Date: November 14, 1985

To: R & D personnel

From: Bill Huber, Leonid Shvarts

Subject: Ring Zero Debugger User Manual

Reference: None

Keywords: Ring Zero Debugger, Debug, Tools

Abstract

This document describes the functionality of and uses for a new tool called the Ring Zero Debugger. The Ring Zero Debugger is an assembly language debugger that allows one to control program execution of Primos code on a real-time, interactive basis. Using this tool, one can effectively suspend the state of the entire system, examine or change nearly any part of the system, and then resume execution transparently. The most important feature of the Ring Zero Debugger is the ability to set breakpoints nearly anywhere in shared code. Despite its name, the Ring Zero Debugger can be used to set breakpoints in either ring.

The primary users of the Ring Zero Debugger are expected to be engineers working on either Primos or Primenet. However, it may prove helpful for other tasks as diverse as debugging new hardware to debugging shared subsystem code.

Table of Contents

1. Introduction

Before discussing details of the operation of the Ring Zero Debugger, some overview of the debugger and this document should be helpful. This chapter will give a brief statement concerning the motivation for implementing the debugger. Next it will describe the debugger in very general terms. Finally this chapter will discuss the format of the rest of the document.

1.1 Motivation

The only methods that exist for debugging Primos ring zero code without the use of the Ring Zero Debugger are very crude. These methods were common two decades ago but are an extremely poor way to do program development today. These methods include putting halt instructions directly in the code and taking tape dumps, or modifying the code to print values of variables on a console. It should be easy to see that this approach has numerous problems. Foremost among them is the large amount of time needed to fully debug even a small piece of code. In general, the need for a program debugger for efficient program development is fairly obvious.

To address this situation, a new tool known as the Ring Zero Debugger has been written. The primary goal of this tool is to increase the productivity of Prime engineers developing code in ring zero. Because of the way that the debugger has been implemented, this increase in productivity should be true for ring three shared code as well. The claim for increased productivity is based on two assumptions. One is that the program development time will be much shorter due to the interactive nature of the debugger. The other is that new or modified code will be more reliable due to the ability to more fully check all code paths and simulate error conditions.

1.2 Ring Zero Debugger High-level Description

The Ring Zero Debugger is an assembly language, system-level debugger. The only assembly language that it supports is Prime V-mode. The Ring Zero Debugger can run on any Prime processor which supports V-mode (including dual processors, such as the P850). The debugger is described as a system-level debugger since it stops the entire system when entered, not just a single process. It also has commands which relate to the system as a whole, in addition to process-specific commands.

As with any debugger, the most significant feature in the Ring Zero Debugger is the ability to set and clear breakpoints. The debugger allows users to set breakpoints nearly anywhere in both ring zero and ring three shared code. Breakpoints can specify a particular process in the system or they can be for any process. The ability to single step through code is also provided.

The other features available with the Ring Zero Debugger are based primarily on three other tools. These other tools are DBG (Prime's Higher-level-language Debugger), Autopsy (an internal tool for analyzing crash dumps), and VPSD (a crude assembly langauge debugger for V-mode). Some of the other features include the ability to reference memory or registers of any process in the system, the ability to completely examine a process's state, the ability to examine certain system data bases, the ability to examine local program variables by name, and the ability to translate addresses from virtual to physical and vice versa. Much of the information is displayed symbolically based on symbols in the Primos load maps.

The debugger is built into Primos as part of the ring zero load. Thus the debugger and its sources will reside on the Master Disk as part of Primos and will be shipped to customers. However, the debugger is not a product and will not be documented in any Prime manual. The only place that the procedure for configuring and entering the debugger is described, is in this document which is only available to Prime personnel. Thus the debugger will exist in Primos at customer sites but most of them will not know it's there or how to use it.

1.3 Document Format

The main purpose of this document is to describe the functionality of the Ring Zero Debugger so that a person unfamiliar with it can learn how to use it. This document assumes the reader has some familiarity with Prime processor architecture and also with some Primos internals. If this is not the case, some background reading may be helpful.

As regards the format of the rest of the document, chapter 2 describes the most basic functions of the debugger. It covers all aspects of the debugger environment and then introduces a few elementary commands. Chapter 3 describes the complete functionality of the debugger by functional groupings. All commands in the debugger are described here. Chapter 4 gives some examples of the way that the debugger can be used to solve various problems. Finally chapter 5 describes some implications of the debugger design that have user-visible effects. All but the most casual user of the Ring Zero Debugger should read this chapter since it gets to the heart of some of the limitations that the debugger has.

In addition to the basic chapters, there are a number of appendices that can be used as a reference for various details. These include information such as finding variables from a program listing, the exact syntax of the debugger commands, the syntax of the assembly langauge, descriptions of error messages, a summary of debugger functional limitations, and notes on maintaining the debugger.

2. The Basics of Using the Ring Zero Debugger

The purpose of this chapter is to introduce users to the basics of the Ring Zero Debugger. Its role is to provide one with enough knowledge to be able to use some of the most fundamental commands. Included in this chapter are descriptions of how to configure and invoke the debugger, how to reference addresses, what some simple commands can be used for, and how errors are reported.

2.1 A Simple Example

Before discussing the details of the debugger, this section will present a simple example of what a small debugging session might look like. This example will show the commands that one might issue to step through a small section of code to verify that it works correctly. The reason for showing the example is not so all the commands will be understood, but rather to start to get a feel for the type of abilities that one has with the debugger.

The small section of code is an actual Primos routine. It is written in assembly language and is shortcalled. The purpose of this routine is to increment a pointer by the size of a page map entry and return that pointer. (The size of a page map entry varies depending on the type of processor.) The name of this routine is pgstep and a segment of its listing is shown below.

```
001351: 011415.000012S (1289) PGSTEP    STL  ARGPTR
001353: 045435.000012S (1290)           LDL  ARGPTR,*
001355: 015414.002122  (1291)           ADL  PME_LENL
001357: 003403.000000X (1292)           JMP  XB%
```

One can see from this listing segment that the routine is very simple. It picks up the argument (a pointer), adds the size of a page map, and returns with the new value in the L register. In the following sequence, the debugger will be entered, a breakpoint will be set at pgstep, the debugger will be exited, and finally a Primos interrupt process will hit the breakpoint causing the debugger to be reentered.

```
\
Debugger entered due to console interrupt.
  Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4)
-> breakpoint pgstep
-> listall
Type   Address         Procedure       Process  Count  Mnemonic
brkpt  6(0)/50143      PGSTEP            Any        1    STL
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process -9 (PNCPCB) was executing at 6(0)/50143 (PGSTEP).
->
```

Now that we have hit the breakpoint, we can examine the code to assure ourselves that we

are at the right place. The code looks like what one would expect so we also try single stepping through two instructions.

```
-> access .
6(0)/50143  STL% SB%+ 12
6(0)/50145  LDL% SB%+ 12 ,*
6(0)/50147  ADL%  50714
6(0)/50151  JMP% XB%+ 0   ?
-> step

Debugger entered due to breakpoint/single step.
  Process -9 (PNCPCB) was executing at 6(0)/50145 (PGSTEP + 2)
-> step

Debugger entered due to breakpoint/single step.
  Process -9 (PNCPCB) was executing at 6(0)/50147 (PGSTEP + 4)
->
```

The next step is to examine the value of the input pointer (which at this point is in the L register) and the value of the page map entry size. The entry size of 1 is correct for the current processor, so we can move on and see that the addition takes place correctly by examining the L register after the addition.

```
-> access_register l
L (high order): 000600
L (low order): 057411
-> access_type octal
-> access 6/50714
6(0)/50714  000000
6(0)/50715  000001  ?
-> step

Debugger entered due to breakpoint/single step.
  Process -9 (PNCPCB) was executing at 6(0)/50151 (PGSTEP + 6)
-> access_register l
L (high order): 000600
L (low order): 057412
->
```

We have now seen that the pointer was incremented correctly. If it hadn't been, we could have modified the value in the L register and let the program continue. One last thing we might be curious about is where the routine was called from. A simple way to determine this is to single step one more time. After doing this we see that pgstep was called from the routine mapio. Finally we delete all breakpoints and leave the debugger.

```
-> step

Debugger entered due to breakpoint/single step.
  Process -9 (PNCPCB) was executing at 11(0)/16760
  (MAPIO + 157).
-> clearall
-> continue
Leaving the debugger.
```

2.2 Terminology

Before describing the Ring Zero Debugger, it is useful to define a few terms which have taken on special meanings when used to discuss different features of the debugger.

active process — Many debugger commands assume a given process if one is not specified explicitly. The process assumed is called the active process. When the Ring Zero Debugger is entered, the active process is set to the process that was pre-empted to run the debugger. The active process can be changed by a Ring Zero Debugger command.

original process — This refers to the process which caused the Ring Zero Debugger to be entered. This notion is significant since single steps operate on the original process.

program variable — In the context of the Ring Zero Debugger, program variables refer to those user-defined names found in various programs. The debugger provides the capability to allow a user to manually define names from local programs and later use these names to display values of the data objects.

symbol — In the context of the Ring Zero Debugger, the term "symbols" refers to those object names found only in the Primos load maps. (both ring 0 and ring 3)

2.3 Data Type Conventions

A common source of confusion with some programs (e.g. Autopsy) is what radix or number system an item is printed in. An attempt has been made throughout the Ring Zero Debugger to display all numbers of the same radix in the same format. The conventions are the following. Single-precision octal numbers are printed as 6 zero-filled octal digits. Double-precision octal numbers are printed as 11 zero-filled octal digits. Single-precision decimal numbers are printed as 5 blank-filled decimal digits. Double-precision decimal numbers are printed as 10 blank-filled decimal digits. Hexadecimal numbers are printed as 4 zero-filled hexadecimal digits.

In terms of particular objects, there are also conventions about the radix used. Process numbers and single step or breakpoint counts are always in decimal. Addresses and contents of registers are always referenced as octal numbers. Offsets from addresses are also assumed to be in octal. Appendix B indicates the radix used in each command should there be any confusion.

2.4 Referencing Addresses

A large number of commands in the Ring Zero Debugger require an address as an argument. For ease-of-use, the debugger allows a number of different ways of specifying an address. The term applied to the different forms the addresses can take is an address-expression. A similar but different form is used just with breakpoint commands. It is called a

break-expression.

2.4.1 Address-expression's

An address-expression can take on 3 different forms. The most obvious is specifying a virtual address directly as in 6000/16100. In this form the segment number and word offset must be specified separated by a /. There is no ring number. One can also specify an offset in octal from the given virtual address as in 6/35120+100 or 15/3200-35.

Another form of address-expression allows an address to be specified as a relative offset from global symbols found in the Primos load maps. If the symbol refers to a procedure name (as opposed to a "common" or "other" symbol type), the value used for the symbol is the starting address of the routine. If the symbol is a "common" or "other" symbol type, the given address is used. The format for symbolic address-expression's is a symbol followed by either a + or - and then the octal offset as in PRWFSS+200.

The last form of address-expression permits addresses as offsets from the base registers. The allowable base registers are known as SB%, LB%, XB%, and a special symbol *. The first 3 base registers refer to the same registers one would specify while writing in PMA. An address is formed by taking the contents of the specified base register for the active process and adding or subtracting the optional offset. The last form, *, refers to the current contents of the original process's program counter. The base register form is useful when examining assembly language where the operands are expressed in terms of base registers, e.g. LDA LB%+412. Some examples of valid address-expressions are shown below.

```
6/35043
6003/1200-100
prwfSS+200
pudcom
sb%+12
*
*+100
```

For a more formal definition of an address-expression, see appendix B.

2.4.2 Break-expression's

The breakpoint commands require arguments similar to address-expression's. These arguments are known as break-expression's. They permit the same virtual address and symbolic forms as do the address-expression's but they do not allow the base register relative form (except for *). Another difference with break-expression's is that they can be used to specify which process should be breakpointed. The breakpoint command will be described later but it should be noted at this point that a breakpoint can be set for either a particular process or for all

processes. To specify a particular process, say process 5, one might say:

```
breakpoint 5:prwfSS+1
```

To specify that any process which encounters the same breakpoint should trap, one would say:

```
breakpoint prwfSS+1
```

Some further examples should demonstrate the different types of allowable break-expression's.

```
breakpoint 6/50212+20
breakpoint 22:15/3120
breakpoint prwf$$+1
breakpoint 230:prwf$$+222
```

For a more formal definition of a break-expression, see appendix B.

2.5 The Active Process

Because the Ring Zero Debugger is a system debugger, it is important for it to be able to examine or change the state of any process in the system. To permit this ability yet retain a simple command structure, many debugger commands assume a certain process. This process is known as the active process. This means that if a command references a process's registers or a process's private address space, the process that it uses is the active process.

The active process is determined by the way in which the debugger is entered but can be changed by a debugger command. If the debugger is entered at coldstart due to a sense switch setting, the active process is process number 1. If the debugger is entered due to a breakpoint or single step, the active process is the process which encountered the breakpoint or single step. If the debugger is entered due to a console interrupt, the active process is the process which owned the "last" register set when the clock process was servicing the system console. (This implies the master cpu on the P850.) The active process can be examined and changed by the lookat command. This command is described in the following section and in other chapters as well.

