

PROGRAM DEVELOPMENT GUIDE

APRIL 1985

COPYRIGHT (C) 1984, 1985 GRiD Systems Corporation
2535 Garcia Avenue
Mountain View, CA 94043
(415) 961-4800

Manual Name : Program Development Guide
Order Number: 29300
Issue date: April 1985

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopy, recording, or otherwise, without the prior written permission of GRiD Systems Corporation.

The information in this document is subject to change without notice.

NEITHER GRiD SYSTEMS CORPORATION NOR THIS DOCUMENT MAKES ANY EXPRESSED OR IMPLIED WARRANTY, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, OR FITNESS FOR A PARTICULAR PURPOSE. GRiD Systems Corporation makes no representation as to the accuracy or adequacy of this document. GRiD Systems Corporation has no obligation to update or keep current the information contained in this document.

GRiD System Corporation's software products are copyrighted by and shall remain the property of GRiD Systems Corporation.

The following are trademarks of GRiD Systems Corporation: GRiD, GRiD Compass, Compass Computer, GRiD Server.

The following is trademarks of Intel Corporation: Intel.

TABLE OF CONTENTS

CHAPTER 1: THE PROGRAM DEVELOPMENT CYCLE

| | |
|---|-----|
| The Development Sequence | 1-1 |
| The Development Environment - GRiDDevelop | 1-3 |
| Conventions for Organizing and Naming Files | 1-3 |
| File Naming Conventions | 1-4 |
| File Titles | 1-4 |
| File Kinds | 1-5 |

CHAPTER 2: THE GRiDDEVELOP PROGRAM

| | |
|---|------|
| The GRiDDevelop Main Menu | 2-1 |
| GRiDDevelop Data Files | 2-2 |
| Creating GRiDDevelop Data Files | 2-3 |
| GRiDDevelop Tokens | 2-4 |
| The GRiDDevelop Pre-Defined Tokens | 2-4 |
| :Bad Tune: | 2-5 |
| :Control controlName: | 2-5 |
| :Debug: | 2-6 |
| :Enter: | 2-7 |
| :Exit: | 2-7 |
| :Good Tune: | 2-7 |
| :Groups: | 2-8 |
| :Link: | 2-8 |
| :Listings: | 2-9 |
| :Log File: | 2-10 |
| :Name: | 2-11 |
| :Objects: | 2-11 |
| :Prefix: | 2-12 |
| :Print To: | 2-12 |
| :Sources: | 2-12 |
| :Test: | 2-15 |
| User-Defined Tokens | 2-15 |
| Command Modifier Characters | 2-16 |
| The GRiDDevelop Commands Menu | 2-16 |
| Command Line Interpreter (CODE-C) | 2-17 |
| The Hex and Decimal Calculator (CODE-=) | 2-17 |
| Change Source Groups (CODE-G) | 2-17 |
| The Options Command CODE-O)..... | 2-18 |
| The Transfer Menu | 2-18 |
| Changing the Development Data File | 2-19 |
| Reading the Development Data File | 2-19 |
| Examining the Log File | 2-19 |
| Printing List and Source Files | 2-19 |

CHAPTER 3. COMPILERS, LIBRARIES, AND INCLUDE FILES

| | |
|--|-----|
| Compiling Programs | 3-1 |
| Compiler Size Controls | 3-1 |
| Libraries | 3-2 |
| Pascal Libraries and Modules | 3-2 |
| FORTRAN Libraries and Modules | 3-2 |
| PL/M Libraries and Modules | 3-3 |
| 8087 Libraries and Modules | 3-3 |
| Invoking the Compilers | 3-3 |
| The System and Language Include Files | 3-4 |
| Examples of Include Control Statements | 3-5 |

CHAPTER 4. THE LINK PROGRAM

| | |
|---------------------------------|-----|
| Invoking the Link Program | 4-1 |
| Link Invocation Examples | 4-2 |
| Link Control Summary | 4-3 |
| Assumeroot | 4-4 |
| Bind | 4-4 |
| Fastload | 4-5 |
| Map | 4-5 |
| Name | 4-6 |
| Overlay | 4-6 |
| Print | 4-7 |
| Printcontrol | 4-7 |
| Purge | 4-8 |
| SegSize | 4-8 |
| The Linker's Print File | 4-9 |

CHAPTER 5. THE DEBUGGER

| | |
|---|------|
| Compile and Link Considerations | 5-2 |
| Invoking the Debugger | 5-2 |
| Debugger Syntax and Terminology | 5-3 |
| CODE-Key Commands | 5-4 |
| The Help (CODE-?) Command | 5-4 |
| The Set Breakpoint (CODE-B) Command | 5-4 |
| The Clear Breakpoint Command | 5-5 |
| The Timing Breakpoint Commands (CODE-1 -- CODE-5) | 5-5 |
| The Duplicate Line (CODE-D) Command | 5-7 |
| The Executive (CODE-E) Command | 5-7 |
| The Fill Memory (CODE-F) Command | 5-7 |
| The Info (CODE-I) Command | 5-8 |
| The Location Display (CODE-L) Command | 5-8 |
| The Message Display (CODE-M) Command | 5-8 |
| The Options (CODE-O) Command | 5-8 |
| The Proceed (CODE-P) Command | 5-8 |
| The Quit (CODE-Q) Command | 5-9 |
| The Register Display (CODE-R) Command | 5-9 |
| The Stack Trace (CODE-S) Command | 5-10 |
| The Tasks/Semaphore Display (CODE-T) Command | 5-10 |

| | |
|--|------|
| The Unassemble (CODE-U) Command | 5-11 |
| The Window Toggle (CODE-W) Command | 5-11 |
| Command Line Commands | 5-11 |
| The Display Address Command | 5-12 |
| The Display Contents Command | 5-12 |
| The Assign Value Command | 5-13 |
| The Memory Dump Command | 5-13 |
| The Examine/Change Memory Command | 5-14 |

APPENDIX A. ALTERNATE DEVELOPMENT APPROACHES

| | |
|---|-----|
| Using the Development Executive Program | A-2 |
| Using the DO Program with Command Files | A-3 |
| Executing Command Files from the User Interface | A-4 |

APPENDIX B. PROGRAM OVERLAYS

| | |
|---|-----|
| The OsOverlay Procedure | B-1 |
| Pascal Overlay Example | B-2 |
| Linking Overlays | B-4 |
| Additional Overlay Considerations | B-5 |
| FORTTRAN Overlay Example | B-5 |

APPENDIX C. SYSTEM FILES AND UTILITIES

| | |
|---|------|
| Syntax Notation | C-1 |
| Entering Commands | C-2 |
| Wildcards | C-2 |
| The System Errors File | C-2 |
| The Activate Program | C-3 |
| The Catalog Program | C-4 |
| Creating a Catalog File | C-5 |
| ! (Exclamation Point) | C-5 |
| ? (Question Mark) | C-5 |
| The Compare Program | C-6 |
| The Deactivate Program | C-6 |
| The Development Executive Program | C-7 |
| The DO Program | C-7 |
| The Dump Program | C-7 |
| The Elapsed Time Program | C-8 |
| The Executive File | C-8 |
| The LADT (List Active Device Table) Program | C-8 |
| The Load Program | C-10 |
| The Prefix Program | C-10 |
| The Softkeys Files | C-11 |
| Programming the Softkeys | C-11 |
| Multiple Softkey Files | C-12 |
| The Status Program | C-12 |
| The Summarize Program | C-13 |
| The Time Program | C-14 |
| The Unload Program | C-14 |
| The Work Program | C-14 |

APPENDIX D. LINK ERROR MESSAGES

APPENDIX E. SOUND

ABOUT THIS BOOK

This book describes how to use the GRiD Compass as a program development tool. To assist you in program development, a powerful and easy-to-use development tool -- GRiDDevelop -- is available. GRiDDevelop provides a flexible development environment where you can quickly edit your source files, compile and link your programs, and then proceed to the debugging, re-editing cycle.

Five programming languages are currently provided: Pascal-86, PL/M-86, FORTRAN-86, C and Assembler-86.

You create program source files using the text editor program, GRiDWrite. The source files (along with any required INCLUDE files) are then compiled using the appropriate compiler (Pascal, Fortran, PLM, C, ASM). Guidelines for using the languages and their compilers are provided in Chapter 3 of this manual. The INCLUDE files are also listed and briefly described in Chapter 3.

The compilers produce list files and relocatable object modules. These modules, along with other modules you may have compiled and library modules, are then linked together using the Link program described in Chapter 4. Programs can be debugged on the GRiD Compass with the GRiD Debug program described in Chapter 5.

A number of useful system utility programs are also available to ease system maintenance tasks accompanying the program development sequence. These programs are described in Appendix C.

CHAPTER 1. THE PROGRAM DEVELOPMENT CYCLE

The GRiD Compass gives you great flexibility in defining how you use the computer and its software when developing programs. The GRiDDevelop program is a powerful and easy-to-use tool that helps you organize your files and greatly speed up the development process. GRiDDevelop is described in detail in Chapter 2. Before discussing GRiDDevelop, however, let's take an overview of typical development sequences and the available tools to assist program development.

The Development Sequence

Figure 1-1 illustrates the general sequence followed when developing programs and also shows some of the software tools that are provided.

Create/Edit->|---Compile--->|----Link---->|Debug

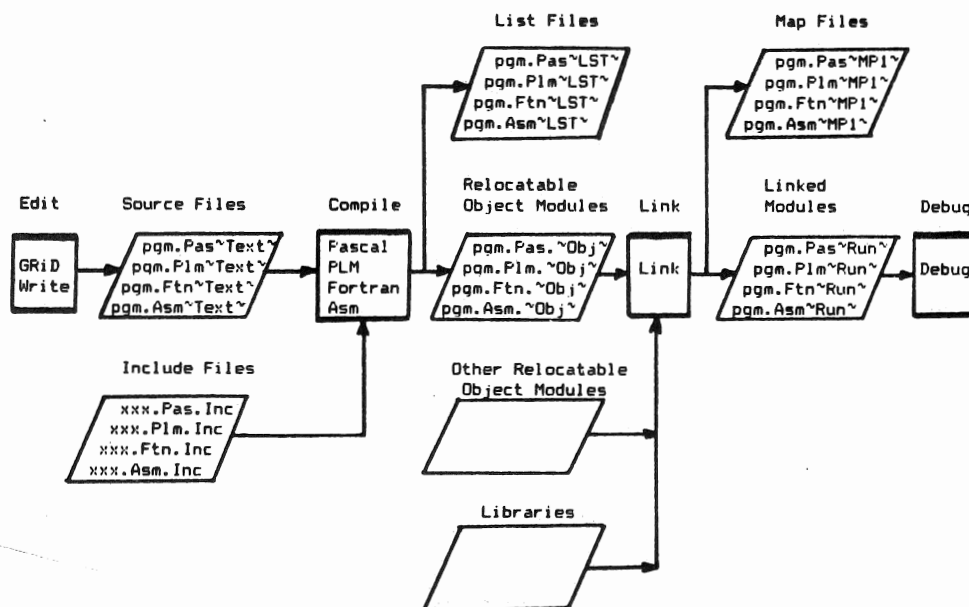


Figure 1-1. The Program Development Sequence

The development process consists of four iterative phases:

- o Editing (writing) program source files with GRiDWrite
- o Compiling source files with one of the language (Pascal, PLM, Fortran, Assembler) compilers
- o Linking compiled object modules with the Link program to produce modules which can be executed (run)
- o Testing and debugging the executable modules.

This four-step sequence is repeated while you refine and debug your program.

When you are creating the text source file for a program, GRiDWrite speeds the process with such features as automatic indentation and fast substitution and duplication of phrases and whole sections of code. For a complete description of GRiDWrite, refer to the GRiDWrite section in the GRiD Management Tools Reference manual.

After you have finished writing or correcting a source program, you must invoke the compiler to translate your text file into an object file.

Invoking the linker program requires a more complicated sequence since it usually involves naming a number of files that are to be linked together. For example, here is a typical linker invocation:

LINK Shell.Pas~Obj~, FormsInit.Plm~Obj~,

```
`w'Libs'DataForms.Pas~Obj~, `w'Libs'largeException.Asm~Obj~,
PQ(`w'libs'SystemCalls~Pub~) TO Shell~Run~ BIND PURGE FASTLOAD
ss(stack(+1000))
```

Typically, you might edit several modules, then compile them one after the other, and finally link the modules together along with various libraries. You would then test and debug the linked, executable module. If errors are discovered, you would repeat the edit/compile/link sequence. The goal of the program development environment is to make this repetitive sequence as easy and fast as possible.

THE DEVELOPMENT ENVIRONMENT -- GRiDDevelop

The GRiDDevelop program (described in Chapter 2) provides a development environment based on the assumption that most software development consists of edit/compile/link/test cycles. You can define many of the characteristics of the environment by filling out a data file with information about source file names, link command lines, subjects for sources, listings, and objects, and other miscellaneous commands. The GRiDDevelop program reads this file for the data to drive the program development cycle.

You use GRiDDevelop data files to specify the files that the GRiDDevelop program will operate on to initiate various development activities such as editing, compiling, linking, and so on.

When you use GRiDDevelop to provide the development environment, GRiDWrite is automatically invoked so you can create, edit and correct your source programs. GRiDDevelop also automatically invokes the appropriate compiler required for your source programs and lets you set any controls that you want to use during the compilation.

Link statement files are set up in the GRiDDevelop data file so that you can issue a complicated link statement with a single keystroke. You can also easily edit the link statement(s) during the development cycle directly from GRiDDevelop.

You have several other alternatives when deciding on the environment you want to use when developing programs using the GRiD Compass. Although we suggest that you use the GRiDDevelop program, since it provides the fastest and most flexible environment, we describe some alternate approaches in Appendix A.

CONVENTIONS FOR ORGANIZING AND NAMING FILES

Although there are few hard and fast rules for organizing your directories and naming files, there are some conventions that have been adopted internally at GRiD and which are assumed by the GRiDDevelop program. Even if you do not use GRiDDevelop, observing

these conventions will be of value to anyone doing program development work using the GRiD Compass.

Figure 1-2 illustrates part of a typical directory on a hard disk device.

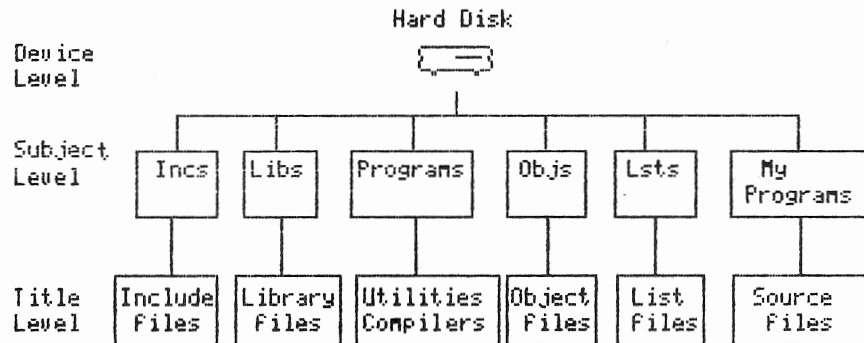


Figure 1-2. Organization of Typical Directory

The purpose of this organizational style is to keep all files that are logically related in the same directory. This keeps the number of titles within each directory from getting too large. This organization also simplifies such maintenance activities as backing up files and obtaining new copies of files, and standardizes references your programs make to include files and libraries. The directory organization shown in Figure 1-2 puts all the include files under one subject (Incs), all library files under one subject (Libs), all object files under the Objs subject, all source files related to a particular programs under MyPrograms, and so on.

FILE NAMING CONVENTIONS

There are two file naming considerations: the file title and the file kind (or type).

File Titles

GRiD-OS imposes two small limitations on file names. First, characters used in the title can be any of the printable ASCII characters between 'space' (ASCII code 20 hex) and DEL (ASCII code 7E Hex) except for the single backquote (`) and tilde (~) characters. Second, the file name cannot exceed 253 characters total including device, subject, title, kind, and the delimiter characters.

The Intel compilers, however, place greater restrictions on file names. They require that file names (including device, subject, title, kind, and the delimiter characters) be no longer than 45 characters. You should therefore ensure that your program names do not exceed this limit.

GRiDDevelop makes some assumptions about file names. (Note: Even if you do not use GRiDDevelop, it is recommended that you observe these conventions.) The first assumption GRiDDevelop makes is that you append some language identification information to all source file titles. For example a Pascal source file should have the name `MyProgram.Pas~Text~`, a PL/M version of this program would be named `MyProgram.Plm~Text~`, an assembly language version would be `MyProgram.Asm~Text~`, and a FORTRAN version would be `MyProgram.Ftn~Text~`. This convention lets GRiDDevelop automatically invoke the appropriate compiler for your source programs. It also makes it easier to organize your files and identify the file you want even if you do not use GRiDDevelop.

The other convention is to identify the include files (for any language) by appending `.Inc` to the name: for example, `MyProgram.Inc~Text~`.

File Kinds

GRiD-OS and some GRiD applications require that files be of a certain "kind" in order to perform some activities. For example, if a file is an executable program, the system requires that its kind be `~Run~`; otherwise, the file cannot be executed. The file kind suffix also provides additional information about the contents of the file so that you can tell quite a bit about a file just by looking at its kind.

When you are running application programs under GRiD-OS with the Executive program, you can select a data file, and the system automatically invokes the executable program to work or operate on that data file. The program that will be implicitly invoked to do the work must be of kind `~Run fileKind~`, where "fileKind" matches the Kind of the file being selected. For example, the program `GRiDWRITE~Run Text~` works with a file that has a Kind of `~Text~`. The file that is to be operated on must have a kind that matches the fileKind of the program being implicitly invoked. For example, a file named Memo that you want to edit using GRiDWrite would be of kind `~Text~`. Thus, its complete name would be `Memo~Text~`.

To implicitly invoke and initiate execution of the application program (the program that is to do the work), just select the subject and title of the file you want from the File form and press `CODE-RETURN`.

During the program development cycle, some file kinds are appended

automatically by various utilities; others must be appended by the user. Some of the file kinds you will encounter and use are listed below in alphabetical order.

(NOTE: Some kinds will always appear in all caps while others are shown with just the first letter capitalized. Those in all caps are appended by Intel software such as the compilers. GRiD-OS, however, does not differentiate between upper and lower case: you can use any mix of upper and lower case in file names.)

- ~Com~ A Com (Command) file contains a list of executable files. You must add the ~Com~ kind when naming the file.
- ~LIB~ This kind is usually appended by the Lib utility program to identify modules that are part of the library, although you can specify any kind you want with the Lib program.
- ~LST~ The compilers create LST (LiST) files. LST files are program listings with statement and line numbers, error messages, and other programming information. The compiler automatically adds the ~LST~ kind.
- ~MP1~ An MP1 file is the linker's Map file. The Link program automatically appends the ~MP1~ kind.
- ~OBJ~ The compilers generate unlinked OBJ (Object) files and append the ~OBJ~ kind.
- ~Run~ Run files are executable files that are created by the Link program. Note, however, that you must specify the ~Run~ kind yourself to the output file title in the link statement.
- ~Text~ Any file created with GRiDWrite will have the kind ~Text~ appended to it unless you explicitly specify that it be of a different kind (such as ~Com~). Thus, the source text for a program that you create using the text editor will usually have ~Text~ appended to its title.

THE GRIDDDEVELOP PROGRAM

The GRIDDDevelop program provides a development environment based on the assumption that most software development consists of reiterative Edit/Compile/Link/Test/Debug cycles. You can define many of the characteristics of the environment by filling out a data file with information about source file names, link command lines, subjects for sources, listings, and objects, and other miscellaneous commands. The GRIDDDevelop program reads this file for the data to drive the program development cycle.

The GRIDDDevelop program resides in memory at all times. In order to use it, you must have the compilers (that you use), and the programs GRIDWrite and Print in the programs subject.

THE GRIDDDEVELOP MAIN MENU

The GRIDDDevelop program is invoked by filling out the File form specifying a file with a kind of Develop. The program then displays the main GRIDDDevelop menu shown in Figure 2-1. To initiate one of the activities listed on the menu, just select and confirm. The GRIDDDevelop Main menu is the default menu; once you have invoked the GRIDDDevelop program, this menu will be displayed whenever you press ESC.

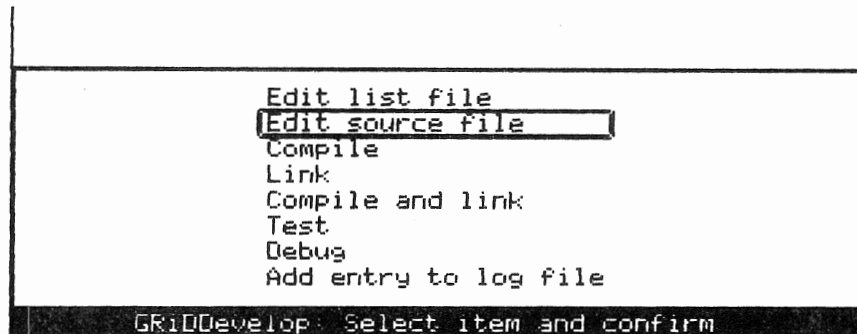


Figure 2-1. The Main GRiDDevelop Menu

The files that are to be acted on by each of these menu items are specified in the GRiDDevelop data file. For example, selecting "Edit source file" brings up a menu displaying the filenames you have specified in the data file. Figure 2-2 shows an example of a list of source files. Confirming one of these files invokes GRiDWrite and brings in the file you have selected so that you can edit the source program.

