of arrays by MAT statements becomes explicitly redimensioned (MAT ZER, MAT CON, MAT IDN, MAT READ and MAT INPUT):

```
                MAT REDIM A(10,10):MAT A=IDN
decompiles as   MAT A=IDN(10,10)
```

Certain statements with multiple arguments appear as single argument (PRINT, READ, COM, DIM, DATA, $CLOSE, SELECT, MAT REDIM, MAT INPUT, and MAT READ):

```
                COM A$10,B$10:COM C$10,D$10
decompiles as   COM A$10,B$10,C$10,D$10
```

Syntax variance of $SELECT statement as of revision 3.0:

```
                SELECT LIST
decompiles as   $SELECT (LIST)
```

NOTE: This only occurs if $KEEPREMS=BIN(0), where $NUMBERS may have any value.

F.3 Differences Between Rev. 1.03 and Later Revs.

The following examples indicate how certain code decompiles under the Interpreter with KEEPREMS=OFF and NUMBERS=OFF, after having been originally compiled under Revision 1.03.

Certain statements with multiple arguments appear as single argument (PRINT, READ, COM, DIM, DATA, $CLOSE, SELECT, NEXT, MAT REDIM, MAT INPUT, and MAT READ):

```
                PRINT A$;:PRINT B$;:PRINT C$
decompiles as   PRINT A$;B$;C$

                READ A$:READ B$:READ C$
decompiles as   READ A$,B$,C$
```

Compounded NEXT instructions from individual NEXTs:

```
                NEXT I: NEXT J
decompiles as   NEXT I,J
```

NOTE: This only occurs if $NUMBERS=BIN(0), where $KEEPREMS may have any value.

F.4 Summary

The importance of noting these decompilation differences is really a matter of programmer "style". The content and syntax of each line of program code function the same, but appear different in program listings and EDIT/RECALL mode, as compared to how the code was initially entered. In some cases a particular line of code may be easier to read when decompiled, while in other cases if may appear more lengthy in content.

APPENDIX G

"RAW" DEVICE COMPATIBILITY CHART

G.1 "Raw" Diskette Chart

This chart lists the "raw" diskette compatibility for all operating system groups currently supported by the NPL.

NPL "RAW" DEVICE COMPATIBILITY TABLE									
				5.25 Diskettes			3.5 Diskettes		
System	Drive Type	$FORMAT	320K	360K	720K	1.2MB	720K (8)	1.44MB (8)	2.88MB (9)
Wang 2200/CS	360K	X	X	X(1)	-	-	-	-	-
	1.2MB*	X(2)	-	-	-	-(3)	-	-	-
MS-DOS	360K	X	X	X(4)	-	-	-	-	-
	720K	X	-	-	-	-	X(4)	-	-
	1.2MB*	X	X	X(4)	-	X(4)	-	-	-
	1.44MB	X	-	-	-	-	X(4)	X(4)	-
	2.88MB	X	-	-	-	-	X(4)	X(4)	X(4)
Intel UNIX &Xenix	360K	(5)	-	X	X	-	-	-	-
	720K	(5)	-	-	-	-	X	-	-
	1.2MB*	(5)	-	X	X	X	-	-	-
	1.44MB	(5)	-	-	-	-	X	X	-
	2.88MB	(5)	-	-	-	-	X	X	X
SuperDOS	360K	X	X	X(4)	-	-	-	-	-
	720K	X(6)	-	-	-	-	X(4)	-	-
	1.2MB*	X	X	X(4)	-	X(4)	-	-	-
	1.44MB	X(6)	-	-	-	-	X(4)	X(4)	-
	2.88MB	X(6)	-	-	-	-	X(4)	X(4)	-
Honeywell XPS-100	720K	(7)	-	-(7)	X	-	-	-	-
Bull DPX/2	720K	(7)	-	-(7)	X	-	-	-	-
NCR TOWER	1.2MB	(10)	X(11)	X(11)	X(11)	-	-	-	-
IBM RS/6000	1.44MB	(5)	-	-	-	-	X	X	-

X SUPPORTED

* Any 360K diskette (DSDD) which has been written to in a 1.2MB drive may be unreliable when accessed on a 360K drive. This is a stated hardware limitation of the 1.2MB drive technology.

NOTE: Diskettes which have been written to on 1.2MB drives can always be read successfully on another 1.2MB drive. In addition, diskettes which have been created on a 360K drive can always be read successfully on any 1.2MB drive. This restriction applies only to reading diskettes on a 360K drive which have been written to on a 1.2MB drive.

NOTES:

(1) Using the "PC Interchange" format. A special $GIO microcommand sequence is required to format 360K diskettes. Refer to Section 15.7.1 of the Programmer's Guide for details.

(2) Neither 360K or 320K diskettes can be formatted in the 1.2MB diskette drive on the DS. 1.2MB diskettes created in the native 2200 format (256 byte sectors) on the Wang DS are not supported on any NPL machine.

(3) Although the Wang Documentation specifies that "PC Interchange" format is supported on the 1.2MB diskette, our testing to date has not yielded positive results.

(4) Access to 360K, 1.2MB, 720K, 1.4MB, and 2.88MB "RAW" diskettes under MS-DOS and SuperDOS (excluding 2.88MB) is supported by RTP and RTI, but not by B2C, which requires 320K "raw" formatted diskettes..