2.6 The Effect of Entering the Ring Zero Debugger

There are a number of different ways to enter the Ring Zero Debugger but all of them have the same effect on the state of the system, namely it will appear to be suspended. While in this state, no hardware interrupts will be serviced and no processes other than the Ring Zero Debugger process will be executing. Even phantom interrupt code will not be executed since hardware device interrupts are disabled. However, any DMX in progress will continue. Only the debugger process will be executing and it will not relinquish control to Primos until the debugger is exited.

One of the more visible signs of suspending Primos by invoking the Ring Zero Debugger is that time will be suspended. This is due to the fact that not even the clock process will run while in the debugger. Therefore the system time-of-day will be off by the amount of time spent in the debugger. Without the clock process running there will also be no basis for local devices to timeout. Thus when one leaves the debugger, all device i/o will continue normally. However, a system that has invoked the debugger will probably appear down to other systems on the network, depending how long one is in the debugger.

2.7 Operational Procedures

Before discussing the specifics of commands in the Ring Zero Debugger, one must know how to get the debugger, how to configure it, how to enter it, and finally what the command environment looks like inside the debugger. These topics are discussed in this section.

2.7.1 Getting the Ring Zero Debugger

The Ring Zero Debugger is built into Primos as part of the ring zero load. This happened at Primos revision 20.1 and will be true of any later version. Everything that the debugger needs is already built into Primos. Thus one need only have a recent enough version of Primos and having the debugger is guaranteed.

2.7.2 Configuring the Ring Zero Debugger

In order to provide the various functions described in this document, the Ring Zero Debugger must use a large amount of wired memory. In fact the entire debugger must be both coldstart resident and wired. This does not make much difference to an engineer debugging code in the lab but it certainly makes a difference to Prime customers who are interested in performance and throughput. Thus one of the features of the debugger is the ability to be configurable. If the Ring Zero Debugger is not configured, no wired memory will be used just for the debugger. In fact, there will be no discernable effect of any kind on the system. The means for communicating configuration information to the debugger is through the processor sense switches.

The processor sense switches are set by giving an argument to the boot command when the system is first coldstarted. A common sense switch setting is 14114. With the addition of the Ring Zero Debugger to Primos, there are now two newly defined bits. One bit, bit 2, will cause the Ring Zero Debugger to be configured. The other bit, bit 3, will cause the debugger to be entered during coldstart code. Thus, to come up in the debugger when coldstarting, a typical sense switch setting would be 74114, whereas just configuring the debugger would be 54114. (The illogical combination of bits which indicates "enter the debugger during coldstart but don't configure the debugger" is ignored.)

2.7.3 Ways to Enter the Ring Zero Debugger

There are three different ways to enter the Ring Zero Debugger. It can be entered by Primos initialization code during coldstart, by issuing a special key sequence on the system console, or by encountering a previously set breakpoint (or single step).

The way to enter the Ring Zero Debugger during coldstart is to boot the system with the previously discussed sense switch settings. This will cause the debugger to be entered the first time process exchange has been turned on. This is important because it means that nearly all Primos coldstart initialization code can be debugged using the Ring Zero Debugger. The only code that can't be debugged during coldstart, besides the boot program on disk, is the coldstart code before process exchange is turned on.

Another, more common way to get into the Ring Zero Debugger will be to just issue a special key sequence on the system console. This special sequence is control-c backslash or '^c\'. It can be issued at any time and should cause the debugger command level to be entered. This is known as a console interrupt. (If one needs to input the sequence '^c\' to user 1 while the debugger is configured, one can type '^c^c\'. This will not cause the debugger to be entered but will put '^c\' in the user 1 input buffer.) If for any reason there is outstanding or unprocessed input to the debugger when it is entered with a console interrupt, the input will be ignored.

2.7.4 The Command Environment

The only way to communicate with the Ring Zero Debugger is through the system console. Communicating with the debugger will not affect regular Primos user 1 operations since Primos is basically suspended while the debugger is running.

The Ring Zero Debugger does not change the baud rate of the system console. Thus whatever rate you have it set for in Primos (using the asrate config directive) will be the rate you will see while in the debugger. The exception to this is when the debugger is entered during coldstart before the config file is read. In this case the baud rate of the system console depends on the boot program on the disk.

The Ring Zero Debugger will use the system default erase and kill characters. These values can be set by config directives (erase and kill). The system supplied defaults are " for the erase character and ? for the kill character.

The debugger also supports xon-xoff and quits. A control-s stops output and a control-q resumes output. A control-p will cause a quit to happen in the debugger. A quit will mean the currently executing operation will be aborted and the debugger will return to command level. A quit also causes the command buffer for the debugger to be emptied, in the event that there are other, not-executed commands in it. Unlike Primos quits, the command cannot be restarted. With xon-xoff and especially quits, it should be noted that the quit or xoff may

not always appear to be working immediately, especially at low baud rates. This has to do with the fact that there is a buffer on the VCP that must empty before console input can be read.

One other point should be made about xon-xoff. While executing in the debugger, all echoing of characters is done by the VCP, not the debugger. This means that when one types xoff on the terminal, it is always echoed back to the terminal. This can create problems if the terminal treats an echoed xoff as a flow-control character and locks up the terminal keyboard. This happens on PST100 and PT45 terminals but not on a GE Terminet or a Perkin Elmer Fox. The solution to the problem on a PST100 is to use the "pause" key instead of control-q and control-s. There is no known way to get around this problem on a PT45.

Some final points about the command environment within the debugger concern the command line. A command line can only be 1 line long. However, multiple commands can be stacked on the same line by separating them by a ;. There are no continuation characters to extend debugger input beyond one line. The maximum length of a line is 256 characters.

2.8 Basic Commands

Having discussed the debugger command environment, this next section will describe how to use some of the most basic commands. A more thorough discussion of these and other commands is given in the next chapter.

2.8.1 Entering and Leaving the Ring Zero Debugger

The most common means of entering the Ring Zero Debugger is to use a special key sequence as described earlier. Typing this key sequence on the system console will immediately invoke the debugger, thereby suspending Primos. Regardless of the way the debugger is entered, it will always print a line indicating that the debugger has been entered and stating the reason for it having been entered. Then it will print the unique debugger prompt, ->, and wait for a debugger command.

Ring Zero Debugger commands perform a variety of functions but only two of these commands cause Primos to resume execution. One of these, the step command, will be discussed in the next section. The other command is continue. The continue command, abbreviated as 'c', causes the debugger to stop execution and thus allows Primos to resume just where it was interrupted. The following example shows what would be seen at the system console when the Ring Zero Debugger is entered and then left.

```
OK,\
Debugger entered due to console interrupt.
 Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> continue
Leaving the Debugger.
stat me
User                              No  Line Devices
SYSTEM                             1   asr <LABSYS>

OK,
```

In this example, the user enters the debugger by typing control-c \. This causes the debugger to print both the banner lines and prompt, and then wait for input. The debugger always indicates which process was interrupted by the debugger and where this process was executing. At this point the user types "continue" to leave the debugger. The debugger indicates that it is being exited and at that point Primos resumes execution. One should note that there is no additional prompt from user 1 when Primos begins to run again. This is due to the fact that the execution of the debugger is completely invisible to Primos. User 1 is completely ignorant of the fact that the debugger has been entered and debugger commands may have been issued.

2.8.2 The Status Command

An important part of debugging is determining the current state of a process. A process's state consists of the process's address space and registers. The status command can be used to examine important parts of a process's state. The easiest form of this command takes no arguments. This form will display information about the active process. An example of this command is shown below after encountering a breakpoint at location pagtur+1.

```
Debugger entered due to breakpoint/single step.
    Process 3 was executing at 6(0)/45205 (PAGTUR + 1).
-> status
Process 3  WSH                    *** Owns register set 1 ***
   Level: Priority 1 user
   Type:  Normal terminal user
   State: Ready
   PB:    6(0)/45205 (PAGTUR + 1)
   LB:    6(0)/46002 (PAGTUR)
   SB:    6000(0)/1264       XB:     6(0)/55534
   L:     000010 100077      E:      000000 000000
   X:     000000             Y:      177777
   FAR0:  000000 000000      FLR0:   000000 000000
   FAR1:  000000 000000      FLR1:   000000 000000
   Keys:  034100             Modals: 100077
   Fcode: 045206 000040      Faddr:  6(0)/1703

->
```

The status command first indicates the process number and, if available, the login name of the process. If the process is an interrupt process, it will have a process number of less than 1. (Interrupt processes are a special form of process used to service devices such as disks and

terminals, or for other special purposes.) Interrupt processes have there own special names, such as DK1PCB for the first disk interrupt process. If the process happens to own a register set. (as opposed to just owning a pcb), this will also be indicated.

The next three lines printed by the status command convey information about the specific process being examined. Level refers to the ready list level. Type refers to the process type. Examples of different types are phantom, NPX slave, network process, etc. Interrupt processes do not show a type. Finally the state indicates whether a process is on the ready list or waiting on a semaphore.

The remaining lines of the status command output show register values. All values are shown in octal. The base registers are printed as pointers. The names that follow the PB and LB register correspond to names found in the Primos load maps by looking up the values of the given register. In the case of the PB or program counter, the value is an educated guess at which routine and offset the program is executing in. In the case of the LB, a match always indicates that the process is in the named routine, except when this routine shortcalls another routine.

2.8.3 Accessing Memory

A crucial ability of the Ring Zero Debugger is the ability to examine and change memory. The most straight-forward way of doing this is to use the access command. This command allows one to examine any resident memory in the system in a variety of formats. The format assumed in the access command is set by using the access_type command.

The access_type command allows one to examine memory with the access command in 6 different formats. One can reference memory as ascii text, as a bit string, as a decimal, hex, or octal number, or, lastly, as V-mode assembly language. The corresponding arguments to the access_type command to set the assumed type are ascii, bit, decimal, hex, octal, and symbolic. These can be abbreviated as a, b, d, h, o, and s. The current access_type can be determined by issuing the access_type command with no argument.

Having set the access_type, one can now examine memory by using the access command. It takes any address_expression as an argument. The command will cause the specified location of the active process's address space to be displayed in the access_type format. At this point one may enter a new value for the accessed location. Whether this is done or not, the debugger expects a special terminator character which indicates one of the following: examine the next location in sequence (carriage return), examine the previous location in sequence (^), or abort the access command without changing the current location (? or /).

One other command which should be noted is the lookat command. The access command is set up so that address_expression's that reference a private address (e.g. 4000/100), assume that it is the private address space of the active process. However, if one wants to examine the

private address space of a process other than the current active process, the active process must be changed. This is the function of the lookat command. It takes a process number as an argument and makes this process the new active process.

An example should help clarify the way access_type, access, and lookat are used. In this example, the access type is set to octal and a sequence of numbers is corrected. Next the private address space of process 5 is examined.

```
-> access_type
Current access type is symbolic.
-> access_type octal
-> access_type
Current access type is octal.
-> access 6/5000
6(0)/5000  1
6(0)/5001  2
6(0)/5002  4
6(0)/5003  4 ↑
6(0)/5002  4 3
6(0)/5003  4 ?
-> access 6/5000
6(0)/5000  1
6(0)/5001  2
6(0)/5002  3
6(0)/5003  4 ?
-> lookat
Active process is -20.
-> lookat 5
-> lookat
Active process is 5.
-> access 4000/100
5:4000(0)/100  064000
5:4000(0)/101  177777 ?
->
```

A more efficient way of examining a number of words of memory is to use the dump command. It requires two arguments, a starting and an ending address. Both these arguments must be valid address-expressions. The dump format also depends on access_type.

```
-> access_type
Current access type is octal.
-> lookat
Active process is 5.
-> lookat 3
-> lookat
Active process is 3.
-> dump 6000/10 6000/15
3:6000(0)/10  000003 000300 000717 006362 000000 000000
->
```

2.8.4 Breakpoints

The most important function provided by the Ring Zero Debugger is the ability to set breakpoints. Breakpoints can be installed nearly anywhere in shared code. When a breakpoint is encountered by a process, control is passed to the debugger prior to the execution of the instruction at the specified location.

Ring Zero Debugger breakpoints provide the option of setting a breakpoint for either a particular process or any process which encounters it. They also provide the ability to have an associated count. This count indicates that the breakpoint will trap only when the breakpoint has been passed through a specified number of times. A complete description of breakpoints is given in section 3.2.

A simple example of installing a breakpoint is shown below. In the example below, two breakpoints are installed.

```
-> breakpoint prwf$$+1
-> breakpoint 6/44242
->
```

A very important point to note about breakpoints is that one must be sure that the given address is actually the beginning of a valid instruction. The Ring Zero Debugger has no way of knowing whether a specified location is code or data. In fact, it cannot even tell whether a location contains the first word of an instruction or the second word. If a breakpoint is put in the wrong place, the breakpoint will probably never invoke the debugger but rather cause the system to be corrupted. Thus one should never set a breakpoint at just any random address without knowing beforehand that the given location contains the start of a valid instrucion.