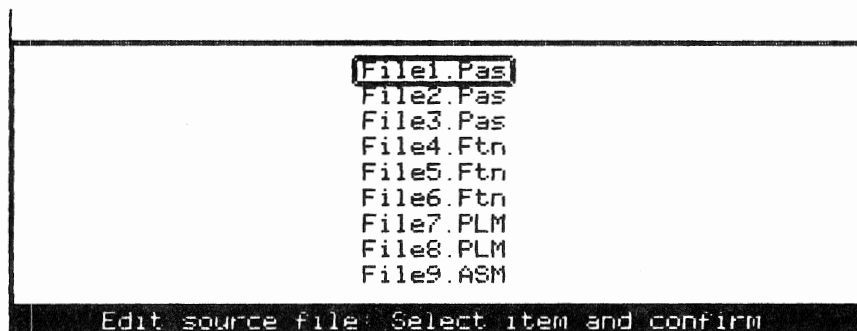


Figure 2-2. The Edit Source File Menu

All of the items listed on the Main menu in Figure 2-1 will be described as we proceed through this chapter. Since a description of the GRiDDevelop data file provides an understanding of how the Main menu works, we will discuss items on the Main menu in conjunction with the data file.

GRiDDEVELOP DATA FILES

You use GRiDDevelop data files to specify the files that will be operated on during the various development activities such as

editing, compiling, linking, debugging, and so on. The data file consists of text comprising tokens, filenames, and command lines. The data files have a Kind of "Develop".

You can have as many GRiDDevelop data files as you want -- use a separate one for each development task you have in progress. Each data file requires no more space than a small text file -- typically one to three thousand bytes.

Once a GRiDDevelop data file has been created using GRiDWrite, it can be easily edited to meet changing demands of the development cycle. You can add source files designations, change Link command statements, and so on, by selecting "Change this development file" from GRiDDevelop's Transfer menu which is described later in this chapter.

Creating GRiDDevelop Data Files

There are two ways that you can create a new GRiDDevelop data file: the method you use depends on whether you already have a Develop file in your system.

If there are no existing Develop files in your system, follow these steps:

1. Fill in a File form specifying a file with a Kind of "Develop" and confirm the form. The system will prompt you with the message "Confirm to create new file". Confirm the form again to create the new develop file.
2. GRiDDevelop will be loaded into memory and will display its Transfer menu with the selection outline surrounding the item "Change this develop file". Confirm this selection.
3. The GRiDWrite program will be invoked by GRiDDevelop along with the file (currently empty) that you specified in step 1.
4. Type the tokens you need (described beginning immediately after this section) into the text file. When you have finished specifying the desired tokens, press CODE-Q and confirm to quit GRiDWrite. Control is returned to GRiDDevelop which displays its Main menu. The development process is now being guided by the new Develop file you have just defined.

If you already have a Develop data file in your system, it is usually easier to simply make a copy of the existing file (using GRiDManager or the "Write to a file" item from GRiDWrite) making sure that the new file also has a Kind of Develop. You can then select the new file from the File form to invoke GRiDDevelop. Now you can edit the new file (using the "Change this develop file" item

from GRiDDDevelop's Transfer menu) to fit the requirements of the new development project.

A sample Develop data file is provided on diskette to simplify getting started with GRiDDDevelop. Use it as a model that can be edited to fit your specific needs. A complete listing of this sample Develop file is provided at the end of this chapter.

GRiDDDEVELOP TOKENS

GRiDDDevelop recognizes all text in a data file that is enclosed within a pair of colons (:xxx:) as "tokens". A token is interpreted by GRiDDDevelop as a command specifying how it should handle the text that immediately follows the token. GRiDDDevelop recognizes a set of pre-defined tokens and can also accept user-defined tokens. Pre-defined tokens are operated on by GRiDDDevelop in a predetermined manner. For example, the GRiDDDevelop token ":Sources:" tells GRiDDDevelop to treat the files listed after the token as source files that can be edited and compiled. User-defined tokens are simply any tokens not pre-defined by GRiDDDevelop -- you specify the activity that should be initiated by GRiDDDevelop as a result of the user-defined token. The paragraphs that follow describe all of the pre-defined tokens. Examples of user-defined tokens will be described later in this chapter.

Predefined GRiDDDevelop Tokens

The following tokens are pre-defined and cause GRiDDDevelop to initiate specific activities:

- :Bad Tune:
- :Control:
- :Debug:
- :Enter:
- :Exit:
- :Good Tune:
- :Groups:
- :Link:
- :Listings:
- :Log File:
- :Name:
- :Objects:
- :Prefix:
- :Print To:
- :Sources:
- :Test:

Each of these tokens is described in detail in the pages that follow.

:Bad Tune:

Each development data file can specify one of these tokens. The Bad Tune token lets you define a sequence of notes that will be output to the speaker in the Compass computer. This token assumes that you are using a Compass that is equipped with a built-in modem and that you have the file named Sound~Device~ in the Programs subject of your system. This device must be activated either via the Command Line Interpreter, a System.Init file, or by using an :Enter: token. See the :Enter: token later in this chapter for an example.

The :Bad Tune: token lets you enter a text string following the token. The characters in this text string are interpreted as sound, or a "tune", according to the rules described in Appendix E. The tune defined by the character string after the :Bad Tune: token will be "played" whenever an error occurs during a compilation or link operation.

:Control controlName:

Each development data file can specify as many of these tokens as needed to set up controls that will be presented as choices on the "Compile form". The token is followed by a list of the controls that are to be used when the compiler is invoked. For example, the following token

:Control Yes with Debug: DEBUG NOPRINT

would display the form shown below when you select Compile from the Main menu:

| | No | Yes | Yes with Debug |
|----------------|-------------------------------------|--------------------------|--------------------------|
| File1.Pas..... | <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File2.Pas..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File3.Pas..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File4.Ftn..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File5.Ftn..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File6.Ftn..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File7.PLM..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File8.PLM..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| File9.ASM..... | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Compile source files Fill in form and confirm

You can then specify which files are to be compiled and which are to be compiled with the DEBUG NOPRINT controls applied.

: Debug :

Each development data file can specify one or more Debug tokens. Following the each Debug token, you can specify any number of command lines each of which must be ended by pressing RETURN. Typically, one of these command lines would be an invocation of the debugger along with your program. The following is an example of the use of the Debug token:

```
:Debug:
  Debug MyProgram~Run~ TestFile~Text~
```

Now, when you select the Debug item from the Main menu, the Debug program is invoked to operate on the MyProgram~Run~ file which uses TestFile~Text~. After this sequence has been completed, you would automatically be returned to the Main menu.

If more than one debug command file is needed, then multiple debug tokens can be defined in a GRiDDDevelop data file. Any text that follows the keyword Debug and that is enclosed within the colons will be displayed on the Debug menu. You can then select which of the command sequences is to be performed as part of the debugging sequence. You can specify as many of these tokens as required in each development data file. The following example illustrates the use of multiple Debug tokens:

```
:Debug Use test data file #1:
  Debug MyProgram~Run~ TestFile#1~Text~

:Debug Use test data file #2:
  Debug MyProgram~Run~ TestFile#2~Text~
```

Now, when you select Debug from the Main menu, the following Debug menu would be displayed.

| |
|---|
| <div>Use test data file #1</div> <div>Use test data file #2</div> |
| Debug Select item and confirm |

You can now select the debug command file you want from this menu and the debugger will be invoked along with your program and the desired test data file.

: Enter :

Each development data file can specify one Enter token. The token is followed by a command or series of commands that are to be executed the first time the development data file is brought into memory. For example, the following token

```
:Enter:
    Activate ``Hard disk``programs`Sound~Device~
```

causes the Sound device to be activated when this development data file is first brought into memory. (The Sound device is used in conjunction with the :Good Tune: and :Bad Tune: tokens described elsewhere in this chapter.)

: Exit :

Each development data file can specify one Exit token. The token is followed by a command or series of commands that are to be executed when you exit the GRiDDevelop program. For example, the following token

```
:Exit:
    Deactivate ``Hard disk``programs`Sound~device~
```

causes the Sound device to be deactivated when you exit the GRiDDevelop program.

: Good Tune :

Each development data file can specify one of these tokens. The Good Tune token lets you define a sequence of notes that will be output to the speaker in the Compass computer. This token assumes that you are using a Compass that is equipped with a built-in modem and that you have the file named Sound~Device~ in the Programs subject of your system. This device must be activated either via the Command Line Interpreter, a System.Init file, or by using an :Enter: token. See the :Enter: token earlier in this chapter for an example.

The :Good Tune: token lets you enter a text string following the token. The characters in this text string are interpreted as sound, or a "tune", according to the rules described in Appendix E. The tune defined by the character string after the :Good Tune: token will be "played" whenever a pause is processed (see "Command Modifier Characters" later in this chapter for a discussion of "pauses") or when a compile, link, or compile and link operation is successfully completed.

: Groups:

Each development data file can specify one of these tokens to indicate which group or groups of source files should be displayed as choices for such activities as editing or compiling. (Refer to the :Sources: description for a description of how to create groups of files.) If the :Groups: token is not specified, then all groups will be displayed when the Develop file is first entered. The :Groups: token lets you specify which groups of files should be initially displayed. For example, if you have groups named Sample1, Sample2, and Sample3 you can specify that the files in groups Sample1 and Sample3 be initially displayed with the following token:

```
:Groups: Sample1
        Sample3
```

NOTE: the :Groups: token must appear after all :Sources: tokens in the Develop file.

While you are in GRiDDevelop, you can change the groups that are displayed using the CODE-G command. However, settings established using CODE-G are discarded when you leave GRiDDevelop and the :Groups: token will be used upon reentry to GRiDDevelop.

: Link:

Each development data file can specify one or more Link tokens. Following the Link token, you can specify any number of command lines, each terminated by pressing RETURN. Typically, this would include a command line invoking the linker program (Link~Run~) and naming the files that are to be linked and the resultant output file. The entire link statement is one command line and must be terminated with RETURN. The following is an example of the use of the Link token:

```
:Link:
Link ``Hard disk``Objs`epMain.Pas~Obj~, ``Hard
disk``Objs`epFolders.Pas~Obj~, ``Hard
disk``Objs`epUtility.Pas~Obj~, ``Hard
disk``Objs`epFormsInit.Plm~Obj~, ExecPac~Font~, ``Hard
disk``Libs`CompactException.Asm~Obj~, ``Hard
disk``Libs`CompactSystemCalls~Lib~ TO ExecPac~Run~ BIND NOPURGE
NOFASTLOAD PC(PURGE) ss(stack(2000)) PRINT(``Hard
disk``Lsts`ExecPac~MP1~)
```

If more than one link file is needed (for example, when linking overlays), then multiple links can be defined in a Development file. Any text that follows the key word Link and that is enclosed within the colons is displayed on the Link form and you can select which link command statement is to be performed. You can specify as many of these tokens as required in each development data file. The

following example illustrates the use of multiple Link tokens:

```
:Link Root:
LINK SampleRoot.Pas~Obj~, ``Hard disk``Libs~p86rn0~lib~, ``Hard
disk``Libs~p86rn1~lib~, ``Hard disk``Libs~p86rn2~lib~, ``Hard
disk``Libs~p86rn3~lib~, ``Hard disk``Libs~8087~Lib~, ``Hard
disk``Libs~LargeSystemCalls~Lib~, ``Hard
disk``Libs~DqLarge~Lnk~ TO SampleRoot~Lnk~ OVERLAY(ROOT)
NOPRINT

:Link Overlay1:
LINK SampleOverlay1.Pas~Obj~, ``Hard disk``Libs~p86rn0~lib~,
``Hard disk``Libs~p86rn1~lib~, ``Hard disk``Libs~p86rn2~lib~,
``Hard disk``Libs~p86rn3~lib~, ``Hard disk``Libs~8087~Lib~ TO
SampleOverlay1~Lnk~ OVERLAY(SampleOverlay1) NOPRINT

:Link Overlay2:
LINK SampleOverlay2.Pas~Obj~, ``Hard disk``Libs~p86rn0~lib~,
``Hard disk``Libs~p86rn1~lib~, ``Hard disk``Libs~p86rn2~lib~,
``Hard disk``Libs~p86rn3~lib~, ``Hard disk``Libs~8087~Lib~ TO
SampleOverlay2~Lnk~ OVERLAY(SampleOverlay2) NOPRINT

:Link Run Sample:
LINK SampleRoot~Lnk~, SampleOverlay1~Lnk~, SampleOverlay2~Lnk~
TO SampleRoot~Run~ BIND SS(STACK(+1500)) PC(PURGE)
```

Now, when you select the Link item on the Main menu, the following Link form will be displayed:

| | |
|---|---------------------------------|
| <input checked="" type="checkbox"/> No <input type="checkbox"/> Yes <input type="checkbox"/> Display only | |
| Root..... | <input type="text" value="No"/> |
| Overlay1..... | <input type="text" value="No"/> |
| Overlay2..... | <input type="text" value="No"/> |
| Run Sample..... | <input type="text" value="No"/> |
| Link Fill in form and confirm | |

You can then select which link command file(s) you want to be executed from this form.

: Listings:

You can specify one of these tokens in each development data file. The device-subject string you specify (for example, ``Hard Disk``Lsts``) is automatically prepended to the filenames that you have specified with the :Sources: token when those source files are compiled. The ~LST~ extension is automatically appended to the resultant files by the compiler. Here is an example of using the :Listings: token:

```
:Listings: ``Hard disk``Lsts`
```

Then, if you compiled source files (fileName1 - fileName4) this would produce list files having pathnames as follows:

```
``Hard disk``Lsts`fileName1~LST~  
``Hard disk``Lsts`fileName2~LST~  
``Hard disk``Lsts`fileName3~LST~  
``Hard disk``Lsts`fileName4~LST~
```

: Log File:

You can specify one of these tokens in each development data file. The log file can be used to record or log your activities as you write, debug and make changes to programs. The pathname that you specify following the token can be of kind Database or Text. If you specify a log file of kind Database, the GRiDFile program will be invoked by GRiDDevelop to display the contents of the log file. If you specify a kind of Text, GRiDWrite will be used to display the file. If you do not specify a kind along with the log file token, it is assumed that the file is a database and GRiDFile is used. NOTE: this assumes, of course, that you have the GRiDFile program in your system.

To make entries into the specified log file, select the "Add entry to log file" item from the main GRiDDevelop menu. The following form is then displayed:

| | |
|---------|----------------------|
| Number | <input type="text"/> |
| Version | <input type="text"/> |
| Module | <input type="text"/> |
| Comment | <input type="text"/> |

Log file entry. Fill in form and confirm

The form provides four different fields that you can fill out to keep track of programming activities. The information you put into each of the fields is entirely up to the you but the form was designed with the following uses in mind.

The first field, "Number", can be used to record such things as error-report or enhancement-request tracking numbers. The "Version" field can be used to record the version number of the program module(s) currently being worked on. The "Module" field can record the name of the program module(s) being modified and the "Comment" field can be filled out to describe the kinds of changes being made to the module(s).

When you have completed the form with the information you want recorded, press CODE-RETURN to log the entry. As each entry is made in the log file, GRiDDevelop automatically appends the current date and time into the log file as a prefix to each entry.

The Log file entry form is cleared as you confirm each entry to indicate that the entry has been recorded. A blank form is then displayed to allow additional entries. To return to the Main menu after logging entries, press ESC.

To examine the contents of a log file, press CODE-T to display the Transfer menu and then select the "Examine log file" item. If the log file was specified with a kind of Database, GRiDFile will automatically be invoked and you can use the Find command (CODE-F) of GRiDFile to display the contents of the file. If the log file is of kind Text, GRiDWrite is invoked and the contents of the file will be displayed automatically.

: Name :

You can specify one of these tokens in each development data file. The string you specify is automatically displayed as the leading phrase in the message line of the main GRiDDevelop menu. For example, if you specify the following with this token

:Name: Sample Development

the screen displayed by GRiDDevelop would be as shown in Figure 2-3.

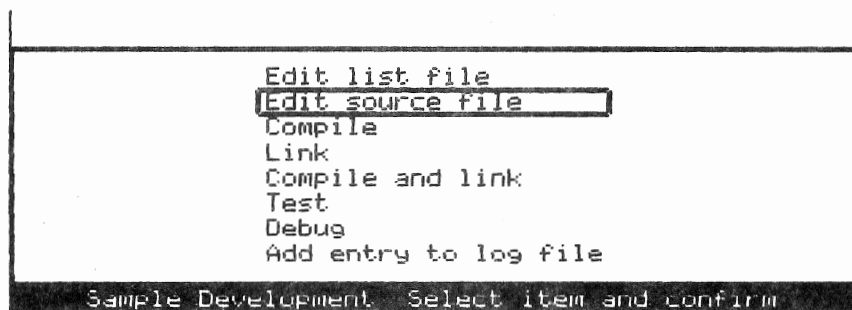


Figure 2-3. Using the Name Token

: Objects :

You can specify one of these tokens in each development data file. The device-subject string you specify (for example, "Hard Disk" "Objs") is automatically prepended to the filenames that you have specified with the :Sources: token when those source files are compiled. The ~Obj~ extension is automatically appended to the

resultant files by the compiler. Here is an example of using the :Objects: token:

```
:Objects: ``Hard disk``Objs`
```

This would cause the object files produced by compilers to have pathnames as follows:

```
``Hard disk``Objs`fileName1~Obj~  
``Hard disk``Objs`fileName2~Obj~  
``Hard disk``Objs`fileName3~Obj~  
``Hard disk``Objs`fileName4~Obj~  
  (and so on)
```

: Prefix:

You can specify one of these tokens in each development data file. The string you specify is the Device-Subject name or simply the Subject name where source files reside. It is recommended that you usually use only the subject name to specify the prefix -- GRiDDevelop will automatically prepend the current system prefix device. This token lets you define source file names in the data file by just specifying the Title: the prefix you specify in the data file will automatically be prepended to the Title. After each command, the prefix is reset to the Subject specified with this token. Here are two examples of using the :Prefix: token

```
:Prefix:  `I/O Driver`
```

```
:Prefix: MyPrograms
```

: Print To:

You can specify one of these tokens in each development data file. The string you specify is prepended to the destination filename before printing and is used to direct printing to a remote device such as GRiDServer. Here is an example of the :Print To: token

```
:Print To: ``Nexus.1:Printer Queue`EpsonFX80``
```

If you do not specify a Print To token, then GRiDDevelop assumes that all printing will be to your local (directly attached) printer. See the section titled "Printing List and Source Files" later in this chapter for additional discussion of the effects of this token.

: Sources:

You can specify one or more of these tokens in each development data file. Following the token, you provide a list of source filenames.

Filenames must be one per line and can include spaces. Leading and trailing spaces are ignored unless enclosed in quotes. Each filename title must end with a suffix indicating the compiler to be invoked for that source file. The following title suffixes are recognized by GRiDDevelop:

```
.Pas  (Pascal)
.Plm  (PL/M)
.Asm  (Assembler)
.Ftn  (FORTRAN)
.C    (C)
```

Here is an example of the :Sources: token

```
:Sources:
  ModelPriv.Inc
  ModelText.Inc
  FormsInit.Plm
  Unparse.Pas
  'Test Model.Pas'
```

Note that if the file name title ends with a suffix other than one of the five recognized by GRiDDevelop, no compiler will be invoked. This lets you have "include" files (.Inc) be listed with your source files so that you can easily edit them using the "Edit source file" command from the Main menu of GRiDDevelop.

If you have programs that have many source files, this token lets you categorize a collection of source files as a "group". Then, by selecting the "Change source groups" item on the GRiDDevelop Commands menu or by pressing CODE-G, you can specify which group(s) of files should be displayed for editing, compiling, and so on (see also the :Groups: token). You can specify as many of the :Sources: tokens as required in each Develop data file. The following example illustrates the use of the multiple :Sources: tokens:

```
:Sources Sample1:
  File1.Pas
  File2.Plm
  File3.Asm

:Sources Sample2:
  File4.Pas
  File5.Plm
  File6.Asm

:Sources Sample3:
  File7.Pas
  File8.Plm
  File9.Asm
```

Now, if you select "Change source groups" from the Main menu, the form shown in Figure 2-4 is displayed:

| | |
|---|------------|
| No | Yes |
| Sample1..... | Yes |
| Sample2..... | Yes |
| Sample3..... | Yes |
| Source groups: Fill in form and confirm | |

Figure 2-4. The Change Source Groups Form

With all three groups set to "Yes" on this form, you would get a display similar to the screen shown in Figure 2-5 if you select "Compile" or "Edit source file" from the Main menu.

| | | |
|--|------------|----------------|
| No | Yes | Yes with Debug |
| File1.Pas..... | No | |
| File2.Pas..... | No | |
| File3.Pas..... | No | |
| File4.Ftn..... | No | |
| File5.Ftn..... | No | |
| File6.Ftn..... | No | |
| File7.PLM..... | No | |
| File8.PLM..... | No | |
| File9.ASM..... | No | |
| Compile source files: Fill in form and confirm | | |

Figure 2-5. The Compile Form with All Groups Enabled

If you enable only groups 1 and 3 via the Change Source Groups form, the Compile form would be as shown in Figure 2-6.

| | | |
|--|------------|----------------|
| No | Yes | Yes with Debug |
| File1.Pas..... | No | |
| File2.Pas..... | No | |
| File3.Pas..... | No | |
| File7.PLM..... | No | |
| File8.PLM..... | No | |
| File9.ASM..... | No | |
| Compile source files: Fill in form and confirm | | |

Figure 2-6. The Compile Form with Groups 1 and 3 Enabled

: Test :

Each GRiDDevelop data file can specify one or more Test tokens. Following the Test token, you can specify any number of command lines, each of which must be ended with RETURN. The following is an example of the use of the Test token:

```
:Test:
  GRiDPlan SampleData
```

Now, when you select Test from the Main menu, GRiDPlan would be invoked along with the file SampleData.