(5) $FORMAT DISK is not supported under Intel UNIX or Xenix. Refer to the Intel UNIX Supplement for details.

(6) $FORMAT DISK for 3.5 inch diskettes is not supported under Protected Mode SuperDOS.

(7) Many restrictions apply to the use of 360K "raw" diskettes on the Honeywell Bull XPS-100 and Bull DPX/2. Refer to Section 10.3 of the appropriate UNIX Addendum in the NPL Release III UNIX V Supplement for more information.

(8) Support of 3.5" 720K and 1.44MB diskettes require Release 3.00 or higher.

(9) Support of 3.5" 2.88MB diskettes require Release 4.00 or higher.

(10) The $FORMAT DISK command is not supported for any "raw" diskettes on the NCR TOWER 32.

(11) Only read and write operations of "raw" diskettes are supported. It is possible to format 640K diskettes using the UNIX format command. According to the NCR documentation, 720K diskettes are not supported. This is because the UNIX format command does not always succeed when a 720K diskette is specified. As a result, you may not be able to format 720K diskettes on the NCR TOWER 32 systems, It is not possible to format 320K or 360K diskettes. Therefore, 320K and 360K diskettes must be preformatted on another system.

For information regarding naming conventions and accessing "raw" diskettes within specific hardware environments, refer to Chapter 5 of the appropriate NPL Supplement.

GLOSSARY

The following terms are commonly used throughout the NPL documentation. Those terms displayed in all lower case refer to specific NPL statement option definitions as used in the NPL Statements Guide.

2227 Device Driver
A Niakwa-developed MS-DOS device driver which communicates with the serial ports to emulate the capabilities of the Wang 2227 Asynchronous Communications Controller.

address-var
An alpha-variable in which the first three bytes contain an ASCII representation of a hex-digit (0-9; A-F) which corresponds to a NPL device address must be specified.

Address-variable
A string variable used to store a three-byte NPL device address. The first three characters of the string are an ASCII representation of a hexadecimal digit.

Alpha (string)

Alpha ("alpha-numeric") is a general term applied to any identifier in an NPL program which is of a string type. For example, a string variable or a FUNCTION which returns a string. The following are examples of alpha identifiers:

```
employeeRec$ = 'readNextEmployeeRecord$()
name$(2) = employeeRec$.Name
firstName$ = STR(name$(2),1,10)
```

alpha-array

An alpha array variable must be specified.

Alpha Array

A variable that represents a one or two dimensional table used to hold strings.

alpha-expression

Any valid alpha expression may be used. This includes the use of literals as well as alpha functions. Refer to LET for further details on alpha expressions.

alpha-receiver

A series of alpha variables delimited by commas. At least one alpha variable must be specified. When multiple variables are specified, all are set to the value of the expression on the right side of the equivalence.

Alpha Receiver

Any alpha which may be assigned a string value. For example, an element of an alpha array or a field variable. Some examples of alpha receivers include:

```
name$
screen_colors$(1,5)
characters$(3)
books$.Title$
STR(buffers$,x,y)
(block$()
```

Alpha Scalar

Any string variable which is not an array.

alpha-variable

An alpha variable, either scalar or array, alpha array element, or STR function must be specified.

Application Software
A program or set of program tools which allow an end-user to perform a certain task. Some examples would be an accounting program or a word processor.

Array
A one or two dimensional table used to hold a set of elements of the same type. In NPL, arrays may be used to store either strings or numerics.

array-variable
A numeric or alpha array variable must be specified.

Array Variable
A variable that represents a table containing either strings or numerics.

Atomized Format
A Wang source code format that is used when storing Basic-2 programs in a format which must be read on a Wang 2200/CS minicomputer.

Autosize
A user-selectable option under the NPL MS-Windows RunTime. If a window is resized, the RunTime will attempt to pick a font size that will allow 24 lines of text to be displayed vertically.

Background Partition
For compatibility with the Wang 2200, the /B RunTime start-up option allows the RunTime to execute as a background task. No terminal input is allowed when a task is running in the background. The status of the output is maintained, but is not visible until the task is switched to the foreground.

Background Task
A process which runs without terminal I/O (user interaction).

BESDK
NPL (formerly "Basic-2C") External Subroutine Development Kit. This kit provides an interface which allows an NPL program to call external functions written in other languages such as C or Pascal.

Boot Program
In NPL, a boot program sets-up device specifications for NPL virtual devices and will load the starting program of an application.

Box Graphics

The use of graphics characters to enclose specific portions of a screen display. The graphics can be either "true" or "character" boxes. "True" boxes are drawn on a screen independent of text, allowing horizontal lines to be placed in between rows. "Character" boxes use the standard character set and, therefore, are limited. With "character" boxes, separating two rows of text requires an extra row of text to draw the line.

Browse Keys

A user-selectable option under the NPL MS-Windows RunTime. This option allows certain keyboard input to be accessed directly from the screen by using a mouse. When this option is active, a window is displayed on the screen with the names of the Niakwa virtual keys that can be accessed with a mouse.

Catalog

The area of a diskimage used for storing program and data files.