Once a breakpoint has been installed, one can issue a command to examine the state of the breakpoint. The command to examine a particular breakpoint is list. It takes a breakpoint-expression as an argument. The command to show the state of all breakpoints is listall. These commands are shown below. A complete description of the fields shown is given is section 3.2.

```
-> list pogtur+1
Type   Address       Procedure        Process  Count  Mnemonic
brkpt  6(0)/45205    PAGTUR + 1         Any        1    CRA
-> listall
Type   Address       Procedure        Process  Count  Mnemonic
brkpt  6(0)/45205    PAGTUR + 1         Any        1    CRA
brkpt  11(0)/31237   PRWF$$ + 1         Any        1    LDA
->
```

Once a breakpoint has been installed at a location, it remains there until it is explicitly removed. That is the function of the clear command. It takes a breakpoint-expression as an argument. The clearall command removes all breakpoints.

```
-> clearall
-> listall
No breakpoints are set.
-> breakpoint prwf$$+1
-> clear prwf$$+1
-> listall
No breakpoints are set.
->
```

2.9 Error Handling

Various types of errors can occur while one is using the Ring Zero Debugger. In the context of the debugger these are broken into 4 classes: user errors, warnings, system errors, and faults.

User Errors

User errors occur when a user has requested something of the debugger that it either does not understand or cannot do. The assumption is that in most cases, the user is responsible for correcting the situation. When such a condition occurs, the current command is always aborted and control returns to the debugger command level. Descriptions of all user error messages are given in appendix D. Some examples of these error messages are shown below.

```
-> foo

*** Debugger user error:
    Unknown command.
-> clear pagtur+2

*** Debugger user error:
    No breakpoint exists at specified address.
->
```

Warnings

Warnings occur when the debugger may have failed to perform some action and therefore its future operation may not always be correct. A user is in no way responsible for causing this condition. Presently there is only one such case in the debugger. Warnings only refer to any debugger operation still in progress. If the operation either terminates or is aborted, then the warning can be ignored.

System Errors

System errors occur when there is some internal inconsistency within the Ring Zero Debugger. The assumption here is that the condition has been caused either by an error in the debugger software or an error in the system hardware. If the problem does not seem as if it is related to the hardware, the problem should be reported as described in appendix F.4.

Faults

The Ring Zero Debugger has a separate fault handler for its own process. If a fault occurs while the debugger is running, the fault handler will print a message describing the fault, abort the current command, and return to command level.

Some faults, notably page and segment faults, can be expected to happen often. The reason for this is that the Ring Zero Debugger does not have the ability to reference non-resident memory. (For a further discussion of this point see section 5.2.) Any command which does so will cause the debugger to take a page fault, abort the current command, and print an error message. If one is determined to reference certain memory, the only recourse is to leave the debugger and attempt to have Primos bring it in. One way to do this is by touching it with VPSD. A simple example of a page fault is shown below.

```
-> access p$cidx
41(0)/137775  ARGT
41(0)/137776  LDA# 140316
41(0)/137777  STA# SB%+31
41(0)/140000
*** Fault while in debugger:
    Page fault (type 10) encountered at 55(0)/15233
    Attempt to reference 41(0)/140000
->
```

While most faults encountered by users are innocuous, a few may indicate serious problems. Page and segment faults usually only indicate non-resident or undefined memory. If this does not seem to be the case, or if any other type of fault occurs, the fault probably indicates a debugger software or system hardware error. In this case, the error should be treated like a system error and reported as described in appendix F.4.

3. Ring Zero Debugger Command Descriptions

The purpose of this chapter is to give full descriptions of all the commands in the Ring Zero Debugger. Each section in this chapter describes a number of commands that are related based on their functionality. The format of each section is to give descriptions of the commands followed by examples of all the commands in the current section. In specifying the command syntax, the common convention of surrounding optional arguments by [ ] is used.

3.1 Referencing Memory and Registers

This section will describe commands that can be used to read and write either resident memory or a process's registers. Memory can be printed in a variety of formats for any process. A command to search for patterns in memory is also discussed.

Referencing Memory - The Access Command

The generic format for the access command (abbreviated a) is:

```
Access <address-expression>
```

where

```
<address-expression> is described in section 2.4.1
```

The access command prints the contents of the active process's memory at the specified address and waits for keyboard input. The type used for printing memory is set by the access-type command. The input consists of an optional new value for the current location and a required access command terminator. Any new input must be in the current access-type. Valid terminators are: a carriage return, which causes the next memory location to be accessed, a '^', which causes the previous location to be accessed, and a '?' or a '/', which causes an exit from the access mode without changing the currently accessed location. This command will not wraparound when it hits a segment boundary. It will cross into the next segment.

Referencing Registers - The Access_register Command

The generic format for the access_register command (abbreviated areg) is:

```
Access_REGister <access-registers>
```

where

```
<access-registers> ::= A | B | L | E | X | Y | PB | SB | LB |
                       XB | DTAR0 | DTAR1 | DTAR2 | DTAR3 |
                       KEYS | MODALS | OWNER | FCODE |
                       FADDR | TIMER | FAR0 | FLR0 | FAR1 |
                       FLR1 | 0 | ... | 77
```

The access_register command prints the contents of the specified register for the active process and waits for keyboard input (in most cases). Register values are shown in octal. Allowable input is either a new value for the register, in octal, or just a carriage return. A carriage return leaves the value of the register unchanged. Certain registers cannot be changed (e.g. FCODE and PB). In these cases, the command will not wait for input. The numbers 0 to 77 octal refer to the registers in the system register sets. These registers are DMA channels and microcode scratch registers.

Changing the Access_type - The Access_type Command

The generic format for the access_type command (abbreviated atype) is:

```
Access_TYPE [<access-type>]
```

where

```
<access-type> ::= Ascii | Bit | Decimal | Hex | Octal |
                  Symbolic
```

The access_type command sets the type that is used in printing memory. This attribute is relevant for both the access and dump commands. The access_type attribute remains in effect until the next access_type command is issued. Thus, leaving and re-entering the debugger has no effect on the access-type. The symbolic type refers to V-mode assembly language. If no type is given, the current type is displayed. The access-type's can be abbreviated by the first letter of the type (e.g. ascii=a, etc.).

Displaying Memory - The Dump Command

The generic format for the dump command (abbreviated d) is:

```
Dump <address-expression> <address-expression>
```

where

```
<address-expression> is described in section 2.4.1
```

The dump command prints the region of memory from the first argument to the second argument. The memory examined will be that of the active process. When the end of a segment is reached, the next location referenced is the first word of the next segment in sequence. The type used to print memory is determined by the access_type command.

Changing the Active Process - The Lookat Command

The generic format for the lookat command is:

```
LOOKAT [<process-number>]
```

where

```
<process-number> ::= a decimal number
```

The lookat command makes the specified process be the active process (see 2.5). This command determines which process's private address space will be referenced by debugger commands. It will overwrite the existing active process value. If a process number is given, it must be either a valid interrupt process or a logged-in user process. The active process will not change unless another lookat command is issued or until the debugger is exited. If no argument is given, the currently active process is displayed.

Searching for Patterns in Memory - The Search Command

The generic format for the search command (abbreviated srch) is:

```
SeaRCH <address-expression> <address-expression>
       <search-pattern>
```

where

```
<address-expression> is defined in 2.4.1
<search-pattern> ::= 'string' | <octal-list> <search-mask>
<search-mask> ::= & <octal-list> | <empty>
```

The search command searchs the region of memory from the first argument to the second argument for a given sequence of words. The memory examined will be that of the active process. When the end of a segment is reached, the next location referenced is the first word of the next segment in sequence. The pattern to search for can be a string of up to 20 characters long or a sequence of up to eight 16 bit octal numbers. Single-quote marks can be put inside strings by using two single quote marks for each single quote mark desired. An optional mask of octal numbers can be specified for search patterns specified as octal numbers. The optional mask will be logically AND'ed with memory before the comparison. The optional search mask can be smaller (i.e. fewer words) than the search pattern. The memory region will be searched for all matches except for patterns which overlap a previously found pattern.

Examples

A number of the commands described in this section were shown in examples in the previous chapter. The access, access_type, dump, and lookat commands were shown in examples in section 2.8.3. An example of the access_register command is shown below. In this example, the L and XB registers were supposed to be the same but were not. Thus, the XB register was changed to be the same as the L.

```
-> access_register l
L (high order): 000004
L (low order): 100300
-> access_register xb
XB (high order): 000005 4
XB (low order): 177777 100300
-> access_register xb
XB (high order): 000004
XB (low order): 100300
->
```

The search command allows one to search for patterns as either ascii strings or as a series of octal numbers. In the example below, a particular string (part of an IOAS control string) is searched for. Finding this string could allow one to change it with the access command, should it be wrong.

```
-> search 55/0 55/177777 '%11:2zo%$'
55(0)/53466 %11:2zo%$
->
```

Another situation where the search command could be used is to determine if a particular pattern which has erroneously overwritten some code is in a certain region of memory. If the pattern 101 0 102 0 101 has overwritten some code and you suspect that the pattern may be in a buffer in either segment 27 or 30, you could determine this by issuing the following command:

```
-> search 27/0 30/177777 101 0 102 0 101
27/5326 000101 000000 000102 000000 000101
27/7234 000101 000000 000102 000000 000101
->
```

A final example of the search command may help clarify the way the search mask can be used. Consider a situation where one wants to know if any page in a given segment is "in-transition, going-out". This condition is indicated by having bit 1 of a page's page map be 0 and bits 3 and 5 be 1. If the page map entries for a segment are located between 600/1000 and 600/1077, the following command could be used to find the "in-transition, going-out" pages:

```
-> search 600/1000 600/1077 24000 & 124000
600/1002 24022
600/1054 27512
600/1067 26324
->
```

3.2 Breakpoints and Single Steps

This section will describe the breakpoint and step commands. It will also discuss related commands that can be used to display information about existing breakpoints and commands that can be used to remove existing breakpoints.

Installing Breakpoints - The Breakpoint Command

The generic format for the breakpoint command (abbreviated brk) is:

```
BReaKpoint <break-expression> [<proceed-count>]
```

where

```
<break-expression> is described in section 2.4.2
<proceed-count> ::= 1 to 32767
```

The breakpoint command can be used to set a breakpoint anywhere in shared code. Breakpoints work in both ring 0 and ring 3. There are a number of different ways to specify the location where the breakpoint is to be installed. These are discussed in section 2.4.2. If the optional process number is specified, only that process will stop when the breakpoint is hit. Other processes which encounter the breakpoint will continue transparently. If no process number is specified, any process will stop when the breakpoint is hit.

If the specified address is the name of a procedure, the breakpoint will be set on the first instruction of the procedure. If that instruction is an argt then the breakpoint must be set on the next instruction. Breakpoints are not allowed on argt instructions since they are not real, executable instructions. If an attempt is made to do so, the debugger will treat the attempt as a user error.

The optional proceed count can be used to allow processes to pass through the given breakpoint a specified number of times before stopping. This count is decremented each time the breakpoint is encountered. When the count reaches zero, the debugger command level is entered. If the proceed count is omitted or the count is decremented to zero, it is set to one.

While most commands abort when they have a need to reference non-resident memory, this is not true of the breakpoint command. The breakpoint command should never fail due to non-resident memory. This is because breakpoints are treated as pended operations. (For a description of pended operations see section 5.2.)

Removing a Breakpoint - The Clear Command

The generic format for the clear command (abbreviated clr) is:

```
CLeaR <break-expression>
```

where

```
<break-expression> is described in section 2.4.2
```

The clear command removes the specified breakpoint or single step from the specified location. If the specified location has a single step in progress as well as a previously installed breakpoint, only the breakpoint will be removed.

Removing All Breakpoints - The Clearall Command

The generic format for the clearall command (abbreviated clra) is:

```
CLeaRAll
```

The clearall command removes all breakpoints and single steps from the code. If one of the locations has a single step in progress as well as a previously-installed breakpoint, only the breakpoint will be removed.

Displaying a Breakpoint - The List Command

The generic format for the list command is:

```
LIST <break-expression>
```

where

```
<break-expression> is defined in section 2.4.2
```

The list command displays information about a particular breakpoint or single step. The fields displayed in the list command are described below.

- o Type. The type can be either a breakpoint or a single step.
- o Address. The location in memory where the breakpoint/single step is installed.
- o Procedure. The procedure name given is an educated guess as to which routine the breakpoint/single step is in. The guess comes from taking the address and consulting the Primos load maps. Usually it finds the correct routine, but this is not always the case (especially with assembly language routines).
- o Process. The process field specifies which process the breakpoint/single step is meant for.
- o Count. This field refers to the proceed count for breakpoints and the single step count for single steps.
- o Mnemonic. This field displays the assembly language mnemonic for the instruction at the specified location. If a breakpoint is "pended", this field will be empty (Refer to section 5.2 for a description of pended operations).

Displaying All Breakpoints - The Listall Command

The generic format for the listall command (abbreviated lsta) is:

```
LiSTAll
```

The listall command displays information about all breakpoints and single step's in the system.

The format is the same as for the list command.

Single Stepping a Process - The Step Command

The generic format for the step command (abbreviated s) is:

```
Step [<step-count>]
```

where

```
<step-count> ::= 1 to 32767
```

The step command causes the original process to single step as many instructions as are specified in the step count. If no step count is specified, it is assumed to be one. A single step means that the original process executes the next assembly language instruction in the breakpointed or suspended procedure and then returns control to the debugger. If any fault is taken during the execution of this instruction, any code needed to service the fault will be executed and the next instruction executed before control is passed to the debugger. Hence, many instructions may be executed in order to execute the one that is to be single stepped. It is also the case that any higher priority processes will execute before the single stepped process and may encounter breakpoints before the lower priority process has done the single step. A single step is only allowed if the Ring Zero Debugger was entered due to a breakpoint or previous single step. Another restriction on single stepping is that only one process in the system can be actively single stepping at a time.