If more than one test command is needed, then multiple test tokens can be defined. The text that follows the keyword Test within the colons is displayed on the Test form. You can then select which of the test sequences is to be initiated. You can specify as many of these tokens as required in each development data file. The following example illustrates the use of multiple Test tokens:

```
:Test GRiDPlan:
  GRiDPlan SampleData

:Test GRiDPlot:
  GRiDPlot PlotTestData
```

Now, when you select Test from the Main menu, the Test menu shown below will be displayed:



You can then initiate the desired sequence by selecting it from the Test menu.

USER-DEFINED TOKENS

You can define your own GRiDDevelop tokens which can have any number of command lines associated with them. The tokens you define are placed as items on the GRiDDevelop commands menu. For example, if you have the following token in a development data file,

```
:GRiDManager:
  GRiDManager
```

the Commands menu displayed when you press CODE-? would be as shown

in Figure 2-7.

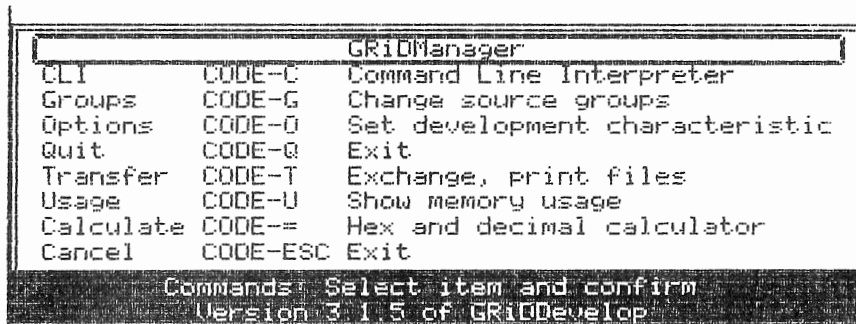


Figure 2-7. The Commands Menu with User-Defined Tokens

Now, you can invoke GRiDManager by selecting it from the Commands menu.

Command Modifier Characters

GRiDDevelop recognizes two characters that can modify the execution of command sequences that follow such tokens as Debug, Link and Enter.

If the first character in a command line is a question mark (?), then after the command is executed the system will pause and not proceed until you press a key on the keyboard.

If the first character in a command line is a semicolon (;), then the command that follows the semicolon is simply ignored.

THE GRiDDEVELOP COMMANDS MENU

The Commands menu appears when you press CODE-? and displays the items shown in Figure 2-7. (Remember, the first item in this figure, GRiDManager, is the user-defined token we used as an example.) Many of the items on the Commands menu are generally self-explanatory and should be familiar to you from other GRiD applications. The Command Line Interpreter (CODE-C), Change source groups (CODE-G), Options (CODE-O), Transfer (CODE-T), and Calculate (CODE-=) commands, however, deserve some additional discussion.

THE GRiDDEVELOP COMMAND LINE INTERPRETER (CODE-C)

GRiDDevelop provides a command line interpreter (CLI) that can be invoked either from the Commands menu or by pressing CODE-C. When

the CLI is invoked, it displays a field at the bottom of the screen that lets you enter text to be interpreted by the CLI. The text entered into this field is interpreted as though it was a command to the Development Executive. Refer to Appendix A for a discussion of the Development Executive program and to Appendix C for a description of the programs that can be invoked via the command line.

Results of the action initiated via the CLI are displayed above the CLI field. You temporarily halt scrolling of large amounts of data resulting from a command (for example, a CAT or DUMP) by pressing CTRL-S. Press CTRL-S a second time to resume scrolling of the display.

You can redisplay the immediately preceding command while in CLI mode by pressing CODE-D. To leave the CLI field and return to the GRiDDevelop Main menu, press ESC.

THE GRiDDEVELOP HEX AND DECIMAL CALCULATOR (CODE-=)

GRiDDevelop provides a hex and decimal calculator that can be invoked either by selecting it from the Commands menu or by pressing CODE-=. When the calculator is invoked, it displays a field at the bottom of the screen that lets you enter numeric data. The operations handled by the calculator are essentially the same as those defined as functions in GRiDPlan. Refer to the GRiD Management Tools reference manual for details. Any number that has a trailing "h" or "H" (for example, 1eh or 2FFh) is interpreted as a hexadecimal number.

You can redisplay the immediately preceding calculator entry by pressing CODE-D. To leave the calculator field and return to the GRiDDevelop Main menu, press ESC.

THE CHANGE SOURCE GROUPS COMMAND (CODE-G)

This command can be invoked either from the Commands menu or by pressing CODE-G and lets you specify which group or groups of source file names are to be displayed for selection. Refer to the :Sources: token description earlier in this chapter for details.

THE OPTIONS COMMAND

The Options command currently lets you specify only one option: whether to halt on errors. If you issue this command by selecting it from the Commands menu or by pressing CODE-T, the form shown below is displayed:

| | |
|-----------------------------------|----|
| Yes | No |
| Halt on errors Yes | |
| Options: Fill in form and confirm | |

If you specify "Yes" on this form, an error encountered while compiling will return you to the GRiDDevelop main menu after the module that produced the error has been compiled. This prevents compilation of subsequent modules that you may have specified via the Compile form. Only compile errors will cause a halt; compiler warnings do not prevent continuation.

If you specify "No" on this form, errors encountered during compilations are ignored and compilation of any other selected modules will proceed. Note that if you have selected files to be both compiled and then linked, the linkage will not be performed if any compile errors are encountered regardless of the setting of the Options form.

THE TRANSFER MENU

Figure 2-8 shows the items on the GRiDDevelop Transfer menu, which appears when you select the Transfer item from the Commands menu or when you press CODE-T.

| |
|-----------------------------------|
| Change this develop file |
| Exchange for another file |
| Read this develop file |
| Examine log file |
| Erase a file |
| Show characteristics of a file |
| Retrieve any file |
| Edit any file |
| Print any file |
| Print list file(s) |
| Print source file(s) |
| Transfer: Select item and confirm |

Figure 2-8. The GRiDDevelop Transfer Menu

The second, fifth, and sixth items on this menu ("Exchange for another file", "Erase a file", and "Show characteristics of a file") operate just as they do in other GRiD applications. They simply bring up a File form that you fill out specifying the file(s) to be acted on. The "Edit any file" and "Print any file" items also bring

up the standard File form to let you select files for editing with GRiDWrite or printing.

The first item ("Change this development file"), the third and fourth items ("Read this develop file" and "Examine log file"), the seventh item (Retrieve any file" and the last two items ("Print list file(s)" and "Print source file(s)") operate somewhat differently than in standard GRiD applications and are described in the paragraphs that follow.

Changing the Develop Data File

The first item on the Transfer menu, "Change this develop file", lets you edit the contents of the develop data file currently being used to drive the activities of GRiDDevelop. When you select this item, GRiDWrite is automatically invoked and the current development data is brought into memory. You can then edit the data to meet the changing characteristics of the development process. When you quit from this editing activity, you are returned to GRiDDevelop with the new contents of the data file now driving GRiDDevelop.

Reading the Develop Data File

If you select the third item on the Transfer menu, "Read this develop file", the contents of the current develop file are re-read by GRiDDevelop to ensure that the activities of GRiDDevelop are being directed by the most current version of the develop data file. This item is handy if you have been using GRiDWrite to edit a text file and have exchanged a standard GRiDWrite text file for a Develop data file. You would then select "Read this develop file" to cause GRiDDevelop to update all of its internal parameters to reflect the new contents of the develop file.

Examining the Log File

This item on the Transfer menu invokes either GRiDFile or GRiDWrite to let you inspect the contents of the file designated as the "Log file". Refer to the discussion of the Log File token earlier in this chapter for additional details.

Retrieving any file

This item on the Transfer menu lets you select another file for execution and allows a subsequent return to GRiDDevelop when the selected file has been exited. Thus, for example, if you select a worksheet file when the "Retrieve any file" message and File form are presented, GRiDPlan will be loaded into memory along with the specified worksheet file. When you subsequently quit or escape from

GRiDPlan, you are returned to GRiDDevelop's main menu.

Printing List and Source Files

The last two items on the Transfer menu, "Print list file(s)" and "Print source file(s)", let you select one or more of your list/source files for printing. For example, if you select "Print source file(s)" from the Transfer menu, the form shown in Figure 2-9 is displayed.

| No | Local | Remote |
|----------------|-------------------------------------|--------|
| File1.Pas..... | <input checked="" type="checkbox"/> | No |
| File2.Pas..... | <input type="checkbox"/> | No |
| File3.Pas..... | <input type="checkbox"/> | No |
| File4.Ftn..... | <input type="checkbox"/> | No |
| File5.Ftn..... | <input type="checkbox"/> | No |
| File6.Ftn..... | <input type="checkbox"/> | No |
| File7.PLM..... | <input type="checkbox"/> | No |
| File8.PLM..... | <input type="checkbox"/> | No |
| File9.ASM..... | <input type="checkbox"/> | No |

Print source file(s): Fill in form and confirm

Figure 2-9. The Print Source File(s) Form

This form displays all of source files in the currently enabled groups (see the :Sources: token for a discussion of groups) and lets you indicate which files or files are to be printed. When you confirm this form, the indicated files will be printed. For example, confirming the form shown in Figure 2-10 will cause source File2 and File3 to be printed to your locally attached printer, and File7 and File8 to be printed to the remote printer specified by the :Print to: token. If no :Print to: token has been specified, then the "Remote" choice will not be displayed.

| No | Local | Remote |
|----------------|-------------------------------------|-------------------------------------|
| File1.Pas..... | <input type="checkbox"/> | No |
| File2.Pas..... | <input checked="" type="checkbox"/> | Local |
| File3.Pas..... | <input checked="" type="checkbox"/> | Local |
| File7.PLM..... | <input type="checkbox"/> | Remote |
| File8.PLM..... | <input type="checkbox"/> | Remote |
| File9.ASM..... | <input type="checkbox"/> | <input checked="" type="checkbox"/> |

Print source file(s): Fill in form and confirm

Figure 2-10. Printing Selected Source Files

The "Print list file(s)" form works in exactly the same way as illustrated for "Print source file(s)".

CHAPTER 3. COMPILERS, LIBRARIES, AND INCLUDE FILES

This chapter describes the compilation procedures to follow to obtain an object file from a program's source file, and discusses the include files that you may need for your programs and library files that are available during linking.

COMPILING PROGRAMS

The compilers for Pascal-86, PL/M-86, FORTRAN-86 and Assembler-86 are described in the following Intel language reference manuals:

- PASCAL-86 User's Guide
- FORTTRAN-86 User's Guide
- PL/M-86 User's Guide
- Assembler-86 User's Guide

The descriptions of the compilers in these manuals are comprehensive but there are several considerations to observe when using them on the GRiD Compass system. These special considerations are discussed in the paragraphs that follow.

Compiler Size Controls

Most of the compilers provide size controls - LARGE, COMPACT, MEDIUM, SMALL. You must use either the compiler's COMPACT or LARGE control. If either a program or a block of data used with the program are larger than 64K, you must use the LARGE control; otherwise use the COMPACT control since this will result in smaller

programs.

Since LARGE is the default for the Pascal-86 compiler, you must specifically specify COMPACT if that is what you desire. The default for the PL/M-86 compiler is SMALL; therefore you must explicitly specify either the COMPACT or LARGE control when compiling PL/M programs. With FORTRAN-86, the only choice is the LARGE case; therefore, FORTRAN programs must be compiled with this control.

LIBRARIES

When you purchase a language, each of the compilers is provided on a diskette under the subject "Programs". Other files associated with each language are provided on the same diskette as the compiler and are listed in the paragraphs that follow.

NOTE: the language diskettes also contain language-specific include files under the subject Incs. These files will be discussed at the end of this chapter.

Pascal-86 Libraries and Modules

The file named Pascal~Run~ is under the Programs subject and contains the Pascal-86 compiler. The remaining files are under the Libs subject and contain the run-time support libraries and modules.

pascal~Run~ -- the compiler

| | |
|--------------|------------------------------------|
| p86rn0~Lib~ | |
| p86rn1~Lib~ | Libraries that must be linked with |
| p86rn2~Lib~ | the Pascal object module if you |
| p86rn3~Lib~ | use any of the Pascal I/O calls. |
| rtnull~Lib~ | |
| DqLarge~Lnk~ | |

If you use the input/output routines provided by GRiD-OS and the Common Code, then you should not use the Pascal I/O procedures READ, READLN, WRITE, and WRITELN, and you need not link in the Pascal run time libraries listed above. Instead, you need only link in the GRiD-supplied library file "LargeSystemCalls~Libs~" or "CompactSystemCalls~Libs~" (depending on whether you are using the LARGE or COMPACT size control when compiling).

NOTE: If you do not link in the Pascal runtime libraries then you must not make any calls to Pascal I/O statements. Also, in the PROGRAM declaration at the beginning of a Pascal program, do not specify the module as "Input, Output" nor any other file names or you will get link errors.

FORTRAN-86 Libraries and Modules

The file named `Fortran~Run~` is under the "Programs" subject and contains the FORTRAN-86 compiler. The remaining files are under the Libs subject and contain the run-time support libraries and modules. Unlike Pascal, when using Fortran you must always link in all of the Fortran run-time libraries listed below.

```
fortran~Run~    -- Fortran compiler

f86rn0~Lib~
f86rn1~Lib~
f86rn2~Lib~      Run-time Libraries
f86rn3~Lib~
f86rn4~Lib~
DqLarge~Lnk~
```

PL/M-86 Libraries and Modules

The file named `plm~Run~` is under the Programs subject and contains the PL/M-86 compiler. The other file contains the run-time support libraries and is under the subject Libs.

```
plm~Run~
plm~Lib~
DqLarge~Lnk~
```

8087 Libraries and Modules

These files contain the run-time support libraries and modules required by the 8087 Numeric Data Processor and must be included if the program being compiled uses the 8087 and if you do not link in the GRiD-supplied library `CompactSystemCalls` or `LargeSystemCalls`. To determine if your program uses the 8087, refer to the appropriate Intel language manual. If you are not certain, try linking without these libraries. If you get unresolved symbols, then then go ahead and link in the 8087 libraries one at a time.

```
8087~Lib~
87null~Lib~
cel87~Lib~
dcon87~Lib~
eh87~Lib~
```

INVOKING THE COMPILERS

The compilers can be invoked automatically from `GRiDDevelop` if you append the appropriate language identification suffix to the source file name (`.Pas`, `.Plm`, `.Ftn`, `.Asm`). (See Chapter 2 for details.) `GRiDDevelop` also lets you specify any compiler controls you require

via the GRiDDevelop data file described in Chapter 2.

If you do not use GRiDDevelop, you can invoke the compilers from a command line (see Appendix A) by simply entering the compiler's name, for example plm or pascal, followed by the source program name and any compiler controls. (Refer to the appropriate Intel language manual for a description of compiler controls usage.) For example:

```
plm MyProgram.Plm~Text~ LARGE DEBUG
```

```
pascal MyProgram.Pas~Text~ NOLIST
```

```
fortran Myprogram.Ftn~Text~ XREF
```

THE SYSTEM AND LANGUAGE INCLUDE FILES

The language compilers provide an INCLUDE control that let you include other source modules for compilation with your program. (Refer to the appropriate Intel language manual for a description of INCLUDE). The include files provided by GRiD are simply a text insertion mechanism: they let you use the declarations of GRiD-developed procedures and functions within your programs without having to laboriously type all of them into your source file.

There are several files that must be included during compilation of your source programs if the program makes any direct, explicit calls to the GRiD-OS. As you develop your own programs you will probably develop your own groups of include files.

Two sets of include files are currently provided on the language diskettes under the subject Incs: one for Pascal programs and one for PL/M programs.

Pascal Include Files

```
Common.Inc~Text~  
OsPasTypes.inc~Text~  
OsPasProcs.inc~Text~
```

```
ConPas.inc~Text~
```

PL/M Include Files

```
PLMLits.inc~Text~  
OsPlmTypes.inc~Text~  
OsPlmProcs.inc~Text~
```

```
ConPlm.inc~text~
```

The Common.Inc~Text~ and PLMLits.inc~text~ files contain some standard declarations used in the Pascal-86 and PL/M-86 languages and should always be included. The OsPasTypes.inc~Text~ and OsPlmTypes.inc~Text~ files contain declarations of data types needed if explicit GRiD-OS calls are made. The OsPasProcs.inc~Text~ and OsPlmProcs.inc~Text~ files contain the definitions of functions and procedures comprising the GRiD-OS calls. The include files above should be included in the order in which we have listed them to avoid undefined symbol errors.

Many more include files are used to define the functions and procedures available in Common Code. Refer to the Common Code Reference manual for information on other available include files.

Examples of Include Control Statements

The following examples illustrate the format of typical INCLUDE controls for the compilers as they would be stated within your ^Text^ source file.

NOTE: The dollar sign (\$) must be in column 1 of your source file to be recognized by the compilers.

Pascal-86 Example:

```
$INCLUDE ('w\incs\Common.Inc^Text^')
$INCLUDE ('w\incs\ConPas.Inc^Text^')
$INCLUDE ('w\incs\OsPasTypes.Inc^Text^')
$INCLUDE ('w\incs\OsPasProcs.Inc^Text^')
```

PL/M-86 Example:

```
$INCLUDE ('w\incs\PlmLits.Inc^Text^')
$INCLUDE ('w\incs\ConPlm.Inc^Text^')
$INCLUDE ('w\incs\OsPlmTypes.Inc^Text^')
$INCLUDE ('w\incs\OsPlmProcs.Inc^Text^')
```


CHAPTER 4. THE LINK PROGRAM

The Link program combines relocatable object modules produced by the language compilers and resolves references between independently compiled modules. The input to the Link program is a list of files and optional controls; the output is a single object file and (optionally) a print file.

The Link program is thoroughly described in the Intel manual "iAPX 86,88 Family Utilities User's Guide" which is supplied with development systems. Refer to this manual for a complete description of Link including descriptions of some potentially useful controls that are not covered in this chapter. This manual also describes the Librarian (Lib) and CREF (cross-reference listing generator) programs that are supplied with development systems.

INVOKING THE LINK PROGRAM

The general syntax of a link invocation is

```
LINK inputList TO outputFile~Run~ BIND SS(STACK(+nnnn))
{controls}
```

Where:

| | |
|------------|--|
| inputList | contains the filenames of object modules and libraries. |
| outputFile | the filename that is to receive the linked output module that the Link program produces. |

| | |
|------------------|---|
| BIND | is a control that must always be specified in the final link of a program to obtain a load time locatable module. |
| SS(STACK(+nnnn)) | is a control that must be specified to obtain a sufficiently large stack Segment Size (SS) for program execution. |
| controls | are the optional controls (summarized in Table 4-1) that modify the standard operation of the Link program. |

Each pathname in the inputList is separated from the preceding file name by a comma and the last pathname in the list is separated from TO by a space. For example:

```
LINK 'w'MySystem'MyFile1~Obj~, 'w'MySystem'MyFile2~Obj~,
~'w'Libs'CompactSystemCalls~Lib~ TO 'w'MySystem'NewFile~Run~
BIND SS(STACK(+1500))
```

The pathname of the output file is separated from TO by a space, and any controls you specify are separated from each other by a space.

LINK INVOKATION EXAMPLES

The following example takes the Pascal object module named MyFile.Pas~Obj~, links it together with several of the Pascal and 8087 library modules located under the subject 'w'libs and produces a linked and bound output module named MyFile~Run~.

```
LINK MyFile.Pas~Obj~, 'w'libs'P86RN0~Lib~, 'w'libs'P86RN1~Lib~,
'w'libs'P86RN2~Lib~, 'w'libs'P86RN3~Lib~, 'w'libs'CEL87~Lib~,
'w'libs'EH87~Lib~, 'w'libs'8087~Lib~, 'w'libs'DCON87~Lib~,
'w'libs'DqLarge~Lnk~ TO MyFile~Run~ BIND SS(STACK(+1500))
PC(PURGE)
```

NOTE: You can put this link invocation sequence into a GRiDDevelop data file and then initiate the link operation from the GRiDDevelop menu. See Chapter 2 for examples. If you do not use GRiDDevelop, you can create a command (~Com~) file and then initiate the link with the Do program. See Appendix A for examples of command files.

If you do not use any Pascal input/output procedures then you need not link in Pascal run-time libraries (P86RN0-P86RN3) nor the 'w'libs'DqLarge~Lnk~ file: instead, simply link in the file 'w'Libs'CompactSystemCalls~Lib~ or 'w'Libs'LargeSystemCalls~Lib~ to obtain the GRiD I/O procedures. In this case, the link invocation sequence would be:

```
LINK MyFile.Pas~Obj~, 'w'libs'CEL87~Lib~, 'w'libs'EH87~Lib~,
~'w'Libs'CompactSystemCalls~Lib~ TO MyFile~Run~
```

```
'w'libs'8087~Lib~, 'w'libs'DCON87~Lib~,
'w'libs'CompactSystemCalls~Lib~ TO MyFile~Run~ BIND
SS(STACK(+1500)) PC(PURGE)
```

The following example shows the link commands for a FORTRAN program:

```
LINK MyFile.Ftn~Obj~, 'w'libs'F86RN0~Lib~, 'w'libs'F86RN1~Lib~,
'w'libs'F86RN2~Lib~, 'w'libs'F86RN3~Lib~, 'w'libs'F86RN4~Lib~,
'w'libs'CEL87~Lib~, 'w'libs'EH87~Lib~, 'w'libs'8087~Lib~,
'w'libs'DCON87~Lib~, 'w'libs'DqLarge~Lnk~ TO MyFile~Run~ BIND
SS(STACK(+1500)) PC(PURGE)
```

LINK PROGRAM CONTROLS SUMMARY

Table 4-1 summarizes the controls available with the Link program that are described in this chapter and shows the default setting for each control.