Catalog Index

An area in a diskimage which contains the directory information of every NPL program or data file stored.

Catalogued File/Program

A program or data file saved in a diskimage whose directory information is stored in the catalog index.

Common Variables

Variables which may be shared by multiple program overlays. Common variables are declared with the COM statement which allocates memory that is no cleared whenever new program overlays are loaded.

Compiler

A program that translates a source language into a different format. A C compiler, for example, translates a C source file into a machine-dependent object file which must be linked with other object file to for an executable file.

In NPL, the B2C compiler translates an NPL source file into machine-independent p-code file. The p-code file may then be executed by the RunTime interpreter (RTI) or by the RunTime program (RTP).

Compiling

Within the NPL environment, compiling is the process of translating an ASCII text file representing an NPL program into a machine-independent p-code file.

Constant

Any string or numeric data value in an NPL program which is not represented by a variable. For example:

```
"ABCD          1234          XYZ"
HEX(0D)        10 123.456    99.99e-5
```

Constant Array

A constant variable that is an array. Under the current release, constant arrays may not be initialized when declared therefore their only practical use is as function parameters.

Constant Scalar

A numeric or string variable which cannot be assigned a different value after it has been declared or initialized. Constant scalar identifier names are always preceded by the underscore character. For example:

```
DIM _MAX_N_CHARS = 10          return$1 = HEX(0D)
_specialChar$1 = "X"
```

Date/Time Stamp

Date and time information that is saved with a program or data in a NPL diskimage and that can be viewed later in a diskimage listing.

Declaration

A declaration is a statement which defines an identifier and associates it with a variable. It will also associate all relevant attributes of the variable. Here, the word variable is used in the loosest possible context. Declarations may be made for data variables, statement-labels, functions, procedures, marked DEFFN subroutines, field variables, record variables, named public sections or INCLUEd modules. The attributes defined by the declaration will vary, depending on the specific variable type. The following are examples of definitions:

```
DIM _MAXIMUM_RECORD_SIZE=4096
DIM X, Buffer$(_MAXIMUM_RECORD_SIZE)
COM Y
=Bail_Out
FUNCTION 'TaxRate(Gross_Income)
PROCEDURE 'StepByStep/PUBLIC
DEFFIN'100
FIELD Misc_Deduction1=_Standard_Numeric$
RECORD PayrollRecord
```

```
PUBLIC DustyOldFunctions
INCLUDE T "MYLIB"
USES DustyOldFunctions
```

Development Software

Software provided by Niakwa to a NPL developer. This includes the Niakwa compiler, terminal, keyboard, screen files, BESDK files, the ENABLED file, and the NPL Utilities.

DET

(see Device Equivalence Table)

device-address

A valid NPL device address in the format /xxx where x is a hex-digit must be specified. Refer to Chapter 7 of the NPL Programmer's Guide for details on NPL device addresses.

Device Address

A physical file or device in the native operating system is accessed indirectly through a logical device address. The address is specified as an ASCII representation of a unique three digit hexadecimal number.

Device Driver

A program used under the active operating system that communicates with specific hardware devices such as a serial port or video adapter.

Device Equivalence Table (DET)

A look-up table used by the NPL RunTime to link a logical device address to a physical file or device managed by the native operating system.

Device Sharing

In a multi-user environment, devices such as a printer may be accessed by two or more users at the same time. Typically, an application will require exclusive access to the device during the course of an operation; therefore, some form of lock/unlock mechanism is used. When a device is locked, its access is limited to the application which issued the lock. Until that application releases control of the device by issuing an unlock, no one else may access it. NPL supports the lock/unlock mechanism through the $OPEN and $CLOSE statements.

Disk Catalog Index

An area in a diskimage which contains directory information of every NPL program or data file stored.

diskimage
Refers to a NPL diskimage file or a "raw" disk device. Refer to Chapter 7 of the NPL Programmer's Guide for details on diskimage files and "raw" devices.

Diskimage
A logical disk device used to store NPL program and data files. Physically it is a normal native operating system file.

DLL
(see Dynamic Link Library)

DOS-Extender
System software which provides a layer between protected and real modes under MS-DOS. Under a DOS-extender environment, an application may be developed to run in protected mode with its larger address space/memory and still have access to the ROM BIOS or device drivers.

DT
(see Internal Device Table)

Dynamic Link Library (DLL)
An external library of dynamically linked procedures and functions used under MS-Windows. A dynamically linked module has its external references resolved at load or run time. This is different from a library whose modules have their external references resolved by a separate linking phase.

ENABLED File
A special file used with the development package which allows use of the NPL Run-Time interpreter (RTI) at end-user sites.

Environment
A set of variables which define conditions under which system commands, system software, or applications should execute.

System software, such as the 386/DOS-Extender or MS-Windows extender which can sit on top of a native operating system and execute other programs such that the original system is functionally enhanced.

Environment Variable
A variable used to hold information to be used by the operating system or applications. For example, the variable NIAKWA_RUNTIME describes the directory where the NPL RunTime should look for NPL system files.

Expanded Memory
Physical memory above one megabyte that is not addressable on 80x86 class computers (except in protected mode or by an XMS driver).

Extended Memory
Physical memory above one megabyte that is addressable on 80x86 class computers.