Examples

In the following example, a breakpoint will be placed at the beginning of the routine clSget. This routine deals with parsing the command line. To cause the breakpoint to be hit, the Primos command "date" is entered. After being hit, the breakpoint is removed.

```
-> breakpoint cl$get+1
-> listall
Type   Address         Procedure       Process  Count  Mnemonic
brkpt  13(0)/45661     CL$GET + 1        Any        1    JMP
-> continue
Leaving the debugger.
date

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 13(3)/45661 (CL$GET + 1).
-> clearall
-> listall
No breakpoints are set.
->
```

Having gotten into the routine clSget, the next command shown is the step command. After a step is issued a few times, one may wonder what instructions are being executed. The access command is used to examine the instructions. With a little experience, one could tell that

these instructions are building an argument list in preparation for making a shortcall to another routine. A shortcall is accomplished with a jsxb instruction.

```
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 13(3)/45703 (CL$GET + 23).
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 13(3)/45705 (CL$GET + 25).
-> access cl$get+23
13(0)/45703  EAL% LB%+ 426
13(0)/45705  STL% SB%+ 65
13(0)/45707  EAL% SB%+ 0
13(0)/45711  STL% SB%+ 67
13(0)/45713  EAL% LB%+ 400
13(0)/45715  STL% SB%+ 71
13(0)/45717  EAL% SB%+ 65
13(0)/45721  STL% SB%+ 74
13(0)/45723  EAL% SB%+ 71
13(0)/45725  JSXB% LB%+ 374 ,*  ?
```

Having seen the code that is being executed, one can see the effect of using a step count with the step command. After single stepping through about 19 instructions, one can also see that the code was in the process of shortcalling the routine mkonuS to setup an onunit. Next a much larger step count is given. This count is enough to allow Primos to begin to print out the date. Finally, a continue command is issued letting Primos print out the rest of the date.

```
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 13(3)/45707 (CL$GET + 27).
-> step 3

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 13(3)/45715 (CL$GET + 35).
-> step 15

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 13(3)/46462 (MKONU$ + 24).
-> step 100
30 Aug 85 18
Debugger entered due to breakpoint/single step.
     Process 1 was executing at 6(0)/106055 (ACCOM$ + 1).
-> continue

Leaving the debugger.
:49:52 Friday
OK 18:49:55  0.115  0.000
```

3.3 Examining a Process's State

The commands in this section allow one to examine the state of a particular process. The term state refers mainly to the process's current register values and its current stack frame. Also included is the previous stack history and terminal buffers, if they exist. Some of the commands described in other sections also allow access to process state information, but the primary ones are here.

Displaying the Current Arguments - The Arguments Command

The generic format for the arguments command (abbreviated args) is:

```
ARGumentS
```

The arguments command prints the name of the current procedure for the active process, the number of arguments defined in the routine (from the ECB), and then prints the values of any arguments passed to this procedure. The address of each argument is printed followed by the first 2 words of the value of the argument in octal (2 words are printed regardless of the true length of the argument). The number of arguments shown reflect the number actually passed to the current procedure which may be less than the number defined.

Displaying a Process's PCB - The PCB Command

The generic format for the pcb command (abbreviated p) is:

```
Pcb [<process-number>]
```

where

```
<process-number> ::= a decimal number
```

The pcb command prints the contents of selected fields of a process's process control block followed by its concealed stack frames. Concealed stack frames are only shown if they are non-zero. Those that have not yet been built into fault frames are marked as "active". The process number must be a valid interrupt process or a configured user process. If no process number is specified, the active process is assumed.

Display Process Information - The Status Command

The generic format for the status command (abbreviated stat) is:

```
STATus [<status-options>]
```

where

```
<status-options> ::= <process-number> | ALL | USer |
                     INTerrupt
<process-number> ::= a decimal number
```

The status command displays the current status for either a specified process, all processes, user processes, or interrupt processes. If no process or class of processes is specified, the active process is assumed. The status of a process is represented by the current values of its registers. These registers are read from the machine register sets if the process owns one. Otherwise, the information comes from the pcb. If a class of processes is specified, only limited information will be printed out about a process, and logged-out processes will be ignored. If a single process is specified, all registers associated with the process will be displayed, even if the process is logged-out. A description of the information shown by the command is given in section 2.8.2.

Examining a Process's Stack - The Trace Command

The generic format for the trace command (abbreviated t) is:

```
Troce [<process-number> [<address-expression>]]
```

where

```
<process-number> ::= a decimal number
<address-expression> is described in section 2.4.1.
```

The trace command prints stack frames for the given process. Any process number specified must be a valid interrupt process or a logged-in user process. Trace allows one to interactively examine the previous state of any process by displaying detailed information from each individual frame on the stack. This information includes the process's base registers, the size of the stack, and the names of the calling and called procedures. Subcommands to the trace command allow one to move up or down the stack to any given frame and to print additional information about the stack frame. The following subcommands are defined for the trace subsystem:

Arguments	Prints the arguments at this level of the stack.
Current	Redisplays the current stack frame.
Father [<n>]	Makes the calling procedure's stack frame the current one. If n is specified, n frames of the stack are pushed before reaching the current one.
GOTO [<levels>]	Make <level> the current one. If omitted, <level> defaults to 1.
Quit	Leaves the trace command subsystem.
Son [<n>]	Makes the called procedure's stack frame the current one. If n is specified, n stack frames are popped before arriving at the current one.
STack [<start> [<end>]]	Dumps the current stack frame in octal. Start and end are relative locations within the current frame. If not specified, start begins at '10 and end defaults to the end of the stack frame.
User <process-number> [<address>]	

Starts tracing the specified process's stack.

If the process number is not specified in the trace command, the stack examined is that of the active process. If no address is given the stack trace begins with the current SB of the given process. Otherwise, the trace begins with the given address.

Displaying a Process's Terminal Buffers - The Ttybuf Command

The generic format for the ttybuf command (abbreviated tt) is:

```
TTybuf [<process-number>]
```

where

```
<process-number> ::= a decimal number
```

The ttybuf command prints the contents of the specified process's input and output terminal buffers. The output is printed as Ascii characters. Non-printing characters are shown as the character ".". The process number must be one of the configured user processes on this system. It should be noted that not all user processes have terminal buffers. For example, processes which are phantoms or slaves do not. If no process number is specified, the buffers printed are those of the active process. If process 1 is specified, the process 1 message buffer will be printed in addition to the regular input and output buffers.

Examples

The following example will illustrate the arguments command. In this example, a breakpoint has previously been placed in the locate routine. The Primos command "avail" is issued to make a process hit the breakpoint.

```
avail

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/14216 (LOCATE + 1).
-> arguments
Current routine: LOCATE

4 arguments at SB%+22:
# 1 at 11(0)/123007  : 000000.005415
# 2 at 6003(0)/1323  : 000000.000001
# 3 at 717(0)/5524   : 000000.000000
# 4 at 717(0)/5520   : 000000.000001
->
```

Next a pcb command is issued. In this case, as is often true, there isn't much that is interesting. One can see that most of the information is out of date by comparing the program counter (pb register) shown in this command's output to what was given before when the current breakpoint was hit. However, this command can occasionally be helpful in trying to determine recent process fault activity. As a matter of fact, the last fault on the

concealed stack in this example shows the illegal instruction fault taken for the previously hit breakpoint.

```
-> pcb

Process:   1
Level:  622                          Link: 000000
Wait list: 0(0)/544                  Abort flags: 0000000000000000
PB: 6(0)/42271                       SB: 6000(0)/1202
LB: 6(0)/42110                       XB: 6(0)/55534
L: 000000 000000                     E: 000000 000000
X: 000000                            Y: 000000
FAR0: 000000 000000                  FLR0: 000000 000000
FAR1: 000000 000000                  FLR1: 000000 000000
Interval timer: 177637 000000        Elapsed timer: 000001 143453
DTAR2: 140002 165101                 DTAR3: 176302 165064
Keys: 034201

Concealed stacks:
   PB              KEYS       FCODE(high)          FADDR
   11(0)/14216     014100     014217               11(0)/1703
   13(3)/55722     014000     000000               13(3)/60760
->
```

The next example shows the type of output one can get from the status command with the user option. In this example, one will note that two n1locks are owned by process 1 which is the process that has just hit a breakpoint in the locate routine. This is logical given the role that locate plays in the file system.

```
-> status user
Process 1  SYSTEM              *** Owns register set 0 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/14216 (LOCATE + 1)
   LB:    11(0)/16070 (LOCATE (et al))
   Locks owned: FSLOK  UFDLOK

Process 2  (Login name is not resident)
   Level: Priority 1 user
   Type:  Normal terminal user
   State: Waiting at 6(0)/13352 (ASRSEM + 2)
   PB:    6(0)/34235 (WAITA + 74)
   LB:    6(0)/55772 (C1IN$)

Process 29  (Login name is not resident)
   Level: Network process
   Type:  Network process
   State: Waiting at 12(0)/25302 (PNTSEM)
   PB:    6(0)/34454 (WAIT + 4)
   LB:    6(0)/34156 (SETSWI (et al))

->
```

The next example shows how a particular process number can be given. When a specific process number is given, all the process's registers are displayed. When a class of processes is requested (e.g. user), only a few registers are shown.

```
-> status -7
Process -7  DK1PCB
   Level: Disk/Ringnet process
   State: Waiting at 4(0)/534 (DSKSEM)
   PB:    6(0)/37537 (DMA_ERR + 224)
   LB:    6(0)/103110 (Unknown)
   SB:    4(0)/164070       XB:     0(0)/1200
   L:     000000 000000     E:      000000 000000
   X:     000000            Y:      000000
   FAR0:  000000 000000     FLR0:   000000 000000
   FAR1:  000000 000000     FLR1:   000000 000000
   Keys:  034001

->
```

Having hit a breakpoint in locate, the trace command can be used to determine the sequence of routines that were called to get to locate. The trace subcommand "arguments" can be used to examine the arguments passed to locate. The trace subcommand "father" can be used to sequence back down the process's stack.

```
-> trace

Level 1:  LOCATE
Root: 6003  SB: 6003/1354  Size: 152 words  Type: 000000 (PCL)
Keys: 014000
Call at 11(0)/122570 (READ_ENT (et al) + 1540)
SB: 6003(0)/1150 LB: 11(0)/123306

(trace)> arguments
Current routine: LOCATE

4 arguments at SB%+22:
# 1 at 11(0)/123007  : 000000.005415
# 2 at 6003(0)/1323  : 000000.000001
# 3 at 717(0)/5524   : 000000.000000
# 4 at 717(0)/5520   : 000000.000001

(trace)> father

Level 2:  READ_ENT
Root: 6003  SB: 6003/1150  Size: 132 words  Type: 000000 (PCL)
Keys: 014000
Call at 11(0)/117342 (FNDENT (et al) + 52)
SB: 6003(0)/430 LB: 11(0)/120054

(trace)> father

Level 3:  FNDENT
Root: 6003  SB: 6003/430  Size: 336 words  Type: 000000 (PCL)
Keys: 014000
Call at 11(0)/136463 (AT$ANY + 235)
SB: 6003(0)/164 LB: 11(0)/137322

(trace)> father

Level 4:  AT$ANY
Root: 6003  SB: 6003/164  Size: 164 words  Type: 000000 (PCL)
Keys: 014100
Call at 13(3)/51726 (SRSFX$ + 1306)
SB: 6002(3)/3224 LB: 13(0)/52720

(trace)> quit
->
```

The final example shows the output from the ttybuf command. In this particular example the active process is 1. This means that in addition to the normal input and output buffers, the user 1 message buffer is shown as well.

```
-> ttybuf

User 1 message buffer (600 bytes long) at 7(0)/17224:
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................

Input buffer (400 bytes long) for user 1 at 7(0)/7142:
    ..........................................................
    ..........................................................
    ..........................................................
    ..............................SE -091385 -1903
    CO -CONTINUE
    date

Output buffer (600 bytes long) for user 1 at 7(0)/0:
    t_log -net -off
OK, RDY -LONG
OK 00:02:51  49.154  100.700
/* Enter time and type CO -CONTINUE.
OK 00:02:51  0.084  0.000
CO -PAUSE
OK 00:02:51  0.060  0.033
OK 19:03:04  0.303  0.690
OK 19:03:04  0.048  0.000
MAX ALL
OK 19:03:04  0.103  0.000
COMO -NTTY
OK 19:05:08  0.060  0.060
CO -END
OK 19:05:08  0.054  0.000
13 Sep 85 19:10:52 Friday
OK 19:10:55  0.187  0.166

->
```

3.4 Examining the State of the System

This section will describe commands that give information about the state of the system in general rather than a specific process. These commands allow one to examine the Primos n1locks, the system ready list, and certain system registers. The status command, described in the previous section, is another way to look at the state of the system as a whole.