Table 4-1. Summary of Link Controls

| Control | Abbrev. | Default |
|---------------------------|-----------|-----------|
| ASSUMEROOT(pathName) | AR | - |
| BIND ! NOBIND | BI ! NOBI | NOBIND |
| FASTLOAD ! NOFASTLOAD | FL ! NOFL | NOFL |
| MAP ! NOMAP | MA ! NOMA | MAP |
| NAME | NA | -- |
| OVERLAY ! NOOVERLAY | OV ! NOOV | NOOVERLAY |
| PRINT(pathName) ! NOPRINT | PR ! NOPR | PRINT |
| PRINTCONTROLS(PURGE) | PC | |
| PURGE ! NOPURGE | PU ! NOPU | NOPURGE |
| SEGSIZE (STACK(+nnnn)) | SS | -- |

ASSUMEROOT

| Syntax | Abbreviation | Default |
|----------------------|--------------|---------|
| ASSUMEROOT(pathName) | AR | -- |

Definition

ASSUMEROOT is used only in conjunction with the OVERLAY control and suppresses the inclusion of any library module(s) in an overlay if those modules have already been included in the root file identified by pathName. ASSUMEROOT causes the root file to be scanned, and all external, undefined symbols in the overlay modules which have a matching definition in the root file are marked "temporarily resolved." This marking means that while a library search for the symbols will not be made, their status remains externally undefined until the overlays are linked with the root. See Appendix B for examples of the use of ASSUMEROOT.

BIND : NOBIND

| Syntax | Abbreviation | Default |
|--------|--------------|---------|
| BIND | BI | NOBIND |
| NOBIND | NOBI | |

Definition

BIND combines the input modules into a load-time-locatable module that can then be loaded and executed. Since the default for this control is NOBIND, you must always explicitly specify the BIND control during the final link to obtain a module that can be loaded and executed under GRiD-OS.

FASTLOAD : NOFASTLOAD

| Syntax | Abbreviation | Default |
|------------|--------------|------------|
| FASTLOAD | FL | NOFASTLOAD |
| NOFASTLOAD | NOFL | |

Definition

FASTLOAD reduces program loading time and also produces the most compact object file. Loading time is reduced by concatenating data records to a maximum length of 64K. The object file size is reduced by removing such information as local symbols, public records, comments, and type information (unless the object file contains unresolved external symbols). To obtain an executable object file of the smallest size, use the both the FASTLOAD and PURGE link controls.

The FASTLOAD control should not be used if you are going to be debugging the program.

MAP : NOMAP

| Syntax | Abbreviation | Default |
|--------|--------------|---------|
| MAP | MA | MAP |
| NOMAP | NOMA | |

Definition

MAP produces a link map and inserts it in the PRINT file (~MPI~) that is generated by the Link program. (The PRINT file is described at the end of this chapter.) The link map contains information about the attributes of logical segments in the output module. This includes size, class, alignment attributes and overlay name (if the segment is a member of an overlay). If you specify NOMAP, the PRINT file will not include a link map.

NAME

| Syntax | Abbreviation | Default |
|------------------|--------------|------------------------------|
| NAME(moduleName) | NA | Module keeps current name |

Definition

NAME assigns the specified moduleName to the output module's header record. If you do not use the NAME control, the output module will have the name of the first module in the input list. Note that NAME does not affect the output file's name; only the module name in the output module's header record is changed.

The moduleName may be up to 40 characters long and may be composed of any of the following characters in any order: question mark (?), commercial at (@), colon (:), period (.), underscore (_), A,B,C,...Z, or 0,1,2,...9. Lower case letters may be used, but they are automatically converted to uppercase by the Link program.

OVERLAY : NOOVERLAY

| Syntax | Abbreviation | Default |
|------------------------|--------------|-----------|
| OVERLAY{(overlayName)} | OV | NOOVERLAY |
| NOOVERLAY | NOOV | |

Definition

OVERLAY specifies that all of the input modules shall be combined into a single overlay module. If you specify the optional overlayName argument, all segments contained within the overlay module have that name in addition to their segment names and class names. If no overlayName is specified, the Link program uses the module name of the first module in the input list as the overlayName.

You must link each overlay in a program separately before you link all the overlays into a single object module. When linking root and overlay files, the Link program assumes that the first file in the invocation line is the root. When you call the operating system to load the overlay, you must use the same overlay name that you specified (overlayName) with this OVERLAY control. See Appendix B for a complete description of overlays.

PRINT : NOPRINT

| Syntax | Abbreviation | Default |
|-------------------|--------------|------------------------|
| PRINT((pathName)) | PR | PRINT(objectFile~MP1~) |
| NOPRINT | NOPR | |

Definition

PRINT lets you specify the pathname for the PRINT file created by the Link program. (The PRINT file is described at the end of this chapter.) If the PRINT control is not specified or if the control is given without the pathName argument, the print file will have the same pathname as the object file output by the Link program except the Kind extension will be ~MP1~ instead of ~RUN~. NOPRINT prevents the Link program from creating any print file.

PRINTCONTROLS(PURGE)

| Syntax | Abbreviation | Default |
|----------------------|--------------|---------|
| PRINTCONTROLS(PURGE) | PC(PU) | -- |

Definition

PRINTCONTROLS(PURGE) removes all information about the debug or public records from the print file (MP1) produced by the Link program and thus significantly reduces the size of that file.

PURGE

| Syntax | Abbreviation | Default |
|---------|--------------|---------|
| PURGE | PU | NOPURGE |
| NOPURGE | NOPU | |

Definition

PURGE removes all of the debug or public records from the object file and their information from the print file. If you specify both the FASTLOAD and PURGE controls, you will obtain the most compact output object file possible. The records that would be included by NOPURGE and NOFASTLOAD are useful when debugging programs, but are unnecessary for producing executable code.

SEGSIZE

| Syntax | Abbreviation | Default |
|-----------------------|------------------|---------|
| SEGSIZE(STACK(+nnnn)) | SS(STACK(+nnnn)) | -- |

Definition

SEGSIZE(STACK(+nnnn)) specifies the amount of additional memory space needed for the stack segment. The compilers automatically determine how much stack a program needs. If your program did not call any common code or GRiD-OS routines directly and has no re-entrant procedures, the compilers will generate the correct size stack. However, if you do call common code or GRiD-OS routines, they also use your stack and you must increase the size of the stack accordingly. There are no hard and fast rules for the amount of stack you will need. A good first approximation is +1500. If you have a program which crashes in unexpected ways, the first thing you should try is to increase the stack size further.

NOTE: If you omit the plus sign from the size specification, it is treated as the absolute size of the stack segment and could cause failure from an insufficient stack.

THE LINKER'S PRINT FILE

The Link program always creates a print file unless you specify NOPRINT. The optional pathName argument to the PRINT control designates the name of the print file. The default name is the name of the output object file but with a Kind extension of ~MPI~.

The print file may contain as many as five parts:

- o A header (always present)
- o A link map (unless NOMAP specified)
- o A group map (always present)
- o A symbol table (unless PURGE or PC(PURGE) specified)
- o An error message list (always included when errors occur)

Most of the information contained in the print file is used for diagnostic purposes when constructing such things as system loaders and will be of little or no interest to most programmers. The only parts of the print file that may be of general interest and use are the unresolved symbols list which is part of the link map and the error message list at the end of the print file.

The unresolved symbol list itemizes each external symbol whose public definition was not encountered. The module that references the unresolved symbol is also indicated. The printed message that appears under the heading UNRESOLVED EXTERNAL NAMES is as follows:

```
symbolName IN pathName(moduleName)
```

Warning messages are listed consecutively as warning situations are encountered. They may appear before or after the link map.

Errors always terminate processing - an error message will always be the last line in the print file. For a discussion of the interpretation of individual messages, refer to Appendix D.

CHAPTER 5: THE DEBUGGER

The debugger program (Debug) is a symbolic, interactive, multitasking debugger for high-level languages. It lets you debug programs at the source level by examining a program as it executes. Debug lets you:

- o Set breakpoints in the program so you can check the progress of program execution at any point. You can set breakpoints at a line number, at a procedure beginning or end, upon return from a procedure, or at a memory location. Timing breakpoints can also be set.
- o Display/examine the contents of variables, memory locations, and registers.
- o Change the contents of registers, memory locations, and variables.
- o Dump the contents of memory in both hexadecimal and ASCII formats.
- o Check the status of system multitasking operations by displaying information concerning processes, semaphores, and messages.
- o Set timing breakpoints.
- o Alternate between two screens, one for the debugger and the one being used by your program.

Some of the debugger commands are invoked by pressing CODE and one other key, while others are invoked via a command line. The commands are listed in Table 5-1 and will be described in alphabetical order in the pages that follow. The CODE-key commands are described first and then the command line commands.

ESC cancels command entry.
 Command syntax:
 e<name> Display address
 <name> l <absMem> Display value
 <name> = <expression> Assign value
 <absMem> = Dump/change memory
 <name> l <absMem> <len> Dump <len> bytes

| CODE KEY | | CODE KEY | |
|----------|--------------------------|----------|---------------------------|
| B | set Breakpoint | S | display Stack Frames |
| C | Clear breakpoint | T | display Tasks, senaphores |
| D | Duplicate previous entry | U | Unassemble |
| E | get developmentExecutive | V | display Version number |
| F | Fill memory | W | toggle Windows |
| I | display Info, options | 0 | show procedure info |
| L | display Local variables | 1 | set timing range |
| M | display tasks Messages | 2 | clear timing range |
| O | set/change Options | 3 | show timing ranges |
| P | Proceed with execution | 4 | print timing results |
| Q | Quit the debugger | 5 | reset timing results |
| R | display Registers | ESC | cancel the debugger |

✖

Table 5-1. Debugger Command Summary

COMPILE AND LINK CONSIDERATIONS

In order to view symbol names and line numbers, the program to be debugged must be compiled with the \$DEBUG option specified, and the PURGE and FASTLOAD controls in the Link program must not be specified (NOPURGE and NOFASTLOAD are the defaults for these controls).

INVOKING THE DEBUGGER

The Debug program can be invoked by issuing the following command via the command line interpreter of the Development Executive program (described in Appendix A):

```
DEBUG programName [parameters]
```

Note: You can also invoke the Debug program from GRiDDevelop. See Chapter 2 for details.

The optional parameters are those that might be needed by the program being debugged.

When it is invoked, the debugger creates a set of debugging files (ZZZDEBUG.MOD, ZZZDEBUG.PUB, ZZZDEBUG.SYM, ZZZDEBUG.TYP, and ZZZDEBUG.WIN). These are information files used by the debug program and require approximately the same amount of disk space as the program module occupies.

After creating these files, the debugger displays its prompt character, an asterisk (*). You can now issue the debugger commands.

DEBUGGER SYNTAX AND TERMINOLOGY

The following syntax conventions and abbreviations are used throughout this chapter.

decimalConstant -- a number composed of the digits 0 through 9.

hexConstant -- a decimal digit (0-9) followed by any combination of hex digits (0-9,A,B,C,D,E,F) and ending with the character "H". For example, 0FFE8h.

absMem -- an absolute memory address with the segment value followed by a colon and ending with the offset value. For example, 0001h:0FFEh or #AX:#IP (see register notation below).

register -- the number or pounds character (#) followed by the standard Intel symbol for 8086 registers. For example, #AX and #SP.

number -- a decimal constant, hex constant, register reference, or an equation composed of these simple math operations (+,-,*,/) and unary plus and minus. Note that equation operators are evaluated from left to right -- there is no operator precedence.

line# -- any number that has a corresponding statement number in the source code listing.

varName -- the name of a variable in the program being debugged.

procName -- the name of a procedure in the program being debugged.

module -- the name of any module in the program being debugged.

If the varName, procName, or line# you want to refer to is in the current module being debugged, you need not precede the name with the module name. If you want to refer to a name or line# in another module, explicit module references can be made by preceding a varName, procName, or line# with the module name and a colon (:). For example:

NEWMOD:BESTPROC

References to a varName or procName that is in the public domain are made by prefixing the name with a colon (:). For example:

CODE-KEY COMMANDS

Many of the debugger commands can be initiated by simply pressing one of the standard keyboard keys while holding down the CODE key. The paragraphs that follow describe these commands in alphabetical order.

THE HELP (CODE-?) COMMAND

This command displays a brief summary of the debugger commands -- both the CODE-key and command line commands. When you press CODE-? you will get the display shown in Table 5-1 appears.

THE SET BREAKPOINT (CODE-B) COMMAND

This command lets you set breakpoints in your program so you can monitor program execution. You can specify the breakpoints by line number, absolute memory address, or by procedure name. You can also specify that a break occur after the breakpoint has been reached a certain number of times.

When you press CODE-B, the debugger prompts you with the following message:

Set Breakpoint At: [previousBreakpoint]

If you have previously set breakpoints, the most recently established breakpoint will be displayed after the prompt message. If this is the first breakpoint that you are setting, the field following the prompt message will be blank. If you want to set a new breakpoint, backspace to erase the displayed previous breakpoint. You can then enter an absolute memory address, line number, or procedure name and press CODE-RETURN.

If you enter a procedure name, you must specify whether the break should be at the beginning or end of the procedure, or upon return from the named procedure. Debug will will prompt you with:

Begin/End/Return: B

The supplied default choice is B (for break at Beginning of procedure). If you want one of the other choices, backspace and then enter E (for break at End of procedure), or R (for break on Return from procedure). Then press CODE-RETURN.

You will then be prompted with:

Break After Count:1

Note that the debugger supplies a default value of 1, indicating that the break should be made the first time this breakpoint is encountered. If you want some other value, simply backspace to erase the "1" and enter the value you want. Then press CODE-RETURN. The debugger will output the following message:

Entered as Break Table Entry #n

The debugger maintains a table that defines the characteristics of each breakpoint you specify. It sequentially numbers each breakpoint (beginning with 0) as you specify them. You can examine this table using the CODE-I command.

NOTE: A random breakpoint can be caused at any time by pressing CODE-SHIFT-ESC.

THE CLEAR BREAKPOINT (CODE-C) COMMAND

This command clears a breakpoint that you have previously set (using the CODE-B command). The number of the breakpoint is the number assigned to it by the debugger when you set the breakpoint. (You can check this number using CODE-I). If you enter an asterisk (*) instead of a number, all breakpoints will be cleared.

TIMING BREAKPOINT COMMANDS (CODE-1 -- CODE-5)

These five commands let you set, clear and examine timing breakpoints to determine such timing factors as the time spent within specified procedures or line number ranges. The five timing breakpoint commands are as follows:

| | |
|--------|----------------------|
| CODE-1 | set timing range |
| CODE-2 | clear timing range |
| CODE-3 | show timing ranges |
| CODE-4 | print timing results |
| CODE-5 | reset timing results |

When you press CODE-1, the debugger prompts you with the following message:

Start timing At: [previousBreakpoint]

If you have previously set timing breakpoints, the most recently established timing breakpoint will be displayed after the prompt message. If this is the first timing breakpoint that you are setting, the field following the prompt message will be blank. If you want to set a new timing breakpoint, backspace to erase the displayed previous breakpoint. You can then enter an absolute memory address, line number, or procedure name and press CODE-RETURN.

If you enter a procedure name, you must specify whether the break should be at the beginning or end of the procedure named procedure. Debug will prompt you with:

Begin/End: B

The supplied default choice is B (for break at Beginning of procedure). If you want to break at the end of a procedure, enter E (for break at End of procedure) Then press CODE-RETURN.

You will then be prompted with:

Stop timing At: [previousBreakpoint]

The syntax here is the same as for the "Start timing At:" dialog. Once again, you can specify that timing stop at either the beginning or end of a procedure (the default choice is "End". Then press CODE-RETURN. The debugger will output the following message:

Entered as entry #n

The debugger maintains a table that defines the characteristics of each timing breakpoint you specify. It sequentially numbers each breakpoint (beginning with 1) as you specify them. You can examine this table using the CODE-3 command. Up to seven different timing breakpoints can be set.

The results obtained from the timing breakpoints can be displayed using the CODE-4 command. The following example display shows the setting of two timing breakpoints (using the CODE-1 command) and the format of the display obtained using the CODE-4 command:

```
*
Start timing At: erasetablecells
Begin/End: B
Stop timing At: erasetablecells
Begin/End: E
Entered as entry # 1
*
Start timing At: duplicatetablecells
Begin/End: B
Stop timing At: duplicatetablecells
Begin/End: E
Entered as entry # 2
*
Break on NMI interrupt
*
#    %    Microseconds    Ticks    Start/Stop
0    39    10214482    25536206    3/3
1    3     403356    1008390    1/1
2    7     618878    2047196    2/2
*
```

The leftmost column is the timing breakpoint number assigned by the debugger when the breakpoints were specified. Note that number 0, is

assigned by the system. The value for this timing range is calculated beginning with the completion of the first user-specified timing range. For example, the timer for "0" in the display shown above would be started when the end of the EraseTableCells procedure was reached. Thereafter, timer "0" runs whenever the system is operating outside of any user-specified timing range.

The timers for specified ranges run only when the program is operated within the specified begin/end range. Each time the program re-enters a specified timing range, the appropriate timer resumes. The right-hand column of the CODE-4 display indicates the number of passes through the start/stop point of each range and can be used to divide the time in microseconds or ticks to obtain average duration of each timing range.

THE DUPLICATE LINE (CODE-D) COMMAND

This command causes the last line of text entered via the keyboard to be displayed again.

THE EXECUTIVE (CODE-E) COMMAND

This command causes control to be passed to the DevelopmentExecutive interface. The prompt character for this interface (->) will immediately be displayed and you can then use any of the system utilities that are described in Appendix C. To return to Debug from the Development interface, press CODE-Q and then CODE-RETURN; the Debug prompt character (*) will be displayed. The state of your debugging session will remain unchanged.

THE FILL MEMORY (CODE-F) COMMAND

This command lets you fill a specified memory range with a particular bit pattern. After pressing CODE-F you will be prompted for the starting memory address, the number of bytes to be filled, and the pattern byte to be used.

THE INFO (CODE-I) COMMAND

This command displays all the breakpoints set in the break table and also displays the current configurations of the options and system memory utilization as shown in the following example:

| # | Count | Occur | B/E | Break Location |
|---|-------|-------|-----|--------------------|
| 1 | 0 | 1 | | 0000h:0FFFEh |
| 2 | 2 | 3 | | :ReadHex:185 |
| 3 | 1 | 5 | B | Main:Factorial:100 |
| 4 | 0 | 1 | R | :PutChar |

Default Module Name: CurMod
Alternate Window: Y
Memory Available (Bytes): nnnnnn
Memory Allocated (Bytes): nnnnnn

THE LOCATION DISPLAY (CODE-L) COMMAND

This command displays the current location within the program being debugged. The format of the information displayed depends on the type of data available to the debugger: it may consist of just a memory address, or it may be a statement number, procedure name, variable name or some combination of these.

THE MESSAGE DISPLAY (CODE-M) COMMAND

This command displays the current messages (if any) of the current (running) process. The list includes the sending process ID, the message class, the note (if any) and the address for each message.

THE OPTIONS (CODE-O) COMMAND

This command lets you change the default module name, specify whether the debug dialog should be echoed to the printer, and change the alternate window choice to disk (if additional memory space is needed), RAM (the default choice that provides best performance), or none (debug and the module being debugged share the same window).

THE PROCEED (CODE-P) COMMAND

This command simply allows program execution to proceed or continue. Execution will stop either when a breakpoint is reached, when CODE-SHIFT-ESC is pressed, or when the program completes execution. You can also programmatically provide breaks by breaking on zero overflow, out of range, and so on by using the appropriate compiler controls (such

as CHECK on the Pascal-86 compiler). NOTE: you cannot use CODE-P to restart a program that has completed execution -- you must reinvoke the debugger and start from the beginning since control normally returns to the executive program at this point.

THE QUIT (CODE-Q) COMMAND

This command is used to exit from the debugger. Before the exit actually occurs, you will be asked to confirm that you want to quit.

THE REGISTER DISPLAY (CODE-R) COMMAND

This command displays the contents of all the 8086 registers in the following format:

| | | | | | | |
|-------|-------|-------------|-------------|-------|--------|-------|
| AX | BX | CX | DX | BP | SI | DI |
| nnnnh | nnnnh | nnnnh | nnnnh | nnnnh | nnnnh | nnnnh |
| DS | ES | SS:SP | CS:IP | FLAGS | ODITSZ | A P C |
| nnnnh | nnnnh | nnnnh:nnnnh | nnnnh:nnnnh | nnnnh | xxxxxx | x x x |

Following the "FLAGS" section at the end of the second line of the registers display is the setting of the individual flags within the Flags register. The upper line (ODITSZ A P C) indicates the particular flag (described below) and the line below indicates the state of each individual flag. Thus, you do not have to decode the hexadecimal representation of the Flags register to determine the setting of each flag.

- O - Overflow
- D - Direction (Forward/Reverse)
- I - Interrupt (Enabled/Disabled)
- T - Trap (Set/Cleared)
- S - Sign (Positive/Negative)
- Z - Zero (Zero/Non-zero)
- A - Auxiliary carry
- P - Parity (Odd/Even)
- C - Carry

THE STACK TRACE (CODE-S) COMMAND

This command displays the contents of the stack as it is being utilized by the program being debugged. The format of the display is shown in the following example:

Stack Trace:

```
1: [552913ACh] TABLEWRITE:WRITETAB line 456
2: [55291384h] TABLEWRITE:WRITETABLETOFILE line 446
3: [55290A46h] TABLEMENUFORM:WRITETHISFILE line 777
4: [55290736h] TABLEMENUFORM:TRANSFERMENU line 600
5: [55290205h] TABLEMAIN:TABLEMAIN line 626
```

The output shown above indicates nesting five levels deep with the topmost level being the TABLEMAIN procedure in the TABLEMAIN module. The current procedure is WRITETAB at line number 456 in the TABLEWRITE module. The numbers in brackets are the value of the CS:IP registers.