Field Identifier
A special variable type which allows access to a specific field in a record. Field identifiers allow modification of an area in a string buffer by a meaningful name. Field identifiers are always used with a string buffer. The following are examples of field identifiers used with a string variable named buffer$:

buffer$.fileNumber
buffer$.names$
buffer$.table$(5)

Field Level Reset
If an installed Gold Key Security is lost, Niakwa can provide a developer with a reset number which may be used with a special program to (RESET) allow the Gold Key security to be reinstalled (MS-DOS based operating environments only).

file-number
A valid NPL file number in the format #X where 0 X 15 must be specified. Refer to Chapter 7 of the NPL Programmer's Guide for details on file numbers.

File Number
A number which maps to an explicit logical device address stored in the device equivalence table. The number provides a second level of redirection to a native operating system file or I/O device.

Follow
A selectable user option under the NPL MS-Windows RunTime, which allows the RunTime to always keep the cursor in the visible portion of a window.

Foreground Task
A process which can communicate with the operator by means of a terminal display and keyboard. Most NPL applications run as a foreground task. On platforms that support only non-windowed multi-tasking, only one task assigned to a terminal may be the foreground task (task switchers may allow more than one foreground task as well).

FUNCTION
A self-contained, user defined subroutine that may return a string or numeric value. Arguments may be passed by value or by reference to a function. FUNCTIONs may have their own set of private local variables and statement labels.

Function (Private)
A FUNCTION or PROCEDURE has its own set of local variables with scope limited to the FUNCTION or PROCEDURE body. A variable declared in the mainline and a variable of the same name and type declared in a FUNCTION are independent of each other.

Global
(see Global Partition)

Global Partition
A section of memory on a Wang 2200 that is reserved for global variables and functions. These variables or functions can be accessed by all users. NPL does not support global partitions.

Gold Key
Disk-based security which uses a numeric fingerprint to prevent an NPL RunTime from being illegally copied .

Gold Key Number
An encrypted number based on the serial number of the Gold Key.

Handle Table
An internal table used by the RunTime to manage memory.

HELP Processor
A sub-program of the NPL RunTime which allows an end-user to access help information and perform other useful functions.

HELP Screen
The main screen of the Help Processor showing the options that have been made available by the application program.

hex-digit
A hexadecimal digit (0-9; A-F) must be specified.

Hexadecimal (HEX) Digit
A digit in base 16. Hexadecimal digits use the numbers 0-9 to represent 0-9 and letters A-F to represent 10-15 in base 10.

High Memory

The area of MS-DOS memory between conventional (640K) and 1024K (1MB) reserved for use by system hardware such as the ROM, video adapters, etc.

High Memory Area

The first 64K of extended memory (1024K-1088K) as defined by XMS. It is the only area of extended memory that is accessible to MS-DOS programs (HMA).

High RAM

Areas of high memory into which RAM has been mapped. TSRs and device drivers may be loaded here so that use of conventional memory is minimized. Also known as Upper Memory Blocks (UMBs).

Hogging

The same as file locking. In a multi-user environment, locking will prevent that file or device from being accessed by another user until it is unlocked. NPL supports the lock/unlock mechanism through the $OPEN and $CLOSE statements.

identifier

Any legal (long) identifier. Up to 255 alphanumeric characters (A-Z, a-z, 0-9, or _), starting with a letter (digits, _ not permitted as first character).

Identifier

A name that is used to identify a variable, PROCEDURE, FUNCTION, or statement label in an NPL program. Identifiers must consist of the characters: (A-Z, a-z, 0-9, and _) with the first character being an alpha identifier. Identifiers have a maximum length of 255 characters. For example:

A	B4	XYZ123
_MAX_TABLE_ENTRIES	stringTable$(2)	PI_div_by_2
OpenLevel2File()	=Proc_Exit	

Incremental Compiler

A program which compiles source code a small amount at a time as the code is being entered. The RunTime interpreter (RTI), is an incremental compiler which immediately converts NPL statements into p-code as each line is entered.

Immediate Mode

An interactive editing and debugging mode of the NPL RunTime interpreter. When entering a command or program line is entered, it is immediately parsed, compiled to p-code and, if applicable, executed.

Installation Guide
Documentation provided with the RunTime package describing the contents, configuration, and RunTime installation.

Internal Device Table (DT)
A look-up table which links a set of file numbers and default devices to explicit logical device addresses stored in the device equivalence table. This table provides a second level of redirection to a native operating system file or I/O device.

Interpreter
A program which will execute source code without going through a complete compile/link phase.

Interpretive Program
A program which is an interpreter. RTI is an interpretive program which can incrementally compile a NPL program to p-code and then execute it.

Invoke
The "!" option of $SHELL used to start a native operating system process.

IQ
An ad-hoc query application program from Intelligent Query (formerly Programmed Intelligence). This application allows the user to access data from a file in a variety of user-defined formats. Allows users of Niakwa's NDM applications to easily create custom reports.

ISAM Files
Indexed Sequential Access Method, a database storage method.

Keyboard Character Set
Different mapped keyboards that allow NPL to maintain Wang 2200 compatibility.

Keyboard Logging
The process of recording keystrokes and saving them into an ASCII file. The file can be played back by using the NPL $DEMO command.