Displaying the N1locks - The Print_locks Command

The generic format for the print_locks command (abbreviated plocks) is:

```
Print_LOCKS
```

This command displays information about the Primos N1locks. Any processes waiting for a particular lock will be shown. The locks are printed in priority order.

Displaying the Ready List - The Ready_list Command

The generic format for the ready_list command (abbreviated rdylst) is:

```
ReaDY_LiST
```

The ready_list command prints a diagram showing the system ready list. The process for the Ring Zero Debugger is not shown.

Display Certain System Registers - The System_registers Command

The generic format for the system_registers command (abbreviated sysreg) is:

```
SYStem_REGisters
```

The system_registers command prints the pswpb and pswkeys registers as well as the DMA channels. The DMA channels are only shown if the registers are non-zero. The word count of the DMA register is right shifted by 4 to right-justify the field.

Examples

The output from the print_locks command is shown below. (Not all the locks are shown in this example).

```
-> print_locks

FSLOK: Locked for reading by 1 user(s).
No reader(s) waiting
No writer(s) waiting

UFDLOK: Locked for reading by 1 user(s).
No reader(s) waiting
No writer(s) waiting

BLKLOK: Unlocked.
No reader(s) waiting
No writer(s) waiting

MOVLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

SEGLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

PAGLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

->
```

Another example of the status command with the user option is given below. From this output, one can tell which processes own the locks described as locked by the print_locks

command.

```
-> status user

Process 1  SYSTEM                    ••• Owns register set 0 •••
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/14216 (LOCATE + 1)
   LB:    11(0)/16070 (LOCATE (et al))
   Locks owned: FSLOK  UFDLOK

Process 2  (Login name is not resident)
   Level: Priority 1 user
   Type:  Normal terminal user
   State: Waiting at 6(0)/13352 (ASRSEM + 2)
   PB:    6(0)/34235 (WAITA + 74)
   LB:    6(0)/55772 (C1IN$)

Process 29  (Login name is not resident)
   Level: Network process
   Type:  Network process
   State: Waiting at 12(0)/25302 (PNTSEM)
   PB:    6(0)/34454 (WAIT + 4)
   LB:    6(0)/34156 (SETSWI (et al))

->
```

Output from the ready_list command is shown next. In this example, the first process on the ready list is the clock process, followed by process 1, and then the two backstop processes. This particular example came from the situation where process 1 had just hit a breakpoint. In this case, one might expect process 1 to be at the head of the ready list and in many cases it probably would. Apparently here, however, the phantom interrupt code for the clock must have run while the breakpoint was being serviced but before the debugger process had run and inhibited interrupts. Thus the phantom interrupt code notified the clock process, putting it on the ready list. The backstop processes should always be on the ready list and it is common to also see the clock process there.

```
-> ready_list
START -> CLKPCB
           |
           v
         USR001
           |
           v
         BK1PCB -> BK2PCB
->
```

Finally, the system_registers command shows the value of two registers from the system scratch registers and then what it supposes are the active dma channels. The pswpb and pswkeys registers show the values of the pb and keys of the currently executing process when the last machine interrupt was taken. Unfortunately, this is the only information available about the state of the machine at that point. Therefore, it is usually not very

helpful but can be a valuable clue in some situations.

```
-> system_registers
PSWPB: 55(0)/615    PSWKEYS: 014002 100037

DMA channel     I/O address     Word Count
      0         0(0)/174000       000000
      2         0(0)/176000       000000
      6         0(0)/1000         000000
     14         0(0)/151005       007405
     16         0(0)/144265       000000
     20         0(0)/171000       000000
     22         0(0)/175000       000000
     24         0(0)/100255       000300
     26         0(0)/72001        000200
     30         0(0)/10003        002200
     32         0(0)/41272        002200
     34         0(0)/201          002027
     36         0(0)/4003         003500
->
```

3.5 Retrieving Symbolic Information

This section will describe commands which can be used to retrieve symbolic information from the Primos load maps.

Retrieving Symbols - The Lookup_Address Command

The format of the lookup_address command (abbreviated la) is:

```
Lookup_Address <address_expression> [<symbol_type>]
```

where

```
<address_expression> is described in section 2.4.1
<symbol_type> ::= ANY | Common | Other | ECB | PB | LB | LBN
```

Given an address and an optional symbol_type, the Lookup_Address command searches the Primos load maps for the name of a symbol. Primos load maps are created by the SEG program with three types of symbols: routines, common areas and all other symbols. Commons and others have only one address associated with them. Routines, on the other hand, can have several addresses associated with them: ECB address and initial values of PB and LB. If the Lookup_Address command cannot find an exact match, it returns the name of the specified type of the symbol which is closest to the specified address (but less than or equal to the specified address). The only exception is the LBN option which returns the names of all the routines which have the same LB as the specified address. If no symbol_type is specified than ANY symbol_type is assumed. Symbol_type ANY causes the command to search the PRIMOS load maps with all possible options and return the name of the symbol of any type (routines, commons, others) closest to the specified address.

Retrieving symbols address - The Lookup_Symbol command

The format of the lookup_symbol command (abbreviated ls) is:

```
Lookup_Symbol <symbol>
```

where

```
<symbol> ::= an object name from the Primos load maps.
```

Given a name of the symbol from Primos load maps the Lookup_Symbol command returns information for a specified symbol, based on the type of the symbol. The returned information for the commons and the others is the address of the specified symbol. In the case of the routines the returned information is the address of the ECB for the specified routine and the initial values of PB and LB.

Examples

To find the name of the routine whose address of the ECB (or PB or LB) is the closest to the specified address:

```
-> lookup_address 11/57700 ecb
   ROUTINE:   SRCH$$ + 10 from ECB
-> lookup_address 11/62452 pb
   ROUTINE:   GPATH$ + 20 from PB
->
```

To find the name of the common block whose address is the closest to the specified address:

```
-> lookup_address 12/1640 common
   COMMON:   CONTYP + 1
->
```

To find the name of the other symbol whose address is the closest to the specified address:

```
-> lookup_address 12/4067 other
   OTHER:   SLCMCH + 0
->
```

To find the names of all routines which have the same LB as the specified address:

```
-> lookup_address 12/4020 lbn
   ROUTINE:  SLCINI
   ROUTINE:  SLCBND
   ROUTINE:  SLCDS
   ROUTINE:  SLCOT$
   ROUTINE:  SLCOTP
   ROUTINE:  SLIOC
   ROUTINE:  SLCCLK
   ROUTINE:  SLXDLC
   ROUTINE:  SLCRST
->
```

To find the name of the symbol of any type closest to the specified address:

```
-> lookup_address 15/2271
   ROUTINE:   NLOGIN + 2 from PB
->
```

To find the information about given symbol:

```
-> Lookup_Symbol PRWF$$
   ECB of routine:  11/35543
   PB of routine:   11/33046
   lb of routine:   11/35136
-> Lookup_Symbol PAGCOM
   Address of common:  14/614
-> Lookup_Symbol TIMERS
   Address of other:  6/5032
->
```

3.6 Program Variables

It is extremely useful to be able to reference variables in a program by name. DBG provides this ability with the ":" and let commands. With some extra effort on the part of the user, this ability can also be available in the Ring Zero Debugger. The extra effort comes in the form of having to manually define each variable that is to be used. Once the variable has been defined, it can be referenced symbolically as is done with DBG. This section describes the commands that can be used to define and then reference program variables.

Defining Program Variables - The Define_variable command

The format of the define_variable command (abbreviated defvar) is:

```
DEFine_VARiable [<procedure_name>\] <variable>
                <address_expression> [<variable_type>
                <variable_length>]
```

where

```
<procedure-name> ::= procedure from the Primos load map
<variable> ::= name with PL1 identifier syntax
<address_expression> is described in section 2.4.1
<variable-type> ::= Ascii | Bit | char_Vary | Decimal |
                    Octal | Pointer
<variable-length> ::= a decimal number
```

Before any program variable can be referenced, it must be defined with the define_variable command. This command enters a new variable name with the given address and attributes into an internal debugger table. If the address is specified in terms of a base register, the evaluation of the specified address does not take place until the program variable is actually referenced using the ":" or let commands.

The Ring Zero Debugger contains only one internal table for program variables. Therefore,

there is no notion of scope. However, the debugger does try to protect users against accidental modification of memory when a program variable is assigned a value. This could happen if a program variable was assigned a value while the active process is not executing in the program that the variable is defined for. The means for preventing this from happening is to allow a procedure name to be associated with the variable during definition. Then when the variable is referenced, the debugger compares the program variable's procedure with the current procedure of the active process. If they are different, the reference is not allowed.

The information needed for the define_variable command comes from a listing of the program which defines the variable. (See appendix A for details on getting this information from a listing). The main piece of information concerns the address-expression which defines where the variable is located. If the address is given as an offset from one of the base registers then the procedure-name must be specified. Otherwise it is optional. The variable type and length can also be determined from the listing. If both the type and length are omitted, the type is assumed to be octal and the length is assumed to be 1. The types known to the debugger are listed below.

Ascii	A nonvarying character string of length-specified bytes. The length argument is required. The maximum length is 256 characters.
Bit	A bit string. Length is assumed to be 1 word. Any length argument is ignored.
char_Vary	A varying character string. The length in bytes is a required argument. The maximum length is 256 characters.
Decimal	A signed decimal number of length-specified words. If length is omitted, it is assumed to be one.
Octal	An octal number of length-specified words. If length is omitted, it is assumed to be one.
Pointer	A memory address. The length is assumed to be 2 words. Any length argument is ignored.

Defined variables are always active until deleted. If a user defines a new variable which already exists then the Debugger will replace newly defined variable with the old one. The current maximum on the number of defined variables in the debugger is 25.

Examining Program Variables - The ':' Command

The format of the ':' command is:

```
: <variable> [<variable_type>]
```

where

```
<variable> is described in define_variable command
<variable-type> is described in define_variable command
```

This command prints the contents of the specified variable in the format specified by the variable-type option. If no type is specified then the variable is printed using the type specified when the variable was defined. However, there are certain restrictions on the use of this command. First of all, if the variable was defined for a particular routine (a routine name was specified in a define_variable command) then the active process must be executing in this routine. Secondly, it is not possible to examine a variable using ascii or char_vary type if the variable was defined as bit, octal or decimal type with a length of 1 word.

Displaying Program Variables - The Display_variable Command

The format of the display_variable command (abbreviated disvar) is:

```
DISplay_VARiable [<variable>]
```

where

```
<variable> is described in define_variable command
```

Display_Variable command displays the current definition of the specified variable. The definition includes variable name, variable type, variable length, variable address and any optionally-specified procedure name. If no variable is specified then all defined variables with their attributes are displayed.

Deleting Program Variables - The Delete_variable Command

The format of the delete_variable command (abbreviated delvar) is:

```
DELete_VARiable [<variable>]
```

where

```
<variable> is described in Define_Variable command
```

The delete_variable command deletes a specified variable from the list of defined variables. If no variable is specified, all defined variables will be deleted. However, the debugger will query the user before deleting all defined variables.

Changing Program Variables - The Let Command

The format of the Let command is:

```
LET <variable> = <new_value>
```

where

```
<variable> is described in define_variable command
<new_value> ::= a value expressed in the type of its
                target
```

The let command assigns a new value to a specified variable. The specified value must be of the same type as a defined variable. Also if the variable was defined for a particular routine (a routine name was specified in a define_variable command) then an active process must be executing in this routine. Character strings must be surrounded by single quotes. Single quotes inside character strings require two single quotes for every single quote desired. If the length of a new value of a character string exceeds the length of the defined variable then the new character string will be truncated to the length of the defined variable. If the length of a new character string is less than a length of a defined variable then the new character string will be padded with blanks - left justified (for Ascii type only). If a variable is a bit string then only up to 16 bits may be specified (just 1 word). However, if less than 16 bits is specified then the word will be left justified - the rest of the bit string will be padded with '0'b. If a variable is a pointer type then the user has an option of supplying a ring number. The valid ring numbers are 0 and 3 only.

Examples

To define a new program variable use the define_variable command:

```
-> define_variable prwf$$\code SB%+105 decimal 1
-> define_variable status 10/11776
->
```

To examine a program variable use ':' command:

```
-> : code
  12
-> : code o
  00014
-> : status
  00000
->
```

To display one or all program variables along with their attributes use the display_variable command:

```
-> display_variable code

Procedure      Variable     Address      Type        Length
PRWF$$         CODE         SB%+105      DECIMAL       1
-> disvar

Procedure      Variable     Address      Type        Length
_PRWF$$        CODE         SB%+105      DECIMAL       1
               STATUS       10/11776     OCTAL         1
->
```

To delete one or all defined variables use the delete_variable command:

```
-> delvar code
-> disvar

Procedure     Variable     Address     Type        Length
              STATUS       10/11776    OCTAL         1
-> delvar

OK to delete all defined variables?  yes
-> disvar

No variables are defined.
->
```

To assign a new value to a defined variable use LET command:

```
-> disvar

Procedure     Variable     Address     Type        Length
DISKIO        CNT          SB%+44      DECIMAL       1
              STRING       4000/10000  ASCII        20
              BIT          4001/105    BIT           1
              PTR          4002/1000   POINTER       2
-> let cnt = 100
-> : cnt
  100
-> let string = 'It''s only a test'
-> : string
It's only a test
-> let bit = 11101
-> : bit
  1110100000000000
-> let ptr = 55(0)/27770
-> : ptr
  55(0)/27770
->
```

3.7 User-defined Commands

This section describes the commands that allow users to manipulate what are known as user-defined commands. These commands allow one to define a new command which can then be used to invoke a series of previously defined commands. User-defined commands are similar to the abbreviation facility in Primos except for two points. Abbreviations in Primos accept arguments while user-defined commands do not. The other point is that user-defined commands can be recursive. This is not true of abbreviations. The commands covered in this section allow users to define user-defined commands, to display current definitions of user-defined commands, and to delete specific user-defined commands.