THE TASKS/SEMAPHORE DISPLAY (CODE-T) COMMAND

This command displays a list of all processes on the process queue and all semaphores on the semaphore queue. The format of the display is shown in the following example:

| ProcID | State | Pri | Pid/Sem | TimeLmt | #Msgs | MemUsed |
|--------|---------|-----|---------|---------|-------|---------|
| d 3561 | semWait | 1 | 3508 | 0 | 0 | 6656 |
| d 3632 | msgwait | 1 | 65535 | 0 | 0 | 512 |
| d 3685 | ready | 1 | 0 | 0 | 0 | 43856 |
| d 3799 | msgWait | 1 | 3787 | 0 | 0 | 512 |
| 3787 | running | 128 | 0 | 0 | 0 | 2896 |
| d 3489 | ready | 200 | 3513 | 0 | 0 | 64 |
| d 3549 | loadPkg | 255 | 0 | 0 | 0 | 20576 |

| SemaID | Count | Note | Busy | Creator |
|--------|-------|------|-------|---------|
| 3527 | 0 | 0 | 0 | 3685 |
| 3503 | 1 | 0 | 3578 | 3489 |
| 3501 | 0 | 0 | 65535 | 3489 |

If a line is preceded by the letter "d", it means that the process on that line is a process that is being debugged. The ProcID column, gives the process identification number assigned by GRiD-OS when the process

was created. The State column gives the current state of each process. The possible states are the following: running, ready, message wait, semaphore wait, timed wait, timed message wait, timed semaphore wait, or a loaded package (such as common). The Pri column gives the current priority of each process. The Pid/Sem column lists either the process being waited on (if in a message wait) or the semaphore being waited on (if in a semaphore wait). The TimeLmt column indicates the time remaining to wait on a timed semaphore or timed message wait. The #Msgs column gives the number of messages on the message queue of each process. The MemUsed column lists the bytes of memory used by each process.

The semaphore information is listed after the task information. The SemaID is the identification number assigned to the semaphore by GRiD-OS. The Count column lists the number of processes waiting for each semaphore. The Note column gives the note (if any) that was included with each semaphore. The Busy column lists the most recent process waiting on each semaphore. The Creator column gives the identification number of the process that created each semaphore.

THE UNASSEMBLE (CODE-U) COMMAND

This command disassembles specified sections of your source code and displays an assembly language listing on the screen. If you press CODE-U while the debuggers prompt character (*) is displayed but while there is nothing entered on the command line, disassembly begins at the current value of the CS:IP registers. You can cause disassembly to begin at a particular memory address or line number by typing that address or number on the command line and then pressing CODE-U instead of CODE-RETURN. Disassembly continues to the end of the module or until you press ESC. To temporarily halt scrolling of data, press CTRL-S. To resume display, press CTRL-S a second time.

You can also disassembly a single procedure or function by typing the procedure/function name on the command line and the pressing CODE-U. In this case, disassembly stops when the end of the procedure is reached.

THE WINDOW TOGGLE (CODE-W) COMMAND

This command toggles or switches you back and forth between the debugger window or screen and the application window.

COMMAND LINE COMMANDS

The commands described in the paragraphs that follow are initiated by entering text via a command line in response to the prompt (*) character. These commands let you display the addresses and contents of various program locations, assign values to registers, memory locations and program locations, and dump the contents of memory.

THE DISPLAY ADDRESS COMMAND

To display the address of a variable, procedure, line number or memory location, type "@" followed by the varName, procName, line#, or absMem. The debugger will display an equal sign (=) followed by the address in the format "segment:offset". For example:

```
@189 = 07AFh:02A6h (1967:678)
```

Note that the address is displayed in both hex and decimal formats.

THE DISPLAY CONTENTS COMMAND

To display the contents of a variable or memory location, type the varName or absMem location followed by CODE-RETURN. The debugger will display an equal sign (=) followed by the value of the specified item.

Variables are displayed according to the format they were declared in. For example, assume that you have declared a variable named 'a' as follows:

```
a : ARRAY [1..10] OF Char;
```

You could display all characters in the array by typing the variable name (a) and pressing CODE-RETURN or you could display the fifth element in the array by typing:

```
a[5]
```

NOTE: You can terminate the display of long variable structures by pressing ESC.

Local variables can only be displayed if you have broken within the procedure where the local variables are defined.

The contents of specified memory locations (absMem) are displayed as byte values.

You can also display the contents of a variable at one memory location as though it were of the type of another variable. The syntax for this is:

```
varName1 AS varName2
```

This would cause the contents of varName1 to be displayed using the type associated with varName2.

THE ASSIGN VALUE COMMAND

To assign a value to a variable or memory location, type `varName` or `absMem` followed by an equal sign and the value to be assigned. The value assigned will be echoed back and displayed. For example:

```
*1935:678=7Bh
Value Assigned = 123 (7Bh, "(", true)
```

Note that the value echoed back from a memory location is displayed in decimal, hex, ASCII interpretation (if printable), and boolean value.

You can assign values to any simple type variable except Reals. If the value you assign is larger than the value type for `varName`, the value is truncated to the appropriate size. Values assigned to memory locations are assumed to be of type Byte.

THE LOAD REGISTER COMMAND

To load a value into one of the 8086 registers, type `"#"` followed by the register name (AX, BX, IP, etc.), an equal sign and the value to be assigned. The value assigned will be echoed back and displayed. For example:

```
*AX=270Fh
Value Assigned = 9999 (270Fh)
```

Note that the value echoed back from a register is displayed in decimal and hex.

THE MEMORY DUMP COMMAND

To display the contents of a section of memory, type the variable name, procedure name, line number or memory address indicating the starting location where the dump is to begin. Then type one space and a number (`byteCount`) indicating the number of bytes to be dumped. The memory contents will be displayed in tabular form with 8 values per line beginning on the line following the request. If the last line is short, it is filled out to a length of 8. The format of each line is starting memory address (for that line), 8 hex values, 8 ASCII values. For example:

```
*078fh:02a6h 20

Address = 078Fh:02A6h (1935:678)
02A6h 8Bh 0Eh E2h 0Ch 49h CEh 8Bh 16h *....I...*
02AEh E4h 0Ch 42h CEh 89h 56h FAh 3Bh *..B..V.*
02B7h 4Eh FAh 7Fh 3Bh 89h 4Eh F8h 8Bh *N.■;.N..*
```

Note that if a memory location contains a value that is not a valid, displayable ASCII character, a period (.) is displayed in the ASCII field of the dump display.

You can terminate the display of a large memory dump by pressing ESC.

THE EXAMINE/CHANGE MEMORY COMMAND

This command lets you sequentially examine bytes of memory and then either change the contents of each location or leave the contents unaltered. To initiate the display of memory contents, type the variable name, procedure name, line number, or memory address indicating the starting location where the examination is to begin followed by an equal sign (=). After you've typed the starting location, the equal sign, and pressed CODE-RETURN, the memory address and current contents of that address will be displayed in hex. You can then type in a new value to replace the existing value or press CODE-RETURN to leave the existing value unchanged. The debugger then displays the next sequential address and its contents. This sequence continues until you enter a period (.) or ESC to terminate the command. For example:

```
141=CODE-RETURN
07AFh:02A6h = 8Bh  7Bh CODE-RETURN
07AFh:02A7h = 0Eh  CODE-RETURN
07AFh:02A8h = E2h  CODE-RETURN
07AFh:02A9h = 0Ch  FFh CODE-RETURN
07AFh:02AAh = 7Bh  . CODE-RETURN
```

This sequence begins examining memory contents at line number 141 which is at memory address 07AFh:02A6h. This starting location contains 8Bh and the contents are then changed to 7Bh. The contents of the next two locations are displayed and left unchanged. The contents of memory location 07AFh:02A9h are changed from 0Ch to FFh and the command is then terminated after the contents of the next location are displayed.

APPENDIX A. ALTERNATE DEVELOPMENT APPROACHES

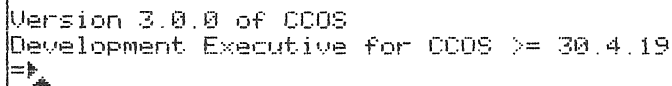
Although the GRiDDevelop program described in Chapter 2 is powerful and easy to use, there may be certain tasks or situations where you prefer another approach. Or, perhaps your personal preference due to past experience on development systems may lead you to seek a different, more familiar approach. To meet these needs, several other approaches are provided and have been used at GRiD prior to the availability of GRiDDevelop.

Let's now look at alternatives to GRiDDevelop: the Development Executive program and command (~Com~) files used with the Do program.

USING THE DEVELOPMENT EXECUTIVE PROGRAM

The DevelopmentExecutive program is a command line interpreter that lets you enter text strings to initiate commands. The system utility programs (described in Appendix C) comprise the commands that you enter via the command line. NOTE: In this context, the compilers and the linker program can also be considered as "utilities" and can be invoked from the command line.

You get into the DevelopmentExecutive program by selecting it from the File form. The DevelopmentExecutive interface displays an arrow as its prompt symbol and the prompt symbol is accompanied by a blinking triangle -- the system cursor. Figure A-1 shows the screen displayed by the DevelopmentExecutive program.



```
Version 3.0.0 of CCOS  
Development Executive for CCOS >= 30.4.19  
=>
```

Figure A-1. The DevelopmentExecutive Interface

Whenever the prompt symbol and the cursor are displayed, you can enter text to specify the utility program that is to be run and any parameters that the program requires. The cursor shows you where the next character you type will appear on the screen. You can edit the command line by moving the cursor using the leftArrow and rightArrow keys and erasing entries or portions of entries with the BACKSPACE key. You can retrieve the last command line entered by pressing CODE-D.

The command line is terminated and the command presented to the system by pressing RETURN or CODE-RETURN. Thus, only a single command at a time can be issued via the DevelopmentExecutive. Therefore, in order to compile several modules, you have to invoke the compiler from the command line for each module after the preceding module compilation had been completed. Then, you must

type in the lengthy linker invocation sequence from the command line. If any errors are encountered along the way, you must repeat the entire sequence, performing each step one at a time. Fortunately, there is a way of simplifying this procedure while using the DevelopmentExecutive. You can create command (~Com~) files and initiate them via the command line interpreter or by selecting them from the File form.

USING THE DO PROGRAM WITH COMMAND FILES

The Do program lets you execute a prearranged sequence of commands contained in a special file -- a command file. The Do program reads the commands from the file and presents them one at a time to the command line interpreter of the DevelopmentExecutive as though you were typing them in via the keyboard. A command file can contain a single command, a command with a long list of parameters, or multiple sequences of commands. Thus, command files save you time and effort by letting you create 'canned', reusable command sequences.

To create a command file, follow these steps:

1. Using GRiDWrite, create a file that has each command (and any parameters) on its own line.
2. End each command line with a carriage return. Note: Be sure that you put a carriage return in at the end of the last line.
3. Save the file specifying a Kind of ~Com~.

To execute a command file from the DevelopmentExecutive, type the command Do and follow it with the file's pathname. You don't have to include the kind -- ~Com~. The execution syntax is:

Do pathname

Let's look at an example which illustrates the power of command files to simplify the program development process. Earlier in this chapter, we gave examples of compiler invocations and a linker invocation initiated from the DevelopmentExecutive command line. The two compiler invocation commands and the linker invocation could all be placed in a single command file that would look like this:

```
Pascal 'w0'MyPrograms'Shell.Pas~Text~
PLM 'w0'MyProgram'FormsInit.Plm~Text~
LINK 'w0'MyPrograms'Shell.Pas~Obj~,
      'w0'MyPrograms'FormsInit.Plm~Obj~, 'w0'Libs'DataForms.Pas~Obj~,
      'w0'Libs'LargeException.Asm~Obj~,
      'w0'Libs'LargeSystemCalls~Lib~ TO 'w0'MyPrograms'Shell~Run~
BIND PURGE FASTLOAD PC(PURGE) MAP ss(stack(+1500))
```

If this command file were named 'w0'MyProgram'CompileLink~Com~, you

could cause the entire sequence to be executed by issuing the following command to the DevelopmentExecutive:

```
Do 'w0'MyPrograms'CompileLink
```

First, the Pascal compiler would be invoked and the file Shell.Pas~Text~ compiled. Next, the PLM compiler would be invoked and the file FormsInit.Plm~Text~ compiled. Finally, the linker would be invoked and all the indicated modules would be linked together.

You can enter comments into a command file by placing each comment on its own line and making the first character a semicolon (;) character. The semicolon tells the Do program that the line is not executable. This capability is handy for "commenting out" selected parts of the command file. For instance, if the file FormsInit.Plm~Text~ had not been changed since the last time it was compiled, you could skip that command by inserting a semicolon in front of it. The command file would then look like this~

```
Pascal 'w0'MyPrograms'Shell.Pas~Text~
;PLM 'w0'MyProgram'FormsInit.Plm~Text~
LINK 'w0'MyPrograms'Shell.Pas~Obj~,
'w0'MyPrograms'FormsInit.Plm~Obj~, 'w0'Libs'DataForms.Pas~Obj~,
'w0'Libs'largeException.Asm~Obj~,
'w0'Libs'LargeSystemsCalls~Lib~ TO 'w0'MyPrograms'Shell~Run~
BIND PURGE FASTLOAD PC(PURGE) MAP ss(stack(+1500))
```

As the Do program reads each command (or comment) from the file, it displays the command on the screen. You can suppress the display of commands by entering the command \$NOLIST in the command file on its own line. You can subsequently enable display of commands and comments by entering the command \$LIST in the command file.

EXECUTING COMMAND FILES FROM THE USER INTERFACE

Command files can be executed directly from the File form of the user interface. This approach is also faster since you simply fill in the File form and confirm -- you don't have to type in a text string to initiate the command file. For example, to execute the command file described earlier ('w0'MyPrograms'CompileLink~Com~) from the user interface, just fill out the File form as shown in Figure A-2.

| | | | |
|---|-------------|---------|--|
| 21-Nov-83 | | 5:19 pm | |
| CompileLink | | Com | |
| UpdateText.Pas | | Text | |
| | | | |
| Device | Hard Disk | | |
| Subject | MyPrograms | | |
| Title | CompileLink | | |
| Kind | Com | | |
| Password | | | |
| Select a file and confirm or Press CODE-? for help | | | |

Figure A-2. Executing a Command File from the File Form

You can also execute the command file from within an application such as GRiDWrite using the Transfer form as shown in Figure A-3.

| | | | |
|------------------------------------|----------------------------------|------|--|
| CompileLink | | Com | |
| UpdateText.Pas | | Text | |
| | | | |
| | | | |
| Device | Hard Disk | | |
| Subject | MyPrograms | | |
| Title | CompileLink | | |
| Kind | Com | | |
| Password | | | |
| Next action | Get new file and its application | | |
| Exchange: Fill in form and confirm | | | |

Figure A-3. Executing a Command File from the Transfer Form

Note that the next-to-last item on the Transfer form is "Next

action" and the initial choice is "Get new file and its application". In this case, the file being retrieved is the specified command file and "its application" is the program Do~Run Command~.

When you want to edit a command file using GRiDWrite, you cannot directly retrieve the file with the File form since this would automatically retrieve the Do program with the file instead of GRiDWrite. Instead, you must already be in GRiDWrite and then must choose "Get new file only" for the "Next action" item on the Transfer menu. Figure A-4 shows the screen when issuing the Transfer command from GRiDWrite to retrieve a command file.

| | |
|------------------------------------|-------------------|
| Get new file and its application | |
| Get new file only | |
| | |
| Device | Hard Disk |
| Subject | MyPrograms |
| Title | CompileLink |
| Kind | Com |
| Password | |
| Next action | Get new file only |
| Exchange. Fill in form and confirm | |

Figure A-4. Retrieving a Command File from the GRiDWrite Transfer Form

APPENDIX B. PROGRAM OVERLAYS

Overlays let you design programs that use the minimum amount of RAM (Random Access Memory) and thus make the maximum amount of RAM space available for data. This is accomplished by having only a part of a program (the "root" module) present in memory at all times. You bring other parts of the the program (the overlays) into memory only when they are needed to perform a particular activity. When an overlay is not being used, it is stored on a mass storage device (bubble memory, hard disk, or floppy disk). When an overlay is no longer needed in memory, it can be unloaded from memory and another overlay brought into the same, or overlapping memory space.

The penalties paid for this more efficient use of memory are reduced speed (when an overlay module is needed, it must be read into memory from the storage device) and slightly more complicated debugging and linking procedures. If your application demands a greater amount of memory for data and can tolerate the performance reductions inherent with overlays, you can utilize the overlay capabilities provided by GRiD OS and implemented using the Linker program. The purpose of this appendix is to clarify the additional factors introduced into program structure and linking operations by the use of overlays.

THE OSOVERLAY PROCEDURE

This GRiD-OS call loads a specified overlay program module into memory. Only one level of overlays is allowed (a program that has been brought into memory as an overlay cannot then issue an `OsOverlay` call). This routine can be called only from the root

(non-overlaid) module which must be present in memory at all times. The format for the call is:

```
PROCEDURE OsOverlay (VAR name : ShortString;  
                    pid : Word;  
                    VAR error : Word);
```

Parameters

name -- a ShortString record containing the name of the overlay. The overlay name is defined using the linker overlay control (See chapter 4 for details).
pid -- the process ID of the overlay. Usually, this will be the same as the pid returned by OsWhoAmI; that is, the overlay is part of the same process that is issuing the OsOverlay call.
error -- the number of any error encountered while calling the overlay.

This procedure call is straightforward and does not add much to the complexity of a program. The only consideration you must remember is that you can use this call only from the root module.

WARNING: When an overlay module is loaded into memory, the previous overlay's code and data segments are overwritten. Therefore, you cannot have any static variables in the data segment of an overlay: they must be in the root module.

PASCAL OVERLAY EXAMPLE

Three Pascal program modules (SampleRoot, SampleOverlay1, and SampleOverlay2) are shown below. During execution of SampleRoot, each of the overlays is loaded into memory (using the OsOverlay call) and then the procedures DoSampleOverlay1 and DoSampleOverlay2 in the overlays are executed.

```
MODULE SampleRoot;  
  
  $INCLUDE ('Hard Disk\Incs\Common.Inc~Text~')  
  $INCLUDE ('Hard Disk\Incs\OsPasTypes.Inc~Text~')  
  $INCLUDE ('Hard Disk\Incs\OsPasProcs.Inc~Text~')  
  $INCLUDE ('Hard Disk\Incs\StringTypes.Inc~Text~')  
  $INCLUDE ('Hard Disk\Incs\StringProcs.Inc~Text~')  
  
  PUBLIC SampleOverlay1;  
    PROCEDURE DoSampleOverlay1;  
  
  PUBLIC SampleOverlay2;  
    PROCEDURE DoSampleOverlay2;
```

```

PROGRAM SampleRoot (INPUT, OUTPUT);

VAR  error : WORD;
     ovl1, ovl2 : stringptr;

FUNCTION LoadMyOverlay (ovl: StringPtr) : BOOLEAN;
BEGIN
    ovl^.dummy := ovl^.len;
    OsOverlay (ovl^.dummy, OSWhoAmI, error);
    LoadMyOverlay := (error = okCode);
END;

BEGIN
    WRITELN('I am the root');
    ovl1 := NewStringLit ('SampleOverlay1');
    IF LoadMyOverlay (ovl1) THEN DoSampleOverlay1;
    ovl2 := NewStringLit ('SampleOverlay2');
    IF LoadMyOverlay (ovl2) THEN DoSampleOverlay2;
    OsExit(error);
END.

**** This is SampleOverlay1 -- A Separate Module ****

MODULE SampleOverlay1;

PUBLIC SampleOverlay1;
    PROCEDURE DoSampleOverlay1;

PRIVATE SampleOverlay1;

PROCEDURE DoSampleOverlay1;
BEGIN
    WRITELN('I am overlay 1');
END;

.

**** This is SampleOverlay2 -- A Separate Module ****

MODULE SampleOverlay2;

PUBLIC SampleOverlay2;
    PROCEDURE DoSampleOverlay2;

PRIVATE SampleOverlay2;

PROCEDURE DoSampleOverlay2;
BEGIN
    WRITELN('I am overlay 2');
END;

.

```

LINKING OVERLAYS

When you use overlays, you must individually link the root module and each of the overlay modules and then link all of them together to resolve the symbols between the root and overlays.