Library
A library is a diskimage file containing one or more modules. Each module may contain a number of PROCEDUREs and FUNCTIONs that can be referenced by another program.

Line Remark
Line Remarks are specified by a semi-colon (";") as the first character in the statement. Line remarks are terminated only by a soft carriage return or by the end of the program line. Line Remarks are not terminated by a colon.

line-number
A valid NPL program line number must be specified. Valid values are in the range 0 to 32K.

Line Number
A number which is used to identify a line in a program source file.

Linking
When creating an executable from a main object module, a linker will search through specified object and system/user library files to resolve all external references. If a reference is found in a library module, then the linker will extract the required object code. The linker then completes address calculations for all references creating a single executable.

Literal
(see Literal String)

literal-string
An alpha literal string must be specified. This may be either a string (enclosed in quotes) or a hex literal.

Literal String
Any string value in an NPL program which is not a variable. For example, "any old string" and HEX(05) are considered string literals.

Local Printer
A printer that may be connected off the back of a terminal, usually on a multi-user terminal system.

LIN
(see Long Identifier Name)

Logical Construct
Any structured construct which makes use of a conditional expression. The WHILE/WEND and REPEAT/UNTIL statements are examples of logical constructs.

logical-expression
A conditional expression which evaluates to either true or false, used when making decisions. Refer to the IF/THEN statement for further details on logical expressions.

Logical Expression
A conditional expression which evaluates to a boolean value of true or false.

Long Identifier Name
A name that is associated with a variable, FUNCTION, PROCEDURE, or statement label. All identifiers are long identifiers except short variable names. For example, a function or statement label called x is a long identifier, but the variable X12 is not.

Mainline
The highest level subroutine in a the code hierarchy of an application.

Memory Manager
During memory allocation/de-allocation, the total free space eventually becomes fragmented into small, scattered, uneven sized chunks. The more fragmented available memory becomes the harder it is to allocate large blocks. A memory manager collects and moves free chunks into larger contiguous blocks.

Modules
A group of FUNCTIONs and PROCEDUREs contained in one file that can be used by other NPL programs with the use of the NPL INCLUDE statement. The variables, FUNCTIONs, and PROCEDURES can be either private or public.

Native File System
The organization of files and directories managed by the native operating system. Files stored in a diskimage are not part of the native file system.

Native Operating System
The set of programs that manage system resources on a particular platform.

NDM
The Niakwa Data Manager. This provides NPL applications with fully portable ISAM capabilities through the use of an external library.

NIAKSER.DAT Files
A file used by the RunTime for security and configuration information.

Niakwa Compiler
B2C, a program that provides a mechanism for converting BASIC-2 programs from one form to another in batch mode.

Niakwa Software
System software developed, distributed and supported by Niakwa.

Non-Catalogued Disk Space
The disk platters used with the Wang 2200 has a variable-sized catalogue area used to store program and data files with some space used for an index that holds file information. The remaining space on the platter is called non-catalogued.

Non-Common Variables
When overlaying programs, variables declared with DIM are cleared and variables declared with COM remain. The variables that are cleared are non-common.

Non-Interpretive Program
The RunTime Program (RTP) which can execute p-code that has already been generated. Program debugging, inspection, and modification are not available when using the non-interpretive program.

NPL Development Package
(see Development Package)

NPL Utilities
A group of programs included in the Niakwa Development Software that provides the developer with a set of Utilities that help in the development and maintenance of NPL programs.

Numeric
Numeric is a general term applied to any identifier in an NPL program which is of numeric type. For example, a numeric variable or a FUNCTION which returns a numeric. The following are examples of numeric identifiers:

```
num(20)                                  COS(2*PI)
rec$.employeePay                         totalAmount
'leastSquaresFit(points(), curve(), m, n)
```

numeric-array
A numeric array variable must be specified. A specific element of a numeric-array may not be specified.

Numeric Array
A variable that represents a one or two dimensional table used to hold integral or floating-point values.

numeric-constant

A numeric constant must conform to the format:

[+]d...[.]d]...][E[+]d[d]]
[+]   [-]

where:

d is a digit (0-9). Up to 13 digits may be specified. + or - are signs and may be either leading or trailing. E is the exponent symbol (may be preceded by an optional sign and followed by one or two digits). . is a decimal point.

Numeric Constant

Any integral or floating-point data in an NPL program which is not represented by a variable. For example:

10   123.456   99.99e-5

numeric-expression

A valid numeric expression must be specified. This may include numeric constants, functions, and variables. Refer to LET for further details on numeric expressions.

numeric field-expression:

An alpha-variable followed by a "." and a numeric field identifier. The numeric field identifier may be either a scalar field or an array element. A numeric field-expression refers to a single numeric field of a record, using the named alpha-variable as a buffer for the record.

numeric-receiver

A numeric scalar or numeric array element used to receive the results of an operation.

Numeric Receiver

Any numeric which may be assigned an integral or floating-point value. for example, a numeric variable or an element of a numeric array. Examples of numeric receiver are shown below.

any_num   num_table(10)
matrix(3,3)   taxTable()

numeric-scalar

A valid numeric scalar variable must be specified.