Defining New Commands - The Define_command Command

The format of the define_command command (abbreviated defcom) is:

```
DEFine_COMmand <user-command-name> <command-list>
```

where

```
<user-command-name> ::= a string
<command-list> ::= <command> ; <command-list> | <command>
```

Define_command provides the ability to execute a sequence of commands by entering a single user-defined command. It allows a user of the Ring Zero Debugger to define new commands which are composed of a series of already defined commands and their arguments. Entering the specified command will cause each of the commands specified in the list to be executed.

Some commands can cause one to leave the debugger command level. In these cases, any remaining commands will be executed when the debugger is reentered. The command-list can contain other user-defined commands. User-defined commands can also be recursive, however this must be done carefully. No syntax or semantic checking is done on the command-list at definition time.

One situation to avoid is defining a command with the same name as an existing debugger command. In this case the user-defined command will be effectively ignored since the debugger command line handler looks through the predefined debugger commands before it searches the user-defined command table.

Deleting User-defined Commands - The Delete_command Command

The format of the delete_command command (abbreviated delcom) is:

```
DELete_COMmand [<user-command-name>]
```

where

```
<user-command-name> ::= a string
```

Delete_command deletes the specified command from the list of defined commands. If no user-command-name is given, all defined commands will be deleted. Before deleting all user-defined commands, the user will be queried about the action.

Displaying User-defined Commands - The Display_command Command

The format of the display_command command (abbreviated discom) is:

```
DISplay_COMmand [<user-command-name>]
```

where

```
<user-command-name> ::= a string
```

Display_command prints the current definition of the specified user-defined command. If no user-command-name is given, all user-defined commands will be displayed.

Examples

The following examples will show different ways that user-defined commands can be used. In the first example, a user-defined command is made called "next" which simply steps through code and displays the assembly language for the instruction about to be executed. One point to note is that when a user-defined command is expanded, the expansion is echoed on the command line. In this example, the Primos command avail is entered to cause the breakpoint at prwf$$+1 to be hit. Then the user-defined command "next" is used.

```
-> access_type symbolic
-> define_command next step; dump * *
-> breakpoint prwf$$+1
-> continue
Leaving the debugger.
avail

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/33047 (PRWF$$ + 1).
-> next
-> STEP; DUMP * *

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/33051 (PRWF$$ + 3).
->  DUMP * *

11(0)/33051  STA# SB%+ 34
-> next
-> STEP; DUMP * *

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/33052 (PRWF$$ + 4).
->  DUMP * *

11(0)/33052  ANA#  33303
->
```

The next example shows another way user-defined commands can be used. In this example, a new command is defined which is just shorthand for an existing debugger command which may be somewhat tedious to type. (The debugger command "structure" is discussed. later in this section.) The purpose of the new command is to display the ecb of a routine symbolically. First, the address of the ecb must be determined and then the new command defined. What can't be seen here is worth noting. In defining the command "ecb", a space is typed at the end of the line. This space is needed because the debugger very literally expands what is typed as the definition of define_command and concatenates it with the next token in the command line.

```
-> lookup_symbol srch$$
 ECB of routine: 11(0)/57670
 PB of routine: 11(0)/56630
 LB of routine: 11(0)/57236
-> define_command ecb structure ecb
-> display_command

  List of defined commands:
  NEXT          :   STEP; DUMP * *
  ECB           :   STRUCTURE ECB
->
```

Now that the command has been defined it can be used to examine the ecb of the routine srchSS. The delete_command command is also shown.

```
-> ecb 11/57670
-> STRUCTURE ECB 11/57670
Structure ECB at 11(0)/57670.

   Offset | Field name        | Value
   -------|-------------------|-----------------
   057670 | pb                |  11(0)/56630
   057672 | frame size        |  000160
   057673 | stack root        |  000000
   057674 | args displac      |  000036
   057675 | num of args       |      6
   057676 | lb                |  11(0)/57236
   057700 | keys              |  014000
-> delete_command

OK to delete all defined commands? yes
->
```

The final example of a user-defined command will show how new commands can be defined recursively. As might be imagined, this must be done very carefully. In this example, a new command called "loop" is defined which will set breakpoints on consecutive locations as a process steps through code. After the command has been defined, a breakpoint is set in the routine p$cidx.

```
-> define_command loop breakpoint *; step; loop
-> display_command loop

  Defined command:
  LOOP          : BREAKPOINT *; STEP; LOOP
-> breakpoint p$cidx+1
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(0)/133503 (P$CIDX + 1).
->
```

Now that we are in the routine p$cidx, the loop command is issued. An important point to note here is that recursive commands will always cause an infinite loop. While this is not often desireable, it may be in certain cases. One case is where one may want to set up a test which executes forever. (Much of the debugger was tested this way.) The example below

shows how to set many breakpoints on a certain code path quickly. However, to stop the infinite loop, one has to quit out of the operation as shown below.

```
-> clear p$cidx+1
-> loop
-> BREAKPOINT *; STEP; LOOP
->  STEP; LOOP

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(0)/133504 (P$CIDX + 2).
->  LOOP
-> BREAKPOINT *; STEP; LOOP
->  STEP; LOOP

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(0)/133505 (P$CIDX + 3).
->  LOOP
-> BREAKPOINT *; STEP; LOOP
->  STEP; LOOP

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(0)/133510 (P$CIDX + 6).
->  LOOP
-> BREAKPOINT *; STEP; LOOP
->  STEP; LOOP

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(0)/133511 (P$CIDX + 7).

quit.
->
```

To show that these breakpoints are set, a listall command is issued.

```
-> listall
Type   Address         Procedure       Process  Count  Mnemonic
brkpt  41(0)/133503   P$CIDX + 1        Any        1    LDA
brkpt  41(0)/133504   P$CIDX + 2        Any        1    STA
brkpt  41(0)/133505   P$CIDX + 3        Any        1    EAFA 0
brkpt  41(0)/133510   P$CIDX + 6        Any        1    LDA
->
```

3.8 Miscellaneous Commands

This last section of the chapter describes commands which do not neatly fall into a specific category. Included are such useful functions as the ability to translate virtual addresses to physical addresses (and vice versa) and the ability to examine certain system data bases, field by field.

Resuming Primos - The Continue Command

The format of the continue command (abbreviated c) is:

```
Continue
```

The continue command causes execution of Primos to resume. Other than the step command, this is the only way to exit the Ring Zero Debugger.

A Help Facility - The Help Command

The format of the help command (abbreviated h) is:

```
Help [<commond>]
```

where

```
<commond> ::= any Ring Zero Debugger command
```

The help command displays information about the specified command. If no command is specified, the names of all commands are listed.

Examine the Fields of a Structure - The Structure Command

The format of the structure command (abbreviated struc) is:

```
STRUCture [<definition> [<address-expression>]]
```

where

```
<definition> ::= CLDATA | DISK_QUEUE_BLOCK | ECB |
                 FIGCOM | PUDCOM | PUSTAK |
                 SUPCOM | UPCOM
<address_expression> is described in section 2.4.1
```

The structure command prints the contents of memory starting at the given address and using the predefined definition as a template. Simple definitions of many Primos databases are defined in tables as part of the Ring Zero Debugger. These definitions contain text and data type for each field in a database. Thus, given an address, one can print a defined structure as a list of fields and their values (e.g. PUDCOM or the disk queue blocks). If the structure command is given without arguments, it will list the currently defined databases. If the name of the defined structure can be found in the Primos load maps, the address-expression need not be specified.

The intent of the structure command is that it be constantly expanded and updated. New definitions can be made by adding new entries to tables in the Ring Zero Debugger. If the data base that you define is just for a special situation then you might only modify your copy of the debugger. The more likely situation is that the data base you define is one that others will be interested in and thus the change should be made to actual copy of the debugger in Primos. If an engineer makes a change to an existing data base already known by the debugger, that engineer must be the one to update the definition in the debugger.

Translating Virtual to Physical - The Translate_to_physical Command

The format of the translate_to_physical command (abbreviated tphys) is:

```
Translate_to_PHYSical <address-expression>
```

where

```
<address_expression> is described in section 2.4.1
```

Translate_to_physical translates the given virtual address of the active process into a physical memory address. This command will only work for virtual addresses which translate into resident memory. The physical memory address will be returned as both a 16 bit physical page number and a 32 bit physical address.

Translating Physical to Virtual - The Translate_to_virtual Command

The format of the translate_to_virtual command (abbreviated tvir) is:

```
Translate_to_VIRtual <physical-page-number>
```

where

```
<physical-page-number> ::= a 16 bit octal number
```

The translate_to_virtual command returns the process number and virtual address that translate to the specified physical page number. Any physical page number specified must be configured on the existing system and "owned" by some processes' virtual address space in order for the command to return valid output. This command will not detect the fact that multiple virtual addresses may map into the same physical address (e.g. windowing through segment 0 for i/o). Only one virtual address will be returned in all cases.

Display Current State - The Where Command

The format of the where command (abbreviated wh) is:

```
WHere
```

The where command prints the process number of the original process, the current value of its program counter, and the reason for having entered the Ring Zero Debugger. The output from this command is identical to the banner printed when the debugger is entered.

Examples

The first example will show the results of issuing a help command with no arguments.

```
-> help
:                       Access              Access_REGister
Access_TYPE             ARGumentS           BReaKpoint
CLeaR                   CLeaRAll            Continue
DEFine_COMmand          DEFine_VARiable     DELete_COMmand
DELete_VARiable         DISplay_COMmand     DISplay_VARiable
Dump                    Help                LET
LIST                    LiSTAll             LOOKAT
Lookup_Address          Lookup_Symbol       Pcb
Print_LOCKS             ReaDY_LiST          SeaRCH
STATus                  Step                STRUCture
SYStem_REGisters        Trace               Translate_to_PHYSical
Translate_to_VIRtual    TTybuf              WHere

->
```

Next, the output from specifying a command to help is shown.

```
-> help access_type
Command name:   Access_TYPE  (ATYPE)

   Command description:
   Set the type used by the access and dump commands for
   printing memory.

   Command line arguments:
   [Ascii | Bit | Decimal | Hex | Octal | Symbolic]
->
```

Giving the structure command without arguments, will cause a list of currently defined data bases to be displayed. The assumption is that this list will grow over time. The original list of defined structures was just meant to show the usefulness of the command.

```
-> structure

PRIMOS data bases known to the Debugger:
CLDATA                  DISK_QUEUE_BLOCK        ECB
FIGCOM                  PUDCOM                  PUSTAK
SUPCOM                  UPCOM
->
```

In the following example, the structure command is used to look at the ecb for the prwfSS routine.

```
-> lookup_symbol prwf$$
 ECB of routine: 11(0)/35625
 PB of routine: 11(0)/33046
 LB of routine: 11(0)/35220
-> structure ecb 11/35625

Structure ECB at 11(0)/35625.

   Offset | Field name        | Value
   -------|-------------------|-----------------
   035625 | pb                |  11(0)/33046
   035627 | frame size        |  000202
   035630 | stack root        |  000000
   035631 | args displac      |  000070
   035632 | num of args       |      7
   035633 | lb                |  11(0)/35220
   035635 | keys              |  014000
->
```

The following examples show the output from the translate_to_physical and translate_to_virtual commands.

```
-> translate_to_physical 11/50500
Virtual address 11(0)/50500 translates to physical
address 1710500.
This address is on physical page 744.
-> lookat 1
-> translate_to_physical 6000/200
Virtual address 6000(0)/200 translates to physical
address 552200.
This address is on physical page 265.
-> translate_to_virtual 744
Corresponding virtual address is 11(0)/50000 for process 1.
-> translate_to_virtual 265
Corresponding virtual address is 6000(0)/0 for process 1.
->
```

The last example shows the output from the where command. It should be easily recognizable as the banner that is printed out when the debugger is entered.

```
-> where

Debugger entered due to console interrupt.
  Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4)
->
```

4. Uses of the Ring Zero Debugger

The previous chapters described the various commands in the Ring Zero Debugger and how they could each be used to obtain specific information. The focus of this chapter is on the way that these commands can be used to solve specific sorts of problems.

4.1 Adding New Code

The most common use of the Ring Zero Debugger will be for testing new code that is added to Primos. This section will describe some of the steps that one would typically go through in attempting to debug newly added code.

The first step in attempting to debug new code is to get a listing of the affected module or modules. If the language is Fortran, an expanded listing of the PMA is necessary. If the language is PLP, the listing should include statement offsets. In all cases the listing should have a cross-reference listing at the end. In addition to the typical reasons for wanting a listing of the module to be debugged, a listing is needed for defining program variables and determining where to place breakpoints.