For example, the following sequence from a GRiDDDevelop data file first links the module SampleRoot.Pas~Obj~ with several libraries needed by the program, next links two overlay module files (SampleOverlay1.Pas~Obj~ and SampleOverlay2.Pas~Obj~), and finally links the root module with the two overlay modules.

```
:Link: LINK SampleRoot.Pas~Obj~, ``Hard Disk``Libs~p86rn0~lib~,  
``Hard Disk``Libs~p86rn1~lib~, ``Hard Disk``Libs~p86rn2~lib~, ``Hard  
Disk``Libs~p86rn3~lib~, ``Hard Disk``Libs~8087~Lib~, ``Hard  
Disk``Libs~LargeSystemCalls~Lib~, ``Hard Disk``Libs~DqLarge~Lnk~ TO  
SampleRoot~Lnk~ OVERLAY(ROOT) NOPRINT
```

```
LINK SampleOverlay1.Pas~Obj~, ``Hard Disk``Libs~p86rn0~lib~, ``Hard  
Disk``Libs~p86rn1~lib~, ``Hard Disk``Libs~p86rn2~lib~, ``Hard  
Disk``Libs~p86rn3~lib~, ``Hard Disk``Libs~8087~Lib~ TO  
SampleOverlay1~Lnk~ OVERLAY(SampleOverlay1)  
ASSUMEROOT(SampleRoot~Lnk~) NOPRINT
```

```
LINK SampleOverlay2.Pas~Obj~, ``Hard Disk``Libs~p86rn0~lib~, ``Hard  
Disk``Libs~p86rn1~lib~, ``Hard Disk``Libs~p86rn2~lib~, ``Hard  
Disk``Libs~p86rn3~lib~, ``Hard Disk``Libs~8087~Lib~ TO  
SampleOverlay2~Lnk~ OVERLAY(SampleOverlay2)  
ASSUMEROOT(SampleRoot~Lnk~) NOPRINT
```

```
LINK SampleRoot~Lnk~, SampleOverlay1~Lnk~, SampleOverlay2~Lnk~ TO  
SampleRoot~Run~ BIND SS(STACK(+1500)) PC(PURGE)
```

The key statements in the link commands in this example are as follows:

- * When linking the root module, you must specify that the resultant output file be designated with the control OVERLAY(ROOT). This tells the linker program that this module is a root module.
- * The output files for the two overlay modules must be specified with the controls OVERLAY(overlayName) and ASSUMEROOT(rootName) to tell the linker program both the name of each overlay and name of the root module to which each will be linked.
- * The last link invocation in the command file, must first name the root module, then the overlay modules, and finally the executable output file. The output file consists of the three modules

(SampleRoot, Overlay1 and Overlay2) bound and linked together.
Note: the BIND, PURGE, FASTLOAD and StackSegment (SS) controls
should only be used in this last link statement.

ADDITIONAL OVERLAY CONSIDERATIONS

To obtain most efficient performance with overlays, your root
program should keep track of which overlay is currently in
memory. If you do not do this, an overlay that is already in
memory might be called and needlessly reloaded.

The ASSUMEROOT control, can reduce the amount of time needed to
link and can also produce smaller resultant output files.

When you're debugging a program with overlays, you can set
breakpoints in the overlays but the breakpoints must be set only
after the overlay is loaded and the breakpoints must be cleared
before the overlay is removed.

FORTRAN OVERLAY EXAMPLE

This section shows a FORTRAN program that uses OsOverlay to call
two overlay subroutines.

```
**** This is the Root Module ****  
Program Bench
```

```
Integer*2 ivar , ner  
Integer*2 OSWHOAMI  
Integer*1 INAME (6)  
  
DATA INAME1 /4,83,85,66,49,32/  
DATA INAME2 /4,83,85,66,50,32/  
  
ivar = oswhoami ()  
call osoverlay(INAME1,%VAL(ivar),ner)  
if(ner .eq. 0) call sub1  
call osoverlay(INAME2,%VAL(ivar),ner)  
if(ner .eq. 0) call sub2  
STOP  
END
```

```
**** This is the OverLay1 Module -- A Separate Module ****
```

```
subroutine sub1  
character*1 big(10000)  
big(1) = 'a'  
big(10000) = 'z'
```

```

write(6,1,IOSTAT=IOS,ERR=100) big(1) , big(10000)
format(' big start and stop sub1',a1,2x,a1)
return
WRITE(6,105) IOS
FORMAT('I/O STATUS = ',I6)
RETURN
end

```

**** This is the OverLay2 Module -- A Separate Module ****

```

subroutine sub2
character*1 big(10000)
big(1) = 'a'
big(10000) = 'z'
write(6,1,IOSTAT=IOS,ERR=100) big(1) , big(10000)
format(' big start and stop sub2',a1,2x,a1)
return
WRITE(6,105) IOS
FORMAT('I/O STATUS = ',I6)
RETURN
end

```

The following sequence from a GRiDDDevelop data file first links the module Bench.ftn~Obj~ with several libraries needed by the program, next links two overlay module files (Sub1.ftn~Obj~ and Sub2.ftn~Obj~), and finally links the root module with the two overlay modules.

The link commands for these FORTRAN overlay modules would be as follows:

```

Link Bench.ftn~obj~, ``Hard Disk``Libs\F86RN0~LIB~, ``Hard
Disk``Libs\F86RN1~LIB~, ``Hard Disk``Libs\F86RN2~LIB~, ``Hard
Disk``Libs\F86RN3~LIB~, ``Hard Disk``Libs\F86RN4~LIB~, ``Hard
Disk``Libs\CEL87~LIB~, ``Hard Disk``Libs\EH87~LIB~, ``Hard
Disk``Libs\8087~LIB~, ``Hard Disk``Libs\DCON87~LIB~, ``Hard
Disk``Libs\LargeSystemCalls~Lib~, ``Hard Disk``Libs\DqLarge~Lnk~
TO Bench~Lnk~ OVERLAY(ROOT) NOPRINT

```

```

Link Sub1.ftn~obj~, ``Hard Disk``Libs\F86RN0~LIB~, ``Hard
Disk``Libs\F86RN1~LIB~, ``Hard Disk``Libs\F86RN2~LIB~, ``Hard
Disk``Libs\F86RN3~LIB~, ``Hard Disk``Libs\F86RN4~LIB~, ``Hard
Disk``Libs\CEL87~LIB~, ``Hard Disk``Libs\EH87~LIB~, ``Hard
Disk``Libs\8087~LIB~, ``Hard Disk``Libs\DCON87~LIB~, ``Hard
Disk``Libs\DqLarge~Lnk~ TO Sub1~Lnk~ OVERLAY(Sub1)
ASSUMEROOT(Bench~Lnk~) NOPRINT

```

```

Link Sub2.ftn~obj~, ``Hard Disk``Libs\F86RN0~LIB~, ``Hard
Disk``Libs\F86RN1~LIB~, ``Hard Disk``Libs\F86RN2~LIB~, ``Hard

```



```
Disk`Libs`F86RN3`LIB`, ``Hard Disk`Libs`F86RN4`LIB`, ``Hard  
Disk`Libs`CEL87`LIB`, ``Hard Disk`Libs`EH87`LIB`, ``Hard  
Disk`Libs`8087`LIB`, ``Hard Disk`Libs`DCON87`LIB`, ``Hard  
Disk`Libs`DqLarge`Lnk` TO Sub2`Lnk` OVERLAY(Sub2)  
ASSUMEROOT(Bench`Lnk`) NOPRINT
```

```
LINK Bench`Lnk`, Sub1`Lnk`, Sub2`Lnk` TO Bench`Run` BIND  
SS(STACK(+1500)) PC(PURGE)
```


APPENDIX C. SYSTEM FILES AND UTILITIES

This appendix describes the system files that the GRiD Compass uses and the utility programs available to assist you during program development and system housekeeping. NOTE: Most of the tasks performed by the utility programs described here can be handled more easily by GRiDManager. Unless you have a real need to use the command line interpreter or a need for a specific utility, you should use GRiDManager.

The system utility programs operate on devices and files and are invoked via the command line interpreter. You can run any of them without regard to the current subject since they are under the Programs subject.

SYNTAX NOTATION

Syntax notation in this appendix operates under the following conventions.

- * Keywords (command or function names) are in capital letters.
Examples: CAT, DUMP.
- * Parameters are in lowercase letters.
Example: PREFIX pathname.
- * Square brackets enclose optional parameters.
Example: CAT [pathname].
- * Braces or curly brackets surround a choice of parameters with each parameter separated by a vertical slash.
Example: {real|integer}.

If a parameter choice is an option, the choice is enclosed in square brackets.

Example: [{real}integer{ }].

A note on syntax statements: you must enter parameters in the order given in the syntax statement.

ENTERING COMMANDS

Throughout this appendix we use uppercase letters in writing about commands. As stated above, in the case of syntax notation, command names are in all uppercase. When discussing a command in a sentence, the first letter will be capitalized. For example, "Only the Cat command recognizes the wildcard character."

However, you can enter commands, program names, and file pathnames in any form you want with regard to capitalization. The system understands "CAT," "cat," and even "cAt" as the same command.

WILDCARDS

Some of the utilities recognize one wildcard character -- the asterisk (*). You can substitute one wildcard for any character, for any string of characters, or for no character(s). Wildcards work only with the Cat program.

For example, let's say you have five titles under 'Hard Disk'Morebees -- Brains, Brass, Barrooms, Beanbag, and Edna. Typing a command and following it with 'Hard Disk'Morebees'B*S would cause the command to act on all the file names that begin with B and end with S: Brains, Brass, and Barrooms. Beanbag fails because it doesn't end with S, and Edna neither begins with B, nor ends with S. B* would cause the command to execute on all files except Edna.

A pathname consisting of an asterisk only will act on all files that exist under the current prefix.

THE @SYSTEMERRORS FILE

The file named @SystemErrors~text~ in the Programs subject contains the text that is displayed when a system error is encountered. If this file is not present, an error number will still be displayed when errors are encountered, but there will be no explanatory message with the number.

THE ACTIVATE PROGRAM

This program activates a new device and adds it to the list of currently active devices. 'Activating' a device consists of associating a device name that you specify with the appropriate device driver program and GPIB address. The operating system automatically activates the following devices whenever the system is booted:

| device name | GPIB addr (hex) |
|-------------------|-----------------|
| Floppy Disk | 0005 |
| Bubble Memory | none |
| Hard Disk | 0004 |
| Portable Floppy | 0006 |
| Extra Hard Disk | 000C |
| Extra Floppy Disk | 000D |
| gpib | none |
| bb (bit bucket) | none |
| ci (console in) | none |
| co (console out) | none |

If a device is not physically present, it is not activated. However, you can later activate a device using the "Add a device" command from GRiDManager. The currently active devices will be displayed on the File form.

The driver programs for local mass storage devices are incorporated into the operating system and do not exist as separate files. There are three different modem-related files -- each corresponding to the actual physical modem type used with the computer: the three files are named CompassGRiDInternal, GRiDCaseHayesInternal, and HayesExternal. They are under the Programs subject and their kind is ~Modem~. The modem can be activated by typing the command:

Activate CompassGRiDInternal AS Modem

If you need to activate a device whose driver program does not exist as a separate program, the syntax you must use with Activate is the more complicated form shown below:

ACTIVATE DEVICE device1 AS device2 [m] [gpib-addr]

This will look for device1 in the active device table and will use that device driver to create another device called device2. For example, if you were connecting a second hard disk to your system, you could activate that device as follows:

ACTIVATE DEVICE 'Hard Disk' AS 'Second Disk' m 6

This would activate the second hard disk and assign it the device

name Second Disk with a GPIB address of 6.

THE CAT (CATALOG) PROGRAM

The program lists all the titles in the subject directory, relative to the file level you have specified. You can cause the program to print the requested directory to some device other than the screen (including a text file) by specifying a second pathname. Here is the complete syntax:

```
CAT [pathname1] [pathname2] [!] [?]
```

Typing the Cat program name without parameters will cause the program to display all the titles under the current subject in a tabular form somewhat like the one below (all numbers are in decimal):

```
Files matching 'Hard Disk'mystuff'*
File Name           Length   Last Modified
Gridstar.1~Worksheet~      107   03/16/82 09:45
Statusform~text~          243   02/28/82 15:12
Forecast.1~Text~          1468   03/03/82 11:47
```

storage utilization: 1661/10404 pages, 15.9%

The first line tells the device ('w'), subject ('mystuff'), and title description (* -- the wildcard) for the titles on the screen. The first column displays these titles.

The second column displays the number of bytes that each file occupies.

The first number after "storage utilization" indicates the total number of pages taken by all files on the bubble, diskette or disk.

The second number shows the total number of pages on the bubble, diskette or disk. To find the number of free pages available, subtract the first number from the second.

The third number is the percentage of occupied pages to the total number of pages available on the device.

When the file names are being displayed on the screen, you can stop scrolling by typing CTRL-S. To restart scrolling, press CTRL-S again. Pressing CODE-ESC cancels the Cat program and thus the scrolling.

Note that when you enter Cat without a pathname, the Cat program puts in an invisible wildcard character that defaults to the current device/subject prefix and all the titles within that subject. For example, if your current prefix were 'Hard Disk'Breakfast, the first

line of your catalog display would read:

```
Files matching 'Hard Disk'Breakfast'*
```

However, if you wanted to look at a different prefix grouping, like 'Floppy Disk'Lunch, you would have to enter this explicitly or change your prefix first. To see all the titles under this prefix, you would type:

```
Cat ``Floppy Disk``Lunch`*
```

Creating a Catalog File

The Cat program lets you specify a second pathname if you want to send your catalog information to a text file instead of to the screen. We call this file a "catalog file." This file can be sent to either disk or to an output device, such as a printer. The program prepares the file during execution.

Note that the syntax for a catalog file requires that you precede the name of the catalog file with a pathname for the title(s) you want catalogued. At a minimum, this pathname must be the asterisk (*).

For example, typing `Cat * Catchall` would create a file called `Catchall` and write into it all titles under the default prefix. Similarly `Cat BB*.COM BEBEFILE` would create a file under the default prefix called `BEBEFILE` and put in it all titles beginning with the letters `BB` and ending with `.COM`.

By preceding the name of the catalog file with a device name, you can direct where the system will send the catalog file. Without the device name, the system will set up the file on the default device.

! (Exclamation Point)

Placing the exclamation point after Cat (or after Cat and any of its parameters) will cause the Cat program to display file titles without file lengths and dates and times. As a result, the exclamation point causes the catalog to display much more rapidly.

? Question Mark

Placing the question mark after Cat and a title will cause the Cat program to search for that title under every subject on the currently prefixed device.

THE COMPARE PROGRAM

This program compares two files for equality or inequality and is useful for checking if two files are duplicates. The syntax is simply:

```
COMPARE file1 file2
```

If the two files are identical, the program displays the word "Same", if they are not identical, a display similar to the following example appears:

```
Files are different at location 3591
Page#/Offset: 7 63
```

```
0E07: 32 3A 20 55 73 65 72 20 2:.User.
0E07: 33 3A 20 55 73 65 72 20 3:.User.
```

The "location" in the first line indicates the byte position in the file where a difference between the two files was detected. In this example, the bytes at position 3591 were detected to be different. The Page#/Offset line indicates the media location where the difference was detected. Each page (sector) is 504 bytes long.

The last two lines are a hexadecimal representation of the data in the two files in the area where the difference was detected. The first line is the data from the first file specified after the COMPARE command and the second is the data from the second file. In this example, the byte at 3591 (0E07h) in the first file was 32h the corresponding byte in the second file was 33h.

THE DEACTIVATE PROGRAM

This program will deactivate a device, removing it from the active device table. Refer to the Activate program for a discussion of devices and device activation. The syntax for this program is simply:

```
DEACTIVATE dev
```

where "dev" is the device name as listed in the active device table (see the LADT program).

NOTE: You should not usually deactivate any of the devices that are automatically activated by the system during power up (Floppy Disk, Hard Disk, Bubble Memory, etc.). The drivers for these devices are incorporated into the operating system software and do not exist as separate files. They therefore can not be reactivated using the Activate program described earlier in this chapter.

THE DEVELOPMENT EXECUTIVE PROGRAM

The DevelopmentExecutive~Run~ file under the Programs subject is the program that provides the command line interpreter. Refer to Appendix A for a discussion of the DevelopmentExecutive interface.

THE DO PROGRAM

Do~Run Com~ is the program that lets you execute a command file. The Do program reads the commands from the file and presents them to the system as though you were typing them in at the command line of the development interface. Thus, command files save you time and effort by letting you create 'canned', reusable command sequences. For a discussion of command files and the Do program, refer to Appendix A.

THE DUMP PROGRAM

The Dump program sends the contents of a file in both HEX and ASCII to a specified destination file. If no destination file is specified, the contents are dumped to the screen. The syntax for this program is:

```
DUMP sourceFile [destFile]
```

The information that follows is an example of a dump of the System.Init~Com~ file:

```
FILE = system.init~Com~
```

```
0000  24 6E 6F 6C 69 73 74 0D 0A 61 63 74 69 76 61 74
*$nolist..activat*
0010  65 20 6D 6F 64 65 6D 0D 0A 21 20 60 77 30 60 70  *e
modem..! `w`p*
0020  72 6F 67 72 61 6D 73 60 73 63 72 65 65 6E 77 61
*programs'screenwa*
0030  74 63 68 7E 72 75 6E 7E 0D 0A 0D 0A
*tch~run~.... *
```

```
END OF FILE
```

The four-digit number at the beginning of each row is the hexadecimal offset of the first byte in that row. Thus the first character in the second row (hex 65) is byte number 0010 (hexadecimal) in the file. Next, the hexadecimal representation of each byte in the file is provided, with 16 bytes displayed in each row. To the right, the ASCII representation of each byte is displayed.

THE ELAPSED TIME PROGRAM

This program times the execution of any program you specify. The syntax is:

ELAPSEDTIME pathName

where pathName specifies some executable file. After the specified program has completed execution, control is returned to the Development interface and the time that elapsed since you invoked the program is displayed.

THE EXECUTIVE FILE

This file is loaded into memory whenever the system is booted. It displays the initial File form and is required in order to perform such as activities as exchanging files.

THE LADT (LIST ACTIVE DEVICE TABLE) PROGRAM

The active device table contains a list of all devices that have been activated (See the Activate program earlier in this chapter). A device needn't be on-line to be included in this list. To see the names of all active devices, type LADT and press RETURN. The list can comprise any of the device names normally seen via the File form (Hard Disk, Floppy Disk, Bubble Memory, etc.) plus the following:

- * WORK WORK is a virtual device that provides space for temporary work files used by the compilers, the linker, and programmers.
- * W W stands for Winchester -- the hard disk.
- * F F stands for Floppy. You must add a number to this device name. Address the floppy drive as f0. Note that f will default to f0.
- * PLOTTER PLOTTER is the device name for that portion of the GPIB that connects to a plotter. Only valid when a "Current plotter" has been designated via GRiDManger.
- * PRINTER PRINTER is the device name for that portion of the GPIB that connects to an Epson printer. Only valid when a "Current printer" has been designated via GRiDManger.
- * GPIB A generic term, not an addressable device,

covering all device addresses that hook to the GPIB port.

- * BB The "Bit Bucket" (aka the "Byte Bucket") is a null device, used mostly for testing.
- * CI CI, the keyboard, stands for "Console In."
- * CO CO, the computer screen, stands for "Console Out."

Two optional parameter words (separated by a space) can be issued following the LADT command. The first word is a "mask" and the second word is an 16-bit "value" that results from ANDing the mask word with each device's "mode" parameter (as specified via the OsAddDevice call). The "mode" parameter bits are defined as follows:

Bit #

- 0 -- mass storage. If set to 0, indicates that the device is not a mass storage device. If set to 1, indicates a mass storage device such as hard disk.
- 1 -- visible/invisible. If set to 0 and the mass parameter is TRUE, the device will appear on active device list and be displayed on the File form. If set to 1 or if the mass parameter is FALSE, the device will be invisible.
- 2 -- local/remote. If set to 0, the device is local. If set to 1, the device is remote, that is, accessed through the serial port or a modem).
- 3 -- peripheral bus (GRiDServer devices only).
- 4 -- server. If set to 0, indicates that the device is not a network server. If set to 1, indicates that device is a network server accessed through GRiDLink or PhoneLink.
- 5 -- alias. If set to 1, indicates that aliases such as "W" for Hard disk, or "F" for Floppy disk will be recognized. Otherwise, set to 0.
- 6 -- reprogrammable. If set to 1, indicates that the device (usually a floppy disk) can be reprogrammed to accept different data formats (for example, either 8 or 9 sectors).
- 7 -- search. If set to 0 and if the mass storage bit (bit 3) is set to 1, indicates a searchable device. GRiD-OS may search this device for an appropriate application program, such as GRiDWrite~Run Text~ to use with a file of Kind ~Text~. If set to 1, the device will never be searched.
- 8 -- spool (server devices only). If set to 1, indicates that the device is a spooler device (such as the printer queue). Otherwise set to 0.
- 9 -- admin (server devices only). If set to 1, indicates that the device can be accessed only by server administrators. Otherwise set to 0.
- 10 -- removable. If set to 1, indicates that the device has removable media (for example, a Floppy disk). Otherwise set to 0.

11 -- volume. If set to 1, indicates that the device has been given a volume name. Otherwise set to 0.
12 - 15 -- reserved (always set to 0).

The bits of the "mask" parameter specify which bits of each device's mode word should be examined. The bits of the value parameter specify whether the bits being examined in the mode word should be set to 1 or to 0 in order to qualify for listing.

For example, if the mask parameter issued with LADT is 0003h, bits 0 (mass storage/not mass storage) and 1 (visible/invisible) of the device mode words will be examined. If the value parameter is 0001, then only those devices having the mass storage bit set to 1 and the visible bit set to 0 will be listed by LADT.

THE LOAD PROGRAM

The Load program simply loads an executable module into memory.

LOAD pathName

THE PREFIX PROGRAM

When you boot the system, the default subject is always Programs and the device will be whichever device you directed the system to -- bubble, hard disk, or floppy. (If you did not explicitly specify a device -- by holding down the 'H' or 'F' key during the boot sequence -- the system first tries the bubble, then the hard disk and finally the floppy, until it finds one of those devices ready.)

When speaking in terms of pathnames, we refer to the initial 'device' subject pair as the "default prefix." By "default," we mean that any time the system must access a file, it will try to find the file in question under the default prefix, unless told to look elsewhere. By "prefix", we mean the device-subject pair.

You can override the default prefix by explicitly typing another prefix before a title. To reset the prefix to a different device pair altogether, use the Prefix program.