Numeric Scalar
Any numeric variable that is not an array.

numeric-variable
A valid numeric variable, either scalar or array, must be specified.

Once-a-day Security
The check NPL makes on a Novell NetWare installation that allows the "supervisor" to pass security for that day allowing all other users on the network to access the RunTime for the day without making a security check (since it was already passed for that day).

Operating Environment
A specific flavor or enhancement made to an existing operating system that allows for better user interaction (e.g., MS-Windows).

Operating System
The software that controls the operations of the system in use and allows user interaction (i.e., command structure, I/O, etc).

Operating System Specific Supplement
Documentation specific to the hardware platform being used by the NPL.

Overlay
A smaller component or module of a larger excutable program which is loaded into memory as needed and discarded when not in use. The use of overlays conserves memory use.

.profile File
Used under UNIX, Xenix, or AIX to define the user's system environment when logging in.

PATH
An environment variable which lists a set of default directories to be searched when files need to be located.

P-Code
(refer to Pseudo-Code)

Pixel Graphics
A "semi" or "block" type of graphics. This makes use of the 64 characters from hex C0 to hex FF from the NPL alternate character set.

Platform
The computer hardware upon which the native operating system is running.

Plot Operations
A plotter driver is available in the Niakwa Scientific and Communications Drive Package which allows NPL programs to plot functions on screens with graphics capabilities.

Pointer
A variable which contains the address of another variable. In NPL, pointer variables are only used in function or procedure parameter lists. This allows variables to be passed to functions or procedures by reference as opposed to being passed by value. A variable passed by reference can be modified within the function body, whereas a variable passed by value cannot.

Polling
Process of querying an I/O device such as the keyboard.

Preboot
A setup program used to set programming preference options and then load the NPL Boot Program, if any.

Printer Control Screen
A sub-screen of the NPL HELP system allowing a user to send configuration information to a printer.

Private
Variables or other identifiers may be declared so that they may be referenced directly by only the module or function which declares them. Such variables are said to be "private" to the module or function which makes the declaration.

PROCEDURE
Similar to a FUNCTION but does not return a value.

Process
Any executing program running independently of other running programs.

Programming Atoms
Assigning specific keys of the keyboard to generate BASIC-2 keywords with a single key-press.

Programmer's Guide

An NPL reference manual that provides the developer with a guide to NPL programming techniques. Part of the NPL Technical Reference Guide set.

Protected Mode

An enhanced operating mode of 80x86 microprocessors that allows access to more than 1 MB of memory. Protected mode also offers capabilities that allow operating system to support multi-tasking and protection of system resources. For 80386 class microprocessors, protected mode may also offer virtual memory capabilities.

Pseudo-Code (P-code)

An intermediate code generated by the compiler and interpreter. The code is designed to perform custom operations that are hardware independent.

Public

Variables, FUNCTIONs and some additional identifiers may be declared so that they may be referenced directly by any other module that INLUDEs the module in which the declarations appear. Such variables or FUNCTIONs are said to be public.

Quick Library

An external library of functions which may be dynamically linked to the RunTime under MS-DOS.

Raw Diskette

A physical disk which contains no native file system.

Real Mode

The 8086 operating mode of 80286, 80386 and 80486 microprocessors. In this mode only the first megabyte of memory may be accessed and there is no hardware support for multi-tasking or protection of system resources. On IBM PC-compatible systems, only the first 640K of this memory (so-called conventional memory) may normally contain programs and variables. The remaining 384K is reserved for I/O adapter RAM and ROM space. Access to conventional memory blocks larger that 64K requires the use of a segmented address.

Recursion

The process where a FUNCTION or PROCEDURE makes a call to itself.

Recursive Allocation
Each time a FUNCTION or PROCEDURE is called new space is allocated in memory for all private variables and labels; upon exiting, the space is de-allocated. If a FUNCTION or PROCEDURE makes a call to itself, then new space is allocated upon each call. That is, space is not reused just because the variable names are the same. Memory allocation in this manner facilitates the use of recursive function calls.

REDIRECT Option
An option under the HELP processor which gives control of a terminal to a serial port allowing remote support under MS-DOS.

Release
The number that refers to a particular Release of the RunTime (e.g., Release III, 4).

Remark
(see Line Remark)

Resettable Security
The ability of the MS-DOS based NPL RunTimes to have their Gold Key Security reset if the installable Gold Key Security is lost. This prevents the need for a replacement disk.

Resolve Time
The time between when a program is loaded to run and when it can start executing.

Return-Graphic
The graphic on an editing line that indicates the end of a statement.

Revision
The exact revision number of a particular release of an NPL RunTime (e.g., 3.21, 4.00).

Rights
The ability that users on a multi-user system are granted that allows them to perform certain files and access functions within a directory. These include the ability to read, write, create, delete, modify, etc., files within a directory.

RTI
The Niakwa program which allows interactive editing and debugging of NPL source programs. The Interpretive RunTime program.

RTP
The Niakwa program which will execute p-code generated by the interpreter or compiler. The Non-interpretive RunTime program.