The next step is to enter the Ring Zero Debugger before any process has passed through the newly added or modified module. If the module is not needed during coldstart code, then the system can be coldstarted and the debugger can be entered later on with a console interrupt. On the other hand, if the module is used during coldstart or you're not sure if it is or not, then the debugger should be entered during coldstart.

Once in the Ring Zero Debugger the module's program variables should be defined. Program variables are described in section 3.6. The module listing should be consulted to determine which program variables one might be interested in and what their actual definitions are.

The next step is to set breakpoints in the routine to be debugged and let a process encounter them. The name of the routine should be in the Primos load map as a global symbol so that the breakpoint command can be issued using the routine name directly. Once in the routine, the debugger can be used to single step through the new code. If the module is in PMA then the single step command can be used directly. If the module is in a high level language, then the breakpoint command is needed to single-step through the high-level-language statements. The statement offsets can be read from the listing to determine where the beginning of each statement is.

After each statement, one can use the ":" command to examine the previously defined program variables. If any of the variables are wrong, they can be corrected with the let command. If the module to be debugged also references system databases, these can be examined with commands such as print_locks and structure. A common one to examine might

be pudcom.

When adding new code to Primos, all paths through the new code should be tested. This means setting a lot of breakpoints and frequently examining variables. This was previously impractical to do but is relatively easy to do now. Doing this level of testing is somewhat tedious but is extremely important in order to produce high quality software. It is far easier for an engineer to find a problem at this point than for someone later on to try to determine which of the 1000 Primos modules is responsible for the erroneous behavior.

Another important use of the Ring Zero Debugger is to fully test the error or exception paths through new code. Doing this in the past was very difficult or impossible in most cases. With the debugger, there is no reason for not doing this testing. All error paths should be tested by simulating the error with the debugger. This can easily be done in most cases by setting error codes to different values with the let command.

One caveat about using the Ring Zero Debugger when modifying code is that it is not a substitute for functionally testing code. The debugger should be used first to verify code paths but test plans should be developed that use other means of testing the code, unless no other ways are possible (such as with testing certain error paths). The environment with the debugger configured is somewhat different from the actual environment that the system will run in. Thus, the final testing should use other means to verify correct program behavior.

The steps to follow in adding new code are summarized below:

1. make a listing of the affected module(s)
2. enter the debugger
3. define program variables (define_variable)
4. test all code paths by single stepping through new code and examining local and global variables (breakpoint, step, ;, structure)
5. modify variables to correct errors or to simulate error conditions (let)

4.2 Tracking Down System Failures

There are many different circumstances where the system or a particular process within the system fail. Sometimes this is related to code that has just been modified, but the connection between the failure and the change is not clear. Other times the system failure may occur on released software and there is no clue as to the problem. The Ring Zero Debugger has great potential for helping in many, but not all, of these situations. This section suggests some ways that one might use the debugger to examine certain system problems.

4.2.1 Fatal Process Errors

Fatal process errors can be mildly frustrating to examine without the debugger. There are many reasons for getting these errors. A common one is that the user's ring 3 stack has been corrupted. Whatever the reason, fatal process errors are dealt with by re-initializing the user's command environment. This has the unfortunate property of re-initializing the user's ring 3 stack which probably contained the history (or at least clues) concerning the events before the fatal process error occurred. One would like to be able to examine the state of the system before the stack is reset.

Before the Ring Zero Debugger, the most common way of dealing with this situation was to modify code, take a crash dump, and use Autopsy to analyze it. The routine FATALS in Primos is the one that re-initializes the user's ring 3 stack. By changing a nop instruction in this program at label FTL_HLT_ to a halt, one can take a crash dump before the stack history has been erased. Then the stack can be examined with the trace command in Autopsy.

The debugger should prove to be a much faster way to solve this kind of problem since the system need not be coldstarted again and again. A breakpoint could be set on the label in FATALS and then one could use the trace command of the debugger to see what events lead up to the failure. If the problem is not clear from the trace, breakpoints could be set in one the most recent routines on the stack and the process error repeated. When these breakpoints are hit, the process state can be examined for any problems. If the problem has already occurred, then breakpoints need to be set in the previous routine and the process fault repeated. This procedure would go on as many times as needed until the source of the problem is found.

If the stack is so corrupted that trace will not work, the method just decribed can still be used but it is much more difficult. The key is to determine which routines may have been called before the failure. This requires the ability to figure out where some valid or partially valid stack frames are by just looking at the stack in octal. The access and dump commands can be used for this purpose.

When dealing with a corrupt stack, one may also be able to trace up the stack using the search command. This is possible since every stack frame contains the address of the start of the stack frame for the calling procedure (known as the return sb). If one can find the beginning of the first stack frame at the base of the stack, one could search up the stack (higher addresses) for the value of the address of this first stack frame. If one is found, it is very possibly the return sb in the called procedure's stack frame. The lookup_address command can be used to see if the return lb in the stack frame seems reasonable. If so, the address of the start of the second stack frame is now known. The next step is to search up the stack for this address. This process can continue as long as the frames seem to make sense. The place to put a breakpoint is at the last known routine before the process error.

The steps to follow in debugging a fatal process error are summarized below:

1. put a breakpoint at label FTL_HLT_
2. cause the fatal process error
3. use the trace command when the breakpoint is hit
4. if trace doesn't work use access, dump, search, and lookup_address to sequence up the stack
5. if the source of the problem isn't clear, set breakpoints in the most recent routine, repeat the error, and check the process state
6. move down the stack performing the previous step until the offending routine and code is found

4.2.2 System Hangs

One common way for the system to fail is for it to appear to be hung. When in this state, the system appears completely unresponsive to any kind of input for a long period of time. The difficulty here is determining what a long period of time is. In many situations, it may be the case that the system is functioning correctly but is so overloaded that it appears to be hung. Whether this is the case or not, the Ring Zero Debugger is an excellent way to investigate this kind of problem.

The first step is to enter the debugger by invoking it from the system console. If this fails, there is a serious problem. Failure to enter the debugger indicates one of two things: either there is something preventing the clock process from running or the system is hung in microcode. Issuing a stop command to the VCP separates these two cases. If the system halts, then something is preventing the clock process from running. A tape dump should be taken at this point and examined with Autopsy. If the system does not halt, the system is hung in microcode and there may not be much one can do. A tape dump can be taken, but it will be hard to analyze since there will be no register values for recent processes.

Assuming that one has successfully entered the debugger, the next step is to determine information about the state of the active process. The first piece of interesting information relates to the type of the active process. If it is an interrupt process, the system could be hung in a loop due to bad hardware. If it is the backstop, the system could be deadlocked or waiting for an event that will never happen.

Other interesting information about the active process concerns what it was attempting to do. If the process has terminal buffers, one could see the high-level operation that was being attempted by looking at the process's terminal buffers with the ttybuf command. One can also

examine the most recent command line by looking in the cldata common area with the structure command. The current path of execution can be examined by using the trace command. Finally the current procedure and arguments can be determined by using the arguments and status commands.

There are a few other pieces of information about user processes that can be very telling. If process faults are inhibited, it could be the case that the process is looping inside a critical region. The field inhprf in the common area pudcom determines whether process faults are inhibited. It can be examined with the structure command. The status command will indicate whether a process owns any system nllocks or has deferred any process aborts. A positive interval timer value for a process indicates that the process has monopolized the machine for whatever reason (e.g. looping in interrupt inhibited code). This can be seen with the pcb command. Finally, a fault that repeatedly generates another fault before a fault frame can be built will hang the machine. This can be seen by examining the concealed stack with the pcb command.

If the reason for the system hang is not yet clear, the next step is to examine the state of the entire system with different debugger commands. The system ready list is a good thing to inspect for starters. If there are many processes on it then perhaps a high priority process is dominating the machine. If there are no interesting processes on it, then perhaps the system is either deadlocked or waiting for an event that will never happen.

Another important way to examine system status is to examine the state of all the different logged in user processes. This can be done with the user option to the status command. The key here is to determine what the processes are waiting for. A common and uninteresting event to be waiting for is terminal input. A more interesting event might be waiting for a disk request to be satisfied. Another might be waiting for a lock to become available. The question to ask here is whether these are events that will ever happen. The status command can also be used along with the print_locks command to see if any system nllocks are held. Using the nllocks improperly can easily lead to a deadlock situation.

A user that is of particular importance in trying to figure out the state of the system is often user 1. This user provides a number of system services such as flushing the locate buffers to disk. If user 1 runs into problems, the whole system will suffer since user 1 has priority over other user processes.

All the previous user checks should be made on user 1. If the minalm process abort flag is set for user 1, one can infer that no user processes have run for at least the last minute. This could indicate that the system was hung either in an interrupt process or in interrupt inhibited code. Another place to look is at the user 1 message buffer. This can be displayed with the ttybuf command. It may indicate problems that aren't yet known such as unrecoverable disk errors. The reason this information may not be known is that this buffer is only emptied once a minute. Unrecoverable disk errors can seriously affect system

throughput causing one to think the system is hung.

If examining the state of the active process and the whole system in general do not clarify the source of the problem, the next step is to let the system run again and repeat the previously described examinations. There are three separate approaches that could be used. One is to single step the active process repeatedly to see if it is running in a tight loop. This can be done using the breakpoint and single step commands. If a particular area of code is suspected, breakpoints could be set there and one could let the system run with the continue command. If the problem area is completely unclear, one can just let Primos run for a little while longer, reenter the debugger with a console interrupt, and start the investigation over.

Some of the things to look for to investigate system hangs are summarizied below along with the appropriate debugger commands to use. The list below is only meant to show the kinds of information that one might need to determine the cause of a system hang and how to get it with debugger commands. There are countless other approaches.

1. enter the debugger with a console interrupt
2. check the state of the active process
    - o what kind of a process? (status)
    - o what is the process doing? (ttybuf, arguments, trace, status, structure cldata)
    - o any outstanding process aborts or locks held? (status)
    - o are process faults inhibited? (structure pudcom)
    - o is the process interval timer positive? (pcb)
    - o has the concealed stack overflowed? (pcb)
3. check the system state
    - o which processes are on the ready list? (ready_list)
    - o what interesting events are user processes waiting for? (status)
    - o what nllocks are held and by whom? (status, print_locks)
    - o what is user 1 doing? (status, trace, structure, ttybuf)
    - o is user 1's minalm process abort flag set? (status)
    - o are there messages in the user 1 message buffer? (ttybuf)

4. if the source of the problem is unclear, let the system run a little more

   o if a loop is suspected, single step repeatedly (step)

   o if a specific area is suspected, use breakpoints and single steps (breakpoint, step)

   o if no idea as to source of problem, let the system go and restart the whole sequence beginning with step 1. (continue)

4.2.3 System Halts

A system has obviously failed when it halts. Halts basically fall into 2 different categories. On the one hand there are halts that are coded into Primos to stop the system when some internal inconsistency has been detected. With these halts, the path to the halt and the low-level reason for stopping the system are usually known. The other types of halts are locations that contain data that when interpreted as an instruction, turn out to be halts. In these situations, an error has caused a process to erroneously execute some pathological sequence of instructions that led the process to start executing data. In other words, some software or hardware error has caused the machine to "lose its way" and start executing in some unexpected place. These halts are more difficult to deal with since it is usually very difficult to tell what the path to the halt was. Furthermore, unlike coded halts, these halts give no clue as to what was going on when the system halted.

The Ring Zero Debugger can help in analyzing system halts, but there are definitely situations where it will be of limited use. The major reason for this relates to the fact that Primos must be running in order to use the debugger. Therefore, the debugger cannot be used after the system halts unless the system is warmstarted. However, warmstarts completely change the system state and thus warmstarting to get to the debugger is of limited use in investigating a problem that just happened. It is for this reason that the Autopsy program can sometimes prove to be more useful than the debugger in starting to examine certain halts.

An easy first step in determining the cause of a halt, is to install a breakpoint right on the halt instruction and then try reproducing the problem. In many cases this will cause the debugger to be entered. Once in the debugger, the process's stack history can be determined using the trace command. However, there will also be many situations where the system still halts despite the breakpoint. One good reason for this to happen is if the nature of the failure is such that the system halts in random places. Another reason is if the cause for halting was a stack overflow. Breakpoints will not work on a stack overflow halt since breakpoints require stack space to work properly (see section 5.1.2). In these situations, the best approach is to take a tape dump and use the trace command in Autopsy to determine the process's stack history.

Whether one uses the trace command in the debugger or the trace command on a crash dump with Autopsy, the key is to try to figure out enough of what happened to be able to make a reasonable guess as to where one might set breakpoints. The approach will be to try to set breakpoints in the area where it looks like the problem first occurred. In many cases, this area might be the part of a routine that handles either exceptional conditions or errors. If there have been new or changed modules, these routines would certainly be good candidates for breakpoints. Once these breakpoints have been set, the next step is to try to reproduce the halt again.

If the problem can be reproduced, one of two things will happen: either one of the breakpoints will be hit or the system will halt again. If breakpoints are hit, the state of the system can be determined using debugger commands and then new breakpoints set. This may enable one to follow the path to the halt and thus determine the reason for the system failure. On the other hand, if the system just halts again, one will have to coldstart the system (warmstart if possible to save time) and make better guesses about where to install breakpoints.