To execute the program, type Prefix, a space, and the name of the new default prefix (both the physical device and a subject, i.e., 'f'Lunch). Finally, press RETURN. Note that a tick should not follow the subject name. The default will remain with the new pair, until you give the system a different pair by reinvoking the Prefix program. The syntax is simple:

PREFIX ['device'] subject

For example, Prefix 'f'Programs will cause any further storage

access to look to the floppy drive, under the subject "Programs." Typing Prefix ``Hard Disk`Breakfast will change the default so that subsequent searches for titles look under the "Breakfast" subject.

Note that the device is optional when specifying a new prefix. If you do not include a device name, the new prefix will become the specified subject combined with the previously prefixed device. Thus, if the current prefix is `Hard Disk`Breakfast, typing Prefix Lunch will change the default prefix to `Hard Disk`Lunch.

THE SOFTKEYS FILES

The numeric keys on the keyboard have been programmed to generate often used words and symbols at the command line level, that is, from the development interface. For example, typing CODE-SHIFT-4 will print the word "Pascal" on the screen. Likewise, pressing CODE-3 will cause "Programs" to appear.

Thus, these keys let you quickly generate frequently used command messages. The following table shows all preprogrammed softkey messages and the key combinations for generating them.

| KEY | CODE | CODE-SHIFT |
|-----|---------------------|-------------|
| --- | ---- | ----- |
| 1 | ``Floppy Disk`` | |
| 2 | ``Hard Disk`` | |
| 3 | Programs` | |
| 4 | ``Bubble Memory`` | Pascal |
| 5 | ``Portable Floppy`` | PLM |
| 6 | GRiDWrite | Fortran |
| 7 | ~Text~ | |
| 8 | ~Lst~ | `Printer |
| 9 | ~Com~ | ~Worksheet~ |
| 0 | ~Run~ | ~Graph~ |
| - | Prefix | |

Table C-1. Preprogrammed Softkeys

Programming the Softkeys

You can substitute your own message(s) for any current softkey messages. To do this, edit the file `Hard Disk`Programs`SoftKeys`~Text~. This file contains each message (beginning with "f" and ending with "Cat") separated by a carriage return. Select a message for replacement and erase it. Then type your substitute message in its place. Save the revised file.

To activate your new message(s), you must load the revised file. You have two ways of doing this: either type CODE-= or reboot the system by pressing the reset button.

Your new message will appear whenever you press the key combination that draws its characters from the position in which you placed your message. For example, if you replaced "Fortran" with a favorite subject name, "MyStuff," you would see "MyStuff" every time you pressed CODE-SHIFT-6.

Multiple Softkey Files

You can place different 'SoftKeys~Text~ files under different subjects. Each file can have entirely different messages. In such a case, the file's messages will be available only when you're in that file's subject. Whenever a subject does not have its own SoftKeys file, it will draw messages from the SoftKeys file in the subject "Programs."

To activate the Softkeys file in a subject other than "Programs," type CODE-=. If you don't issue this command, any use of the softkeys will default to Programs's Softkeys file.

THE STATUS PROGRAM

This program displays system status information including memory utilization, the current prefix, and currently loaded packages. To run the program, simply type STATUS and press RETURN. The information displayed will be similar to that shown below:

```

Version 3.0.0 of CCOS
Development Executive for CCOS >= 30.4.19
=>status
current prefix: `Hard Disk`Programs

total free bytes: 93536
number of free blocks: 13
largest free block: 65535
total allocated bytes: 102688
number of allocated blocks: 105
largest allocated block: 65535
=>

```

THE SUMMARIZE PROGRAM

This program analyzes a file's usage of memory and displays the results of that analysis. The syntax is:

```
SUMMARIZE sourcePathName [destPathName] ['commentString']
```

The sourcePathName specifies the file that is to be summarized. The results of the summary will always be displayed on the screen. The optional destPathName lets you specify that the results also be sent to another destination -- typically the printer. The optional commentString must be enclosed in single quotation marks and will be displayed at the beginning of the summary information. For example:

```
SUMMARIZE ``Hard Disk``programs`MyApp~run~ `printer `10-30-83
Summary`
```

This would cause the following information to be displayed on the screen and also printed at the printer:

10-30-83 Summary

File: 'Hard Disk'programs'MyApp~run~

| | | |
|-----------------|-----|-------------|
| Initialization: | 17 | 272 |
| Code/Const: | 94 | 9679 (8927) |
| Fixup: | 43 | 680 |
| Waste: | 0 | 0 |
| Total: | 154 | 10631 |
| Overhead: | | 16.0% |
| Data segment: | | 52 |
| Stack segment: | | 1026 |

The left column of numbers shows how many records are devoted to each category and the right column is the number of bytes in each category.

THE TIME PROGRAM

This program simply displays the current time and date maintained by the clock chip. To display time, simply type TIME and press RETURN.

THE UNLOAD PROGRAM

This program simply unloads a run module that was previously LOAded (either explicitly with the LOAD program, or by the system at boot time). The syntax for the Unload program is:

UNLOAD pathName

THE WORK PROGRAM

This program simply specifies the device that will be the 'work' device. The language compilers and the Link program require temporary work files for their operation. Additionally, system programmers use work files for applications that require temporary files. Work files are discarded upon completion of the operation for which they were being used. These files assume the presence of a virtual device named Work. The system automatically designates the physical device that you boot from as the Work device. If you want to change this default, run the Work program using the following syntax:

WORK 'dev

If you boot your system from Bubble Memory, you might get a device full message when compiling programs. You should change the work device to Hard Disk if you boot from Bubble Memory.

APPENDIX D. LINK ERROR MESSAGES

This appendix describes error messages that may be produced by the Link program. Only those errors deemed likely to occur in the system are listed. Should you encounter an error message generated by one of the Link program that is not listed here, contact the GRiD Customer Support Center.

Remember, it is possible to receive an error message generated by the operating system (GRiD-OS) while you are running the Link program. Refer to the GRiD-OS Reference manual for a complete listing of system error messages.

The Link program generates both error messages and warning messages. They are listed in the pages that follow in numerical order with warning and error messages intermixed.

Error messages are always fatal: they terminate processing of the input file(s) and halt execution of the Link program. All open files are closed and the contents of the print file and the object file are undefined.

Warning messages are not fatal. They are listed consecutively as warning situations are encountered. Read the discussion of the warning carefully to determine whether the resultant code is valid.

ERROR 1: I/O ERROR

What happened The operating system detected an I/O error in the

input file.

What to do Check the pathnames specified for the input file and check for possible media errors.

ERROR 2: I/O ERROR

What happened The operating system detected an I/O error in the print file.

What to do Check the pathnames specified for the print file and check for possible media errors.

ERROR 3: I/O ERROR

What happened The operating system detected an I/O error in the object file.

What to do Check the pathnames specified for the object file and check for possible media errors.

ERROR 4: I/O ERROR

What happened The operating system detected an I/O error in the console file.

What to do Check the pathnames specified for the console file and check for possible media errors.

ERROR 5: INPUT PHASE ERROR

What happened A record encountered during the second phase of linkage did not agree with information gathered during the first phase of linkage. This error is caused by a data transmission error or an internal error in the Link program itself.

What to do Contact the GRiD Customer Support Center. Be prepared to provide a copy of the object file, the Link invocation line, and your version of the Link program.

ERROR 6: CHECK SUM ERROR

What happened The check sum field at the end of one of the object module records indicates a transcription error. This can be caused by any number of data encoding or media errors.

What to do Retranslate the source that produced the specified object module where the error was detected. Then relink.

ERROR 7: COMMAND INPUT ERROR

What happened An error was detected while attempting to read the complete invocation line.

What to do Check the invocation line for errors and try again.

WARNING 8: SEGMENT COMBINATION ERROR

What happened Two segments with the same name can not be combined because they have different combination attributes or incompatible alignment attributes. The linker will continue processing pass 1 but pass 2 will not be started. The resultant output object file is useless and the print file contains limited information.

What to do Retranslate the source that produced the specified file and module. Then relink.

WARNING 9: TYPE MISMATCH

What happened There is a public/external symbol pair for which the type definitions do not agree. The linker continues processing using the first definition only. The object file and the print file should be valid, except the second definition for the symbol is ignored.

What to do Modify the offending public or external declaration and recompile and relink the source file.

WARNING 10: DIFFERENT VALUES FOR SYMBOLS

What happened The same symbol was declared public in two different modules. The specified file and module contains the second definition encountered. The linker continues processing using the value of the first public definition; the second definition is ignored. Both the print file and the object file will be valid.

What to do This situation will often occur in the normal course of events, for example, when you are linking library files along with CompactSystemCalls~Lib~. In such cases, you can usually ignore this warning. If it is a problem, change the name of the symbol in either the specified file or in the file containing the earlier definition of the symbol.

ERROR 11: INSUFFICIENT MEMORY

What happened Because of an extensive use of public symbols, there is insufficient memory for the linker to build its internal tables and data structures.

What to do If possible, unload unneeded packages, such as common. Otherwise, try incremental linkages. That is, link smaller sets of files together using the NOPUBLICS control, then link the resulting composite modules together.

WARNING 12: UNRESOLVED SYMBOLS

What happened External symbols were declared that could not be resolved during this linkage. (This is quite common when performing an incremental linkage.) The print file is valid. The object file must be linked to resolve the external references.

What to do Link the object file to a file that will resolve the external references.

WARNING 13: IMPROPER FIXUP

What happened An external reference makes assumptions about the segment register that do not agree with the

assumption made for the public definition. The linker continues processing. The object file will not be usable, but the print file will be complete and accurate.

What to do Try recompiling with a different model of segmentation, or change the source and reassemble.

WARNING 14: GROUP ENLARGED

What happened The specified group name has been defined twice in two different modules and the segments contained in the two definitions are different. The two groups are combined into one with all segments that were in either group included in the resulting group. Segments with the same segment name, class name, and overlay name are combined. The linker continues processing and both the print file and object file are valid.

What to do No action should be necessary.

ERROR 15: LINK86 ERROR

What happened A fatal, internal error has occurred within the Link program itself.

What to do Contact the GRiD Customer Support Center. Be prepared to provide a copy of the object file, the invocation line, and your version of the Link program.

ERROR 16: STACK OVERFLOW

What happened Link's run time stack used for type matching has overflowed. This can be caused by an overly complex type definition of one of your symbols.

What to do Try incremental linkage (see error 11). If the error persists, contact the GRiD Customer Support Center.

WARNING 17: SEGMENT OVERFLOW

What happened The combination of two or more segments has resulted

in a segment that exceeds 64K. The linker continues processing during the current pass, but the print and object files are not useable.

What to do Reorganize your segments and reassemble.

WARNING 18: IMPROPER START ADDRESS

What happened A start address was found in one of the overlay modules, and none was found in the root module. This error is often caused by misordering the input modules in the input list. The linker ignores the start address in the specified overlay module and continues processing.

What to do If you want the module containing the start address to be the root, relink with that module first in the input list.

ERROR 19: TYPE DESCRIPTION TOO LONG

What happened The type definition is too long to fit in the linker's symbol table.

What to do Contact the GRiD Customer Support Center. Be prepared to provide a copy of the object file, the invocation line, and your version of the Link program.

ERROR 22: INVALID SYNTAX. ERROR IN COMMAND TAIL NEAR #

What happened This error is usually the result of a typographical error in the invocation line. The partial command tail up to the point where the error was detected is printed.

What to do Check the invocation line and reinvoke the Link program more carefully.

ERROR 23: BAD OBJECT FILE

What happened The link program has detected an inconsistency in the fields of a record in the specified input file. This error could be caused by the compiler or could be due

to a media problem.

What to do Recompile and then try relinking. If the problem persists, contact the GRiD Customer Support Center.

WARNING 24: CANNOT FIND MODULE

What happened The specified module cannot be found in the specified library file. The linker continues processing as if the specified module were not in the list.

What to do If the module is important, you can link it into the output object file later.

WARNING 25: EXTRA START ADDRESS IGNORED

What happened A start address has been encountered in more than one module indicating that you have specified more than one main module in the input list. The linker uses the start address encountered earlier and ignores the start address in the module specified here with the warning message. Processing continues with no other side effects.

What to do Do nothing, if the start address in the specified module was intended to be ignored.

ERROR 26: NOT AN OBJECT FILE

What happened The file specified with the error message is not an object file. This error is usually caused by a typographical error in the input list. However, some media problems can also cause this error.

What to do Check the invocation line and try again. If you suspect media problems, try recompiling and relinking.

WARNING 28: POSSIBLE OVERLAP

What happened This warning is issued when the linker combines two absolute segments. Processing continues with no side effects.

What to do If there is an actual conflict, the loader will

detect the overlap.

ERROR 30: LIBRARY IS NOT ALLOWED WITH PUBLICSONLY CONTROL

What happened The file specified with the error message is a library file and libraries are not allowed in a PUBLICSONLY control.

What to do Remove the library file from the PUBLICSONLY argument list and reinvoke the linker.

WARNING 32: EXTRA REGISTER INITIALIZATION RECORD IGNORED

What happened You have included two main modules in your input list. The linker uses the first register initialization record and ignores the second. Processing continues with no side effects.

What to do If the register initialization information in the file specified with the warning message should be used instead of the first such record encountered, then modify your input list. Otherwise, no action is required.

ERROR 33: ILLEGAL USE OF OVERLAY CONTROL

What happened While processing input modules for an overlay, the linker found an overlay definition in the file and module specified with the error message. A module being used for an overlay cannot itself specify an overlay.

What to do Remove the specified file from the input list and relink.

ERROR 34: TOO MANY OVERLAYS IN INPUT FILE

What happened The file and module specified with the error message contains more than one overlay definition.

What to do Remove the specified file from the input list or correct the file so that it has only one overlay definition, then relink.

ERROR 35: SAME OVERLAY NAME IN TWO OVERLAYS

What happened The file specified with the error message contains an overlay that has the same name as an overlay already encountered in the input list.

What to do Remove one of the duplicate names from the input list and relink. If both overlays are needed, relink one of them specifying a different overlay name.

ERROR 36: ILLEGAL OVERLAY CONSTRUCTION

What happened Some of the modules in the input list contain overlay definitions while others do not. This is illegal: all modules in the input list must be the same with respect to overlays.

What to do Remove the non-overlay files and relink.

WARNING 37: DIFFERENT PUBLICS FOR EXTERNAL IN ROOT

What happened The linker has found two symbol definitions in the overlay modules that resolve the same external symbol definition in the root. The definition in the file and module specified with the warning message is ignored and processing continues with no side effects.

What to do Remove the unwanted symbol definition and relink.

ERROR 41: SPECIFIED SEGMENT NOT FOUND IN INPUT MODULE

What happened This error is usually caused by a typographical error in the SEGSIZE control.

What to do Check the input list for accuracy. If necessary, find the module that contains the specified segment and add it to the input list.

WARNING 42: DECREASING SIZE OF SEGMENT

What happened The size change specified in SEGSIZE has caused the

linker to decrease the size of the specified segment. Decreasing the size of a segment can cause sections of code to be unaccounted for during the memory allocation process. Processing continues with no side effects.

What to do This is usually caused by leaving out the plus sign in the `SEGSIZE(STACK+nnnn)` control. Check the input list and correct.

ERROR 43: SEGMENT SIZE OVERFLOW; OLD SIZE+CHANGE > 64K

What happened The size change specified in the `SEGSIZE(STACK(+nnnn))` control caused the segment to become greater than 64K.

What to do Reinvoke the linker with the correct `SEGSIZE(STACK(+nnnn))` control.

ERROR 44: SEGMENT SIZE UNDERFLOW; OLD SIZE+CHANGE < 0

What happened The size change specified in the `SEGSIZE(STACK(+nnnn))` control caused the segment to become less than zero.

What to do Reinvoke the linker with the correct `SEGSIZE(STACK(+nnnn))` control.

WARNING 47: GROUP HAS NO CONSTITUENT SEGMENTS

What happened The group specified with the warning message has no segments and is not placed in the output object file. This error is often the result of a typographical error in the invocation line. The group is left out of the object file and processing continues.

What to do Unless there is a particular need for the specified group, no action is necessary.

WARNING 48: SIZE OF GROUP EXCEEDS 64K

What happened All of the segments that belong to the group specified with the warning message do not fit within the physical segment defined for that group. This

error is usually caused by misuse of the SEGSIZE control. The linker includes all segments in the object file and continues processing the input module. The output module will be executable, although addressing errors may occur.

What to do Examine the invocation line and reinvoke the linker using the SEGSIZE control more carefully.

WARNING 52: OFFSET FIXUP OVERFLOW

What happened While computing an offset from a base, the linker found that the offset was greater than 64K. This is a result of one of the segments of the group being outside the 64K frame of reference defined by its group base. The linker continues processing and the print file will be valid. The output file, however, with regard to the out-of-place segment, will not be usable.

What to do Modify the group definitions in your source file, retranslate and relink.

ERROR 55: ILLEGAL FIXUP

What happened While processing a fixup record, the linker found that the base for the reference and target are different. This is usually a coding error.

What to do Check your source carefully, retranslate and relink.

WARNING 58: NO START ADDRESS SPECIFIED IN INPUT MODULES

What happened The BIND control was specified, and none of the input modules has a start address. This indicates that the input list contains no main module. The CS and IP registers remain uninitialized, and their values are dependent on your system loader. The object module will be valid.

What to do Reinvoke the linker with a main module.

ERROR 60: OUTPUT FILE IS SAME AS INPUT FILE

What happened The pathname of the input file specified with the error message is identical to the output file pathname.

What to do Fix the duplicate-name situation and reinvoke the linker.

WARNING 64: PUBLIC SYMBOLS NOT SORTED DUE TO INSUFFICIENT MEMORY

What happened The number of public symbols in the input-list modules is too large for the linker to sort with available memory resources. The print file lists the public symbols in the order in which they were encountered in the input files. This condition has no effect on the correctness or validity of the output object module.

What to do Increase the amount of available RAM (for example, by unloading unneeded packages) or decrease the number of public symbols.

WARNING 65: ILLEGAL FIXUP: INCORRECT DECLARATION OF EXTERNAL SYMBOL

What happened The declaration of the symbol specified with the warning message was inconsistent with a corresponding public symbol definition and the linker could not resolve the reference. This condition is usually caused by an attempt to access absolute entry points from pre-located code without using the PUBLICONLY control explicitly. The linker internally converts these illegal fixups to legal formats to identify all occurrences in a single execution. Thus the output object module may not be correct, although it will be a valid 8086 object module.

What to do If the warning occurred because of an attempted access of absolute entry points from pre-located code, use the PUBLICONLY control in conjunction with the file that contains public definitions for those entry points.

WARNING 69: OVERLAPPING DATA RECORDS

- What happened The FASTLOAD control was specified, and two data records belonging to the same segment have offsets which make them overlapping. This is usually the result of a translation error, unless you have intentionally overlapped data records. The linker ignores the second record and does not include it in the output file. The code will be unusable.
- What to do If you want an overlap condition to exist, reinvoked the linker but do not use the FASTLOAD control. Otherwise, retranslate, then reinvoked the linker.

WARNING 71: TOO MANY MAIN MODULES IN INPUT

- What happened There are two or more main modules (modules with start address) in the input list. The linker uses the start address of the first main module it reads and ignores the others. The object code will be valid.
- What to do Make sure that the linker's interpretation is suitable to your objectives. If not, modify the input list and relink.

WARNING 72: REGISTER INITIALIZATION CODE EXISTS, NEW INITIALIZATION IGNORED

- What happened Because of a translation or linker problem, two or more initialization codes for 8086 registers were encountered in the input list. The linker uses the first initialization code and ignores the others. The object code will be valid.
- What to do If retranslating or relinking does not correct the error, contact the GRiD Customer Support Center.

WARNING 74: PRINT FILE SAME AS INPUT FILE

- What happened The pathnames of the print file and one of the input files are identical.
- What to do Fix the duplicate-name situation and reinvoked the

linker.

ERROR 75: PRINT FILE SAME AS OUTPUT FILE

What happened The pathnames of the print file and the output file are identical.

What to do Fix the duplicate-name situation and reinvoke the linker.

APPENDIX E. SOUND

The :Good Tune: and :Bad Tune: tokens in GRiDDDevelop let you define a sequence of notes that will be output to the speaker in the Compass computer. These tokens assume that you have the file named Sound~Device~ in the Programs subject of your system and that this device has been activated.

The :Good Tune: and :Bad Tune: tokens let you enter a text string following the token. The characters in this text string are interpreted according to the following rules:

| Character | Result |
|-----------|--|
| A to G | Plays the indicated note in the current octave. (See "0" below for octave control.) |
| #, +, - | Plays the preceding note (A through G) as a sharp note (# or +), or as a flat note (-). ^ti-12 |
| 00 to 06 | Octave. Sets the current octave for all notes that follow until another Octave command character is encountered. If no Octave is specified, the default is Octave 4. |
| > n | Go up to next higher octave and play note n (A-G). |
| < n | Go down to next lower octave and play note n (A-G). |
| L1 to L64 | Length. Specifies length of the note(s) that follow. L1 = whole note, L64 = 1/64th note. Default is 4 (quarter notes). |
| P1 to P64 | Pause for specified length. P1 = whole note, P64 = 1/64th note. |

T32 to T255 Tempo. Specifies number of beats (1/4 notes) per minute. Default = 120.
.
Dot or period. After a note, causes the note to be played as a dotted note (the length of the note is multiplied by 3/2).
V0 to V255 Volume. 0 = minimum, 255 = maximum. Default = 255.