RTPEXT Subroutine
A subroutine used in BESDK to provide NPL with information describing newly defined external functions. The routine provides the number for a GOSUB' statement, parameter count and type checking and LIST' information.

rtix
Custom UNIX version of RTI which has external subroutines linked to it.

rtpx
Custom UNIX version of RTP which has external subroutines linked to it.

rticx
Custom UNIX version of RTI which has external subroutines and NDM linked to it.

rtpcx
Custom UNIX version of RTP which has external subroutines and NDM linked to it.

RTIWIN
MS-Windows version of RTI.

RTPWIN
MS-Windows version of RTP.

RTIWIN.INI
Initialization file used by the RunTime under MS-Windows.

RTI386
Version of RTI to be used within the 386/DOS-Extender environment.

RTP386
Version of RTP to be used within the 386/DOS-Extender environment.

RTISHARE
Sharable version of RTI for Super DOS, which allows more than one user to run the executable at the same time.

RTPSHARE
Sharable version of RTP for Super DOS, which allows more than one user to run the executable at the same time.

RunTime
Either the RTI or the RTP, both of which allow the execution of NPL applications.

RunTime Options
Options specified at RunTime start-up (e.g., /R, /B).

RunTime Package
Package which contains the NPL Installation Guide, RTP, RTI, and their supporting files.

RunTime Program
RTI or RTP.

Scalar Variable
Any non-array variable.

Scientific and Communications Driver (SCD)
A package supplied by Niakwa which provides a driver for complete asynchronous communications by emulating the Wang 2227 communications board and a plotter driver to emulate plotting functions on screens with graphics capabilities.

SCD
(see Scientific and Communications Driver)

Scope (of a variable declaration)
The range of statements in a module for which a reference to a variable of the same name refers to the variable name used in the declaration.

Scrambled
An encryption scheme that is applied to programs in diskimages.

Scramble Protecting
The process of encrypting program code in diskimages.

Scratch
A file in a diskimage may be "scratched" to indicate that the data file or program is out of date, to allow replacement by a newer version, or to allow deletion when the diskimage is reorganized.

A new diskimage may also be scratched or an old one cleared for use (all files deleted) by a SCRATCH DISK operation.

Screen Character Set (NPL)
A generic character set used by NPL which may or may not be different from a native operating system's own character set.

Screen Translation Table
A table which is used to map the NPL screen characters to the native system screen characters so that they will appear the same on different terminals.

Script File
A file containing commands that are executed by the operating system or an application.

Security Fingerprint
Gold Key Security information that has been installed on a hard disk.

Security TSR
A program used by the NPL RunTime for security checks under MS-DOS based systems.

Serial Number
A number associated with every Gold Key security fingerprint. The number is visible from within NPL with the $SER system variable.

Shelling
Using the NPL $SHELL command to exit NPL temporarily and run a program in the native operating system.

simple-statement
A simple statement is a statement that can stand alone as a complete program. Some statements, in particular those used to implement structured programming constructs, are incomplete in that they require a corresponding statement to terminate them (e.g., WHILE requires WEND).

Special Function Keys
Some terminals have a row of 16 function keys which may be used in upper or lower case providing 32 assignable functions. The RunTime supports this through a virtual keyboard.

Statements Guide
An NPL Reference manual that lists all NPL statements, their use, and examples. Part of the NPL Technical Reference Guide set.

statement-label
An identifier which appears elsewhere in a program as a statement label (=identifier).

Statement Label
An identifier which appears in a program that is used as a target in a branching statement, such as GOTO or ON. In the following sample NPL program, the identifier "top_of_loop" is a statement label.

```
100   i = 10
      : =top_of_loop
      : i = i-1
      : IF i > 0 THEN GOTO top_of_loop
```

Static Allocation
Memory allocated for variables that remains for the duration of a program unless explicitly cleared with either a CLEAR V or CLEAR N command. The memory may be implicitly cleared when an overlay is executed.

Step Mode
A mode in the interpreter which allows stepped execution of an NPL program for debugging purposes. The program may be executed one statement at a time with all the facilities of immediate mode available between each executed statement.

Step Range
A specified range of line numbers between which stepped execution is allowed to occur.

Stepped Execution
Execution of one program statement at a time.

string field-expression:
An alpha-variable followed by a "." and a string field identifier. The string field identifier may be either a scalar field or an array element. A string field-expression refers to a single string field of a record, using the named alpha-variable as a buffer for the record.

String Ravel
A side effect which occurs with statements that use both the value of a string and assign a value to another string when the two strings overlap.

Structured Programming

Program development in a modular hierarchical fashion with a well organized flow (path) of execution. Main program tasks are broken down into smaller simpler tasks whose objectives are clearly and easily defined. Large or often repeated tasks are implemented through function and procedure calls. Sets of related functions and procedures are commonly stored in separate program files. Modern programming constructs such as SWITCH/CASE/END SWITCH, WHILE/WEND, REPEAT/UNTIL help with code organization and readability.

System Software

A program or set of program tools which help a software developer perform a task. Some examples would be a compiler or an assembler.

System Variable

A variable in NPL in which system information may be examined or modified. $OPTIONS and $SCREEN are examples of system variables.

Task

(see Process)

Technical Reference Guide (TRG)

Refers to the complete set of generic NPL documentation available to an NPL developer that applies to all NPL platforms (Programmer's Guide and Statements Guide).

Terminal Files

Definition files to allow NPL to work on specific terminals. These files are included with the NPL Development Software.

Terminal Type

Hardware specifics of a terminal or graphics display that a native operating system must know to communicate with it.

TRACE Mode

A mode used for debugging NPL programs. While in this mode variable modifications and control transfers can be displayed as a program is executing.

TRG

(see Technical Reference Guide)

ttys File

Used by the UNIX, Xenix, and AIX RunTimes to determine unique partition numbers for terminals using NPL.

UMB Memory

Upper memory block; memory above 640K (in the area normally reserved for I/O adapter RAM and ROM, but where the addresses are unused by configured devices) which may be made available to programs on 386 class microprocessors using a 386 memory manager driver such as EMM386.SYS or QEMM386.SYS. Also called High RAM.

Undeclared

If a program contains reference to an identifier which is not preceded by declaration for the identifier, the variable or function associated with the identifier is said to be undeclared. Here, the word variable is used in the loosest possible context. Undeclared references usually indicate a programming error or an attempt to reference a variable before its declaration.

Upgrade Guide

Documentation provided with the Upgrade RunTime Package describing the contents, configuration, installation and NPL Upgrade RunTime operation.

Upgrade Package

The Upgrade Guide and diskette media provided by Niakwa to upgrade a RunTime Package to a current NPL release level.

User Limit

A maximum number of users which may be running the RunTime at the same time for a given site.

User Partition

The part of memory available for use by a NPL program code and its defined variables.

Version

The third set of numbers in the NPL release number (e.g., 3.21.15, 4.00.23). This is also referred to as the minor revision level in the NPL documentation (refer to $REV is the NPL Statements Guide).

Variables

A location in memory, referenced by a program identifier, where a date value can be stored and changed during program execution.

Virtual Keyboard

The NPL language assumes the existence of an "ideal" keyboard that contains all of the keys required for the full use of the language. This is referred to as the virtual keyboard.

Virtual Machine Environment

To be fully portable across different platforms, NPL facilitates the handling of all hardware access in a generic manner. Physical (actual) I/O devices or files are mapped to logical devices or files through translation tables. Functionally, a software developer may create an application for no specific or "real" target system.

Virtual Memory

Virtual memory is an operating system technique which allows the operation of processes whose total size exceeds available physical memory. This occurs in such a way as to be invisible to any application program. Virtual memory requires the use of special hardware which is often integrated into the microprocessor. The operating system ensures that any portions of a process which are currently needed are in memory. When portions are not needed, the operating system ensures that the values are preserved, usually by writing them to a hard disk swap file.

XMS Memory

Extended memory specification. Allows MS-DOS programs to utilize extended memory, high RAM and the high memory area in an organized way.

Workspace

The memory used by all program modules, variables, etc. which comprise a single NPL application.

HEL SF'1 SF'1 SF'1 SF'1 SF'2 SF'2 SF'2 SF'2 SF'2 SF'2 SF'2 SF'2 SF'2 SF'2 SF'3 SF'3 EDIT/ HA

! # $ ^ * ( ) _ + BACKSP LOAD PRE] DTAB

> NEX

: "

, . ?

NPL Standard Characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1x	â	ê	î	ô	û	ä	ë	ï	ö	ü	à	è	ù	Ä	Ö	Ü
2x		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
6x	°	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	§	£	é	ç	¢
8x		♦	>	<	→		|	¨	’	‘	^		!!		ß	¶
9x	â	ê	î	ô	û	ä	ë	ï	ö	ü	à	è	ù	Ä	Ö	Ü
Ax	_	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
Bx	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
Cx	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Dx	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
Ex	°	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
Fx	p	q	r	s	t	u	v	w	x	y	z	§	£	é	ç	¢

NPL Alternate Characters

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td>1x</td><td>â</td><td>ê</td><td>î</td><td>ô</td><td>û</td><td>ä</td><td>ë</td><td>ï</td><td>ö</td><td>ü</td><td>à</td><td>è</td><td>ù</td><td>Ä</td><td>Ö</td><td>Ü</td></tr>
<tr><td>2x</td><td></td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>’</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>3x</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>4x</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>5x</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>\</td><td>]</td><td>↑</td><td>←</td></tr>
<tr><td>6x</td><td>°</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>7x</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td><td>y</td><td>z</td><td>§</td><td>£</td><td>é</td><td>ç</td><td>¢</td></tr>
<tr><td>8x</td><td></td><td>◆</td><td>▶</td><td>◀</td><td>→</td><td>└</td><td>|</td><td>¨</td><td>’</td><td>‘</td><td>^</td><td>■</td><td>!!</td><td>↕</td><td>ß</td><td>¶</td></tr>
<tr><td>9x</td><td>•</td><td>◊</td><td>▲</td><td>▼</td><td>↓</td><td>¬</td><td>✓</td><td>°</td><td>{</td><td>}</td><td>Δ</td><td>□</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Ax</td><td></td><td>-</td><td>-</td><td>—</td><td>|</td><td>┐</td><td>┌</td><td>┬</td><td>╵</td><td>┘</td><td>└</td><td>┴</td><td>│</td><td>┤</td><td>├</td><td>┼</td></tr>
<tr><td>Bx</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Cx</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Dx</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Ex</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Fx</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>