INDEX

8086 registers, displaying in debug program, 5-7
 8087, Libraries, 3-3
 87null~Lib~, 3-3
 @SystemErrors file, C-2

A

absMem, debug program, 5-3
 Activate program, C-3
 Activating,
 devices, C-3
 modem, C-3
 serial, C-3
 Addresses, displaying in debug program, 5-9
 Alternate development approaches, A-1
 Alternate window, in debug program, 5-7
 Appending file kinds, 1-5
 Asm title suffix in GRiDDevelop, 2-13
 Assembler, reference manual, 3-1
 Assign value command, debug program, 5-10
 Assumeroot control, link program, 4-4, B-4
 Asterisk (*),
 debug prompt, 5-3
 wildcard character, C-2

B

Bad object file error, link program, D-7
 Bad Tune token, GRiDDevelop, 2-5
 Bad tune, E-1
 Bind, link control, 4-2, 4-4
 Break table entry, debug program, 5-5
 Breakpoints in debug program,
 clearing, 5-6
 setting, 5-5
 proceeding after, 5-7
 use in overlays, B-5

C

C title suffix in GRiDDevelop, 2-13
 Calculator, in GRiDDevelop CLI mode, 2-17
 Cannot find module warning, link program, D-7
 Cat (Catalog) program, C-4
 Catalog file, creating, C-5
 cel87~Lib~, 3-3
 Change source groups command, 2-17
 Change source groups form, in GRiDDevelop, 2-14
 Changing data files in GRiDDevelop, 2-19
 Changing memory contents, debug program, 5-11
 Check sum error, link program, D-3
 Clear breakpoint command, debug program, 5-6
 CLI, 2-17
 Clock chip, C-11

CODE-? command,
 debug program, 5-4
 in GRiDDevelop, 2-16
 CODE-B command, debug program, 5-5
 CODE-C command, debug program, 5-6
 CODE-C, command, GRiDDevelop, 2-17
 CODE-D command,
 debug program, 5-6
 in GRiDDevelop CLI mode, 2-17
 in Development Executive, A-2
 CODE-E command, debug program, 5-6
 CODE-G command, in GRiDDevelop, 2-8, 2-13, 2-17
 CODE-I command, debug program, 5-5, 5-6
 CODE-L command, debug program, 5-6
 CODE-M command, debug program, 5-7
 CODE-O command,
 debug program, 5-7
 GRiDDevelop Options command, 2-17
 CODE-F command, debug program, 5-7
 CODE-Q command, debug program, 5-7
 CODE-R command, debug program, 5-7
 CODE-SHIFT-ESC, causing breakpoints with, 5-6
 CODE-T command,
 debug program, 5-6
 Transfer command, in GRiDDevelop, 2-13
 CODE-W command, debug program, 5-2
 Com file kind, 1-6
 Com files, C-6
 Command files, 1-6, A-3, C-6
 examples of, A-3, A-4
 executing from File form, A-9
 executing from GRiDManager, A-9
 Command input error, link program, D-3
 Command line commands, debug program, 5-9
 Command line interpreter, in GRiDDevelop, 2-17
 Command lines, in GRiDDevelop Debug tokens, 2-5
 Command lines, terminating, A-2
 Command modifier characters, in GRiDDevelop, 2-16
 Command summary,
 debug program, 5-2
 GRiDManager, A-6
 Commands menu, in GRiDDevelop, 2-16
 Commands, debug program 5-4 - 5-9
 Commands, issuing to utility programs, C-2
 Comments, inserting in command files, A-4
 Common, unloading from command files, A-4
 Compact size control, compilers, 3-2
 CompactSystemCalls, 3-2
 Compare program, C-6
 Compile considerations, debug program, 5-7
 Compile form, in GRiDDevelop, 2-5
 Compile source files form, in GRiDDevelop, 2-14
 Compiler reference manuals, 3-1
 Compiler,
 Fortran, 3-1
 Pascal, 3-1
 PL/M, 3-1
 Compilers,
 invoking, 3-3
 invoking from command files, A-4
 invoking with GRiDDevelop, 1-3
 size controls, 3-1, 3-2

- ConFas.inc, 3-4
- ConPlm.inc, 3-4
- Control token, GRiDDevelop, 2-5
- Controls,
 - include, 3-4
 - size in compilers, 3-1, 3-2
- Conventions, file naming, 1-7
- Count, breaking at in debug, 5-5
- Creating catalog file, C-5
- Creating GRiDDevelop data files, 2-3
- CREF program, 4-1
- Cross reference program, 4-1
- CTRL-S, in CLI mode, 2-17
- Current location, displaying in debug program, 5-6
- Cursor, for development executive, A-2

D

- Data file for GRiDDevelop, 2-2, 2-3
 - changing, 2-19
- dcon87*Lib*, 3-3
- Deactivate program, C-6
- Debug command line commands, 5-9
- Debug commands,
 - assign value, 5-10
 - clear breakpoint (CODE-C), 5-5
 - display address, 5-9
 - display memory contents, 5-9
 - duplicate line (CODE-D), 5-6
 - examine/change memory, 5-11
 - executive (CODE-E), 5-6
 - info (CODE-I), 5-6
 - location display (CODE-L), 5-6
 - memory dump, 5-10
 - message display (CODE-M), 5-7
 - message display (CODE-M), 5-7
 - options (CODE-O), 5-7
 - proceed (CODE-P), 5-7
 - quit (CODE-Q), 5-7
 - register display (CODE-R), 5-7
 - tasks/semaphore display (CODE-T), 5-8
 - window toggle (CODE-W), 5-8
- Debug menu in GRiDDevelop, 2-6
- Debug program, 5-1
 - command summary, 5-2
 - compile considerations, 5-2
 - files, 5-3
 - Help command, 5-4
 - invoking, 5-2
 - link considerations, 5-2
 - prompt symbol, 5-3
 - set breakpoint command, 5-5
 - syntax, 5-3
 - link program, 4-8
- Debug token in GRiDDevelop, C-6
 - using command lines with, C-6
 - multiple, 2-6
- Debugger, invoking from GRiDDevelop, 2-6
- Debugging overlay programs, 4-5
- debugging tool set, debug program, 5-3
 - Decreasing size of segment warning, link program, D-10
 - Default menu, GRiDDevelop, 2-1
 - Default module, in debug, 5-7
 - Delimiter characters in file names, 1-4
 - Development approaches, alternatives, A-1
 - Development environment,
 - memory considerations, A-4
 - GRiDDevelop, 1-3
 - Development executive interface, 5-6
 - Development executive program, A-2, C-6
 - Development sequence, 1-1
 - Device driver, modem, C-8
 - Device level, in directories, 1-4
 - Devices,
 - activating, C-3
 - deactivating, C-6
 - Different publics for external error, link program, D-9
 - Different values for symbols warning, link program, D-4
 - Directory, typical, 1-4
 - Display address command, debug program, 5-9
 - Display memory contents command, debug program, 5-9
 - Display of variables, terminating in debug, 5-9
 - Display variable contents, debug program, 5-9
 - Displaying messages, in debug program, 5-7
 - Displaying tasks/semaphores in debug, 5-8
 - Do program, C-6
 - using with command files, A-3
 - DqLarge, 3-3
 - Dump program, C-7
 - Dumping memory contents, debug program, 5-10
 - Duplicate line command, debug program, 5-6

E

- Edit source file menu, GRiDDevelop, 2-2
- Editor, text, See GRiDWrite
- eh87*Lib*, 3-3
- Elapsed time program, C-7
- Enter token, in GRiDDevelop, 2-7
- Error messages,
 - link program, 4-9, D-1
 - system, D-1
- Errors,
 - halting on, 2-17
 - system file, C-2
- Examining log file entries, 2-10, 2-19
- Exclamation point (!), used in Cat program, C-5
- Executable files, 1-6
- Executing command lines from GRiDManager, A-6
- Executive command, debug program, 5-6
- Executive file, C-7
- Exit from debug program, 5-7
- Exit token, in GRiDDevelop, 2-7
- Extra register initialization warning, link program, D-8
- Extra start address ignored warning, link program, D-7

F

- `f86rn0~Lib^` - `f86rn4~Lib^`, 3-3
- Fastload, link control, 4-5, 4-8
- File form,
 - executing command files from, A-9
 - invoking applications from, 1-5
- File groups, 2-8
- File kinds, 1-5
 - list of, 1-6
- File names,
 - assumptions in GRiDDevelop, 1-5
 - conventions, 1-3
 - language identification in, 1-5
 - listing with Cat program, C-4
 - restriction in compilers, 1-5
- Files,
 - creating catalog, C-5
 - command, 1-6, A-3
 - comparing, C-6
 - debug program, 5-3
 - dumping, C-7
 - executable, 1-6
 - GRiDDevelop data, 2-2, 2-3
 - GRiDWrite, 1-6
 - include, 3-4
 - library, 1-6
 - list, 1-6
 - map, 1-6
 - object, 1-6
 - organizing, 1-3
 - printing lists and sources in GRiDDevelop, 2-19
 - run, 1-6
 - system, C-1
 - text, 1-6
- Form,
 - Change source groups, 2-14
 - Compile source files, in GRiDDevelop, 2-14
 - link in GRiDDevelop, 2-9
 - Print list files, 2-20
 - Print source files, 2-20
- Fortran compiler, reference manual, 3-1
- Fortran,
 - overlay example, B-5
 - run-time libraries, 3-3
- Ftn title suffix in GRiDDevelop, 2-13

G

- Good tune, E-1
- Good Tune token, in GRiDDevelop, 2-7
- GRiDDevelop
 - calculator mode
 - command line interpreter (CLi), 2-17
 - Commands menu, 2-16
 - compile form, 2-5
 - data file, 2-2, 2-3
 - data file, changing, 2-19
 - Edit source file menu, 2-2
 - file name assumptions, 1-5

- main menu, 2-1
- message line, 2-11
- overview of, 1-3
- predefined tokens, 2-4
- GRiDDevelop tokens,
 - Control, 2-5
 - Debug, 2-6
 - Enter, 2-7
 - Exit, 2-7
 - Link, 2-8
 - Listings, 2-9
 - Name, 2-11
 - Groups, 2-8
 - Log File, 2-10
 - Objects, 2-11
 - Prefix, 2-12
 - Print to, 2-12
 - Sources, 2-12
 - Test, 2-15
- GRiDManager,
 - commands, A-6
 - adding devices from, C-3
 - executing command files from, A-9
- GRiDWrite files, 1-6
- GRiDWrite,
 - creating command files with, A-3
 - creating source files with, 1-2
 - executing command files from, A-9
 - invoking from GRiDDevelop, 2-3
- Group enlarged warning, link program, D-5
- Group has no constituent sements warning, link program, D-11
- Group names for source files, 2-13
- Groups token, in GRiDDevelop, 2-8

H

- Halting on errors, 2-17
- Help command, debug program, 5-4
- hexConstant, debug program, 5-3

I

- I/O error, link program, D-2
- iAPX 86,68 Utilities User's Guide, 4-1
- Illegal fixup error, link program, D-11
- Illegal fixup: Incorrect declaration warning, link program, D-12
- Illegal overlay construction error, link program, D-9
- Illegal use of overlay control error, link program, D-8
- Improper fixup warning, link program, D-5
- Improper start address warning, link program, D-6
- Include control statements, examples, 3-5
- Include files, 3-4
 - listing with source files, 2-13
 - naming conventions, 1-5
 - organizing, 1-4
 - Pascal, 3-4
 - PL/M, 3-4
- Info command, debug program, 5-6
- Input phase error, link program, D-2

- Input/output routines,
 - GRiD-DS, 3-2
 - Pascal, 3-2
- Insufficient memory error, link program, D-4
- Intel compiler name restrictions, 1-5
- Invalid syntax error, link program, D-6
- Invocation examples, link program, 4-2
- Invoking compilers, 3-3
 - from command files, A-4
 - with GRiDDevelop, 1-3
- Invoking GRiDWrite from GRiDDevelop, 1-3, 2-3
- Invoking the debug program, 5-2
 - in GRiDDevelop, 2-6
- Invoking the Link program, 4-1
 - in GRiDDevelop, 2-8

K

Kind, see File kinds

L

- Language identification in file names, 1-5
- Large size control, compilers, 3-2
- LargeSystemCalls, 3-2
- Lib file kind, 1-6
- Librarian program, 4-1
- Librarian, 1-6
- Libraries, 3-2
 - 8087, 3-3
 - CompactSystemCalls, 3-2
 - Fortran, 3-3
 - LargeSystemCalls, 3-2
 - organizing, 1-4
 - Pascal run-time, 3-2
- Library not allowed error, link program, D-8
- Line#, debug program, 5-3
- Link considerations, debug program, 5-2
- Link controls,
 - Assumeroot, 4-4
 - Bind, 4-2, 4-4
 - Fastload, 4-5, 4-6
 - Map, 4-5
 - Name, 4-6
 - Overlay, 4-4, 4-6
 - Print, 4-7
 - Printcontrols, 4-7
 - Purge, 4-5, 4-8
 - Segment Size, 4-2, 4-8
 - summary of, 4-3
- Link form in GRiDDevelop, 2-9
- Link invocation examples, 4-2
- Link map, 4-5, 4-9
- Link program, 4-1
 - assumeroot control example, B-4
 - error messages, 4-9
 - invoking from command files, A-4
 - invoking, 4-1

- overlay control example, B-4
- print file, 4-5, 4-7, 4-9
- print file, 4-9
 - syntax, 4-1, 4-2
 - warning messages, 4-9
- Link statements, terminating in GRiDDevelop, 2-8
- Link token, in GRiDDevelop, 2-8
- Link tokens, multiple, 2-8
- Link warning messages, D-1
- Link86 error, link program, D-5
- Linker map files, 1-6
- Linker program, invoking, 1-2
 - from GRiDDevelop, 2-8
- Linking overlays, B-3
- List files, 1-6
 - printing, 2-19
- Listing titles with the Cat program, C-4
- Listings token in GRiDDevelop, 2-9
- Load program, C-8
- Load-time-locatable module, 4-4
- Location display command, debug program, 5-6
- Log file token, in GRiDDevelop, 2-10
- Log files, examining, 2-10, 2-19
- Lst extension, appending by compilers, 2-9
- Lst file kind, 1-6

M

- Main menu, GRiDDevelop, 2-1
- Manuals, compiler reference, 3-1
- Map files, 1-6
- Map, link control, 4-5, 4-9
- Modules, executable, 1-2
- Memory,
 - assigning values in debug program, 5-10
 - considerations, in development environment, 4-4
 - contents, examine/change in debug, 5-11
 - displaying contents in debug program, 5-9
 - usage, by a file, C-11
- Memory dump command, debug program, 5-10
- Menu,
 - Debug in GRiDDevelop, 1-6
 - GRiDDevelop commands, 2-16
 - GRiDDevelop default, 2-1
 - GRiDDevelop main, 2-1
 - GRiDDevelop Transfer, 2-18
 - GRiDDevelop's Edit Source File, 2-2
- Message display command, debug program, 5-7
- Message line, displaying in GRiDDevelop, 2-11
- Messages,
 - link errors, 4-9
 - link warnings, 4-9
 - warning, D-1
- Modem,
 - activating, C-3
 - device file, C-8
- Modules,
 - debug program, 5-3
 - Fortran, 3-3
 - linking of objects, 1-2
 - Pascal, 3-2
- MP1 file kind, 1-6, 4-5, 4-7, 4-9

N

Name,
 assumptions in GRiDDevelop, 1-5
 link control, 4-6
 token, example of use, 2-11
 token, in GRiDDevelop, 2-11
Names, adding language identification to, 1-5
Naming conventions,
 for include files, 1-5
 for source files, 1-5
Naming files, 1-4
Naming restrictions in compilers, 1-5
No start address specified warning, link program, D-12
Not an object file error, link program, D-7

O

Obj extension, appending by compilers, 2-11
Obj file kind, 1-6
Object file size, 4-5
 reducing, 4-5
Object files, 1-6
 organizing, 1-4
Object modules, linking, 1-2
Objects token, in GRiDDevelop, 2-11
Offset fix-up overflow warning, link program, D-11
Options command,
 debug program, 5-7
 in GRiDDevelop, 2-17
Organizing files, 1-3
OsOverlay procedure, B-1
OsPascProcs.Inc, 3-4
OsPascTypes.Inc, 3-4
OsPimProcs.Inc, 3-4
OsPimTypes.Inc, 3-4
Output file is same as input file error, link program,
Overlapping data records warning, link program, D-13
Overlay,
 Fortran example of, B-5
 link control, 4-4, 4-6, B-4
 Pascal example of, B-2
Overlays, B-1
 additional considerations, B-4
 breakpoints, B-4
 debugging, B-4
 linking, B-3

P

Pas title suffix in GRiDDevelop, 2-13
Pascal,
 include files, 3-4
 overlay example, B-2
 reference manual, 3-1
 run-time libraries, 3-2

PL/M compiler, reference manual, 3-1
Plm title suffix in GRiDDevelop, 2-13
PLMLits.inc, 3-4
Possible overlap warning, link program, D-6
Predefined tokens in GRiDDevelop, 2-4
Prefix program, C-6
Prefix token, in GRiDDevelop, 2-12
Preprogrammed softkeys, C-9
Print file same as input file warning, link program, D-14
Print file same as output file error, link program, D-14
Print file, link program, 4-5, 4-7, 4-5
Print list files form, 2-20
Print source files form, 2-20
Print to token, in GRiDDevelop, 2-12
Print, link control, 4-7
Printcontrols, link control, 4-7
Printing files in GRiDDevelop, 2-19
Proceed command, debug program, 5-7
procName, debug program, 5-3
Program, Pascal declaration, 3-2
Programming softkeys, C-9
Programs subject, C-1
Prompt symbol, for development executive, A-2
Public records, link program, 4-8
Public symbols not sorted warning, link program, D-12
Purge, link control, 4-5, 4-6

Q

Question mark (?),
 GRiDDevelop command modifier, 2-16
 used in Cat program, C-5
Quit command, debug program, 5-7

R

Read, Pascal procedures, 3-2
Register display command, debug program, 5-7
Register initialization code exists warning, link program, D-17
Register, debug program, 5-3
Restrictions on file names in compilers, 1-5
Root file, 4-4
Root modules, B-1
Run file kind, 1-6
Run-time libraries,
 Fortran, 3-3
 Pascal, 3-2

S

Same overlay name error, link program, D-9
Segment,
 combination warning, link program, D-3
 overflow warning, link program, D-6
 size overflow error, link program, D-10
 size underflow error, link program, D-10

Segment Size, link control, 4-2, 4-8
 Semaphores, displaying in debug program, 5-8
 Semicolon (;),
 GRiDDevelop command modifier, 2-16
 using in command files, A-4
 Sequence, development, 1-1
 Serial, activating, C-3
 Set breakpoint command, debug program, 5-5
 Size controls, compilers, 3-1, 3-2
 Size of group exceeds 64K warning, link program, D-11
 Size, stack segment, 4-8
 Softkeys,
 file, C-9
 multiple files, C-10
 preprogrammed, C-9
 Sound, E-1
 Source file groups, 2-8
 Source files,
 compiling, 1-2
 creating, 1-2
 editing from GRiDDevelop, 2-2
 naming conventions, 1-5
 organizing, 1-4
 printing, 2-19
 Source group names, in GRiDDevelop, 2-13
 Source groups, changing, 2-17
 Sources token,
 in GRiDDevelop, 2-12
 interaction with Listings token, 2-9
 Specified segment not found error, link program, D-10
 SS (see Segment Size)
 Stack overflow error, link program, D-5
 Stack segment, 4-8
 Status program, C-10
 Subject level, in directories, 1-4
 Suffixes, titles in GRiDDevelop, 2-13
 Summarize program, C-11
 Summary of commands, debug program, 5-2
 Summary of link controls, 4-3
 Symbols, resolving during overlay links, E-3
 Syntax,
 debug program, 5-3
 system utilities, C-1
 System errors file, C-2

T

Tasks/semaphore display command, debug program, 5-8
 Terminating link statements, in GRiDDevelop, 2-8
 Test menu, in GRiDDevelop, 2-15
 Test token, in GRiDDevelop, 2-15
 TestName token, in GRiDDevelop, 2-15
 Text editor, see GRiDWrite
 Text file kind, 1-6
 Time program, C-11
 Time, elapsed, C-7
 Title level, in directories, 1-4
 Title suffixes in GRiDDevelop, 2-13

Titles,
 listing with Cat program, C-4
 naming, 1-4
 Toggling windows in debug, 5-8
 Tokens in GRiDDevelop, 2-4
 Bad tune, 2-5, E-1
 Control, 2-5
 Debug, 2-6
 Enter, 2-7
 Exit, 2-7
 Good Tune, 2-7, E-1
 Groups, 2-8
 Link, 2-8
 Listings, 2-9
 Log File, 2-10
 Name, 2-11
 Objects, 2-11
 Prefix, 2-12
 Print to, 2-12
 Source group names, 2-13
 Sources, 2-12
 Test, 2-15
 user-defined, 2-15
 Too many overlays error, link program, D-9
 Transfer menu, in GRiDDevelop, 2-18
 Tune,
 bad, 2-5, E-1
 good, 2-7, E-1
 Two many main modules warning, link program, D-13
 Type description too long error, link program, D-6
 Type mismatch warning, link program, D-3
 Typical directory, 1-4

U - Z

Unload program, C-12
 Unloading common, from command files, A-4
 Unresolved symbols warning, link program, D-4
 User-defined tokens, in GRiDDevelop, 2-15
 Utilities, iAPX 86/88, 4-1
 Utility programs, C-1
 wildcards in, C-2
 Variables,
 assigning values in debug program, 5-10
 displaying contents of in debug, 5-9
 varName, debug program, 5-3
 Warning messages, link program, 4-9
 Wildcards in utility programs, C-2
 Window toggle command, debug program, 5-8
 Window, alternate in debug program, 5-7
 Work program, C-12
 Write, Pascal procedures, 3-2
 ZZZDEBUG files, 5-3