The process of setting breakpoints and seeing if processes encounter them before the system halts may be very time consuming. It may take many attempts before the actual path taken becomes clear. Random or hard to reproduce halts make the debugging even more difficult. Unfortunately, one just has to perservere. The steps described above are summarized below.

1. set a breakpoint at the halt address
2. try to reproduce the problem (i.e. halt)
3. use the trace command to determine the process's recent history
    - o if the breakpoint was hit, use the debugger
    - o if the system halted, take a crash dump and use Autopsy
4. coldstart the system (or warmstart if possible)
5. set breakpoints near where it first seems like the problem may be occurring
6. try to reproduce the problem
7. system hits a breakpoint or halts
    - o if breakpoints are hit then sequence through the code setting breakpoints and examining system state until the problem is found
    - o if the system halted then make better choices about the location of breakpoints and go back to step 4

4.3 Debugging Hardware

While the Ring Zero Debugger was designed primarily for use in debugging software, it can certainly be used to debug new or even malfunctioning hardware. While it is not the best tool to accomplish this (a hardware level debugger like the FEP and diagnostic test programs are better), it can provide some helpful insights in certain situations.

In the case of new hardware, it is often true that the individual functions or pieces have been tested and seem to work yet when the total system is tested with Primos, it doesn't quite work correctly. This happens often with new processor development. It is difficult for a low-level hardware debugger to find many of these problems because it is not looking at the situation from a level that is high enough (e.g. microsteps versus a disk read operation). However, the Ring Zero Debugger is a perfect tool for looking at the problem from the appropriate level. It can be used to examine the situation from the individual instruction level up to the command line level.

There are a number of advantages to using the debugger to debug hardware whether it is newly designed or not. These include the following reasons.

- o Much of the system can malfunction yet the debugger can still be used. This is due to the fact that the debugger can be entered very early on during a coldstart. Primarily all that has happened at this point is that memory has been sized and process exchange has just been turned on. The debugger only needs the basic processor and console interface to work. Thus a malfunctioning controller won't affect the debugger in most cases.

- o Problems can be diagnosed without special hardware or software. While the typical ways for debugging hardware are generally superior to using the Ring Zero Debugger, they usually require special software and often also require special hardware. Since the debugger is built right into Primos, it is always available.

The debugger should also prove helpful for debugging controllers. By setting breakpoints on the i/o instructions in interrupt processes, one ought to be able to see the exact sequence of instructions being issued to the controller. The debugger also provides the ability to look at the DMA channels with the access_register and system_registers commands. If there was any doubt about the data being transferred correctly from the device into memory with a DMX transfer, a breakpoint could be set right after the i/o instruction which initiates the transfer. Then the DMA channel as well as the actual buffer in memory can be examined to see if it happened correctly.

4.4 Debugging a Customer's System

When customers have problems with their systems, Customer Service gets involved. If the problem is stubborn enough or the customer is important enough, then engineers from Prime Engineering get involved. In somes cases, engineers may even have to travel to customer sites. It is very possible that the Ring Zero Debugger could prove useful in solving certain types of customer problems for both Customer Service and Engineering. However, at this point in time, it is not clear that the debugger will be used for this purpose. The goal of this section is to suggest some ways that the debugger might be used to solve customer problems. The hope is that this use of the debugger will be explored.

In terms of Engineering, an important new ability that comes with the debugger is the means to interactively debug a system remotely. The remote console port of the VCP can be used to run the debugger at customer sites. In certain circumstances, this ability permits Prime engineers to diagnose particularly stubborn problems for important customers without having to make a trip to the site. All of the techniques described in this document for solving problems can then be used not only in lab situations, but also to solve certain problems that occur in the field.

From the Customer Service perspective, the Ring Zero Debugger could be used as a supplement to their existing methods for diagnosing problems. If Customer Service people were taught some basic ways to use the debugger in certain situations, they might be able to easily diagnose some kinds of problems. Some examples of the kinds of situations where the debugger could help to get valuable information follow.

- Machine checks due to a malfunctioning controller. In some situations, a system may fail to coldstart due to a machine check. Upon decoding the dswstat register it may be clear that a controller is causing the problem but it is not clear which controller it is. With the debugger, one could examine the location where the processor was executing when the check occurred (contents of the dswpb register) and determine which routine the process was executing in. From the routine name, it is usually fairly clear which controller is involved. One could also set a breakpoint on the instruction. Once the breakpoint is hit, then one could examine the state of the system (or controller) at that point and also single step to be sure that the instruction is the problem.

- Hung system due to a malfunctioning controller. Sometimes when a controller malfunctions, it will appear constantly "busy" to the system. In Primos, most i/o instructions are coded to spin in a loop if the device is busy. This will cause a system to appear hung. The debugger could be used to determine the process that was executing and where, both of which should indicate the controller that needs replacement.

- Determine information about a halt. When a system halts, it is often very difficult to get even the faintest clue about why it halted based on the halt address. Sometimes a message is printed before the machine halts but many times

not. By coldstarting the system with the debugger, one can look up the name of a symbol which corresponds to the halt address. One may also be able to set a breakpoint at the halt address and enter the debugger just before the machine is about to halt. At that point, some basic debugger commands can be issued to determine more information about why the machine is about to halt. While knowing the halt name, current routine, or executing process may not isolate the source of the problem, it should be somewhat helpful in solving the problem.

One other point is that if Customer Service can use the Ring Zero Debugger to solve certain kinds of problems locally, then they could also potentially diagnose the problem remotely using the remote console port. This might allow them to determine what component might be failing before they make a trip to the customer site.

4.5 Shared Subsystems

The Ring Zero Debugger can be used to set breakpoints anywhere in shared code (dtars 0 and 1). The ring of execution can be either 0 or 3. The original intent here was to be able to debug Primos, but there is no reason that the debugger couldn't also be used on shared subsystems such as ED, EMACS, Midas Plus, Cobol, etc. There are certainly better ways to debug these products, namely debug the code as a non-shared version first with DBG. However, the Ring Zero Debugger could be useful should a bug appear in the shared version but not in the non-shared version.

In debugging a shared subsystem, all features of the debugger will function correctly but there will be no symbolic information available. This is because only the Primos load maps are currently written into the debugger. However, with a modest effort, it is possible to get other load maps written into the debugger. The modest effort involves modifying a program which writes the load maps into a Primos program image just before the mapgen program is run. This program is known as dump_maps and is located in the Primos mapgen subdirectory.

The procedure to follow in adding the symbols of a subsystem to the debugger is listed below.

1. make a load map of the subsystem using the same Seg options as are used to make the Primos load maps
2. modify dump_maps to read in the subsystem map (could be instead of or in addition to the Primos maps)
3. rebuild dump_maps
4. put subsystem map in the directory where Primos is built
5. rebuild Primos

4.6 A New Angle on Performance

There are many ways to gather performance data on a system and the Ring Zero Debugger is certainly the last place one should look. Yet surprisingly enough, there are at least 3 ways the debugger can be used to determine information relating to performance.

One way that it can be used is to determine how many times a certain operation happens to provide a given service. For example, how many disk reads does it take to satisfy an avail command? How many calls to locate does it take to invoke emacs? How many page faults are taken to coldstart the system? To determine the answer to these questions, one need only set a breakpoint on the routine that provides the service (e.g. rrec, locate, or pagtur). The breakpoint would be set with a large proceed count. After the service has taken place, the debugger could be reentered and the proceed count examined. The difference in the proceed count is the number of times the operation took place.

Another way that the debugger can be used to gather performance related date is to determine how many times one code path is taken over another. For example if a routine like prwfSS breaks into separate sections based on an input key, one might wonder how often each key is used. This could be easily determined by again setting breakpoints with large proceed counts in each section. After running some test, the proceed counts can be examined to determine the relative frequency that each routine has been called.

A final way that the debugger could be used for performance is to actually count the number of instructions needed to perform some operation. This can be done with the breakpoint and step commands. Breakpoints can be set at the beginning and end of the operation to measure. When the breakpoint at the beginning has been hit, one can issue a single step command with a large step count. When the ending breakpoint is hit, one can examine the remaining step count. The change in the step count is the number of instructions that were executed to perform the operation. This information could be valuable in trying to improve the performance of a small but frequently traversed area of the system.

Obviously, the examples just given demonstrate that the debugger is a crude way to gather performance data. The usage command, GEM (the General Event Monitor) and PBHIST (the pb histogram program) are certainly better sources for performance information. The reason that this use of the debugger is mentioned here is two-fold. One is that the debugger is a relatively easy way to gather some indication of what the situation is. The other is that the information gathered by the debugger may present a new and different perspective. For example, pbhist may tell one what general area of the system is being heavily traversed, but its granularity isn't very fine. The actual reason for a system dwelling in a certain area of code can be more closely examined with breakpoint and step commands.

5. Implications of the Ring Zero Debugger Design

A goal of the design of the Ring Zero Debugger was to make it as independent from Primos as possible. The degree to which this is achieved is important in being able to debug nearly any part of Primos. If the debugger relied on a particular feature of Primos, then one couldn't use the debugger to test changes to that region of the system without possibly breaking the debugger. Therefore the debugger doesn't rely on any features of Primos.

However, the Ring Zero Debugger cannot claim to be totally stand-alone. It is loaded with Primos and runs as a special Primos process. To achieve functionality such as breakpoints, it must deal with Primos in what are certainly non-obvious ways. While a user of the Ring Zero Debugger may not be especially interested in its design, the user is certainly interested when the design imposes certain limitations on the debugger's use. That is the topic of this chapter.

5.1 Effect of Breakpoints

The most difficult and complex part of designing the Ring Zero Debugger was supporting breakpoints. Achieving this functionality without any hardware support was extremely challenging. The final design was largely successful in supporting breakpoints but there are certain times and places where they cannot be used and other times when their use may be confusing. Most of these situations are discussed in the following sections.

5.1.1 General Effects

Before describing some of the limitations of breakpoints, a quick sketch of the way that breakpoints are implemented may help. When a breakpoint command is issued from the Ring Zero Debugger, basically two things happen. One, is that the debugger builds code to simulate the breakpointed instruction. The other is that the original instruction is replaced by an instruction which will cause an illegal instruction fault when encountered. When the breakpoint is hit, the breakpointed process takes a fault. Then the fault handler calls a routine which wakes up the debugger process (which runs as the highest priority process in the system). When one decides to leave the debugger, it manipulates the two extra stack frames previously pushed onto the stack of the breakpointed process. The effect of this manipulation is to cause the process to execute the simulated code and then return to the instruction after the breakpointed one.

This scheme works most of the time but not always. The following paragraphs describe situations where there are limitations.

- Certain instructions are not supported. Certain instructions do not allow breakpoints for a variety of reasons. If one attempts to set breakpoints on any of

these instructions, the user will get an error message. The list of instructions is ARGT, CALF, E16S, E32I, E32R, E32S, E64R, E64V, IRTC, IRTN, LPSW, STEX, and SVC. The quad floating point instructions are also not supported.

- o Certain places must be avoided. Because the debugger is implemented as a separate process, breakpoints cannot be set in places where process exchange is disabled in any way. These places include check handling code, phantom interrupt code, and the tape dump program. It also includes both coldstart and warmstart code before process exchange is turned on. An unrelated case is that breakpoints cannot be set in the frontstop process. (This is only a constraint for dual-processors such as the P850.) The debugger cannot detect any of these situations and therefore will just fail to work properly. Breakpoints also cannot be set in the debugger itself or the illegal instruction fault handlers. These two cases can however, be detected by the debugger.

- o Interrupt inhibited code should be avoided. Critical regions are most often achieved by surrounding the code with interrupt inhibiting instructions such as INHL, INHP, or INHM. Setting a breakpoint in code where interrupts are inhibited will break the critical region. The code to simulate any instruction in the region will work properly, but in the process of servicing the breakpoint, interrupts will be enabled. This is due to the process exchange needed to invoke the debugger. The Prime architecture enables interrupts on a process exchange. Enabling interrupts will defeat the reason for having the critical region. There is no way for the debugger to detect this condition. It is up to the user to either avoid this situation or at least be aware of the consequences. (For example, if it is clear that only one process is executing in the critical region at a time, then the region needn't be avoided.)

- o The whole system slows down. Servicing breakpoints can markedly slow the system down. The degree to which it is slowed is related to the frequency of breakpoints hit. Breakpoints which are set in code that will be executed on every tick of the real-time clock can cause throughput to virtually stop on some processors. In these cases, virtually all processing time is spent servicing the breakpoints. On the other hand, if a breakpoint is hit only a few times a second, it will scarcely be noticed. Some of the effects of slowing down the system are lower throughput, loss of time on the system time-of-day clock, and loss of characters on the system console.

- o Can't modify memory where breakpoints are set. If one attempts to modify a location where a breakpoint is set, the debugger will give the user an error. This situation is rather difficult for the debugger to deal with so no attempt has been made to do so.

- o Won't work with self-modifying code. Breakpoints that are set in code that is self-modifying will not work. When the code modifies itself, it will overwrite the illegal instruction and there is no way that the debugger can determine that this has happened.

- o Breakpoints do not instantly suspend the system. A system is in a sense "suspended" when the debugger process runs since it is the highest priority process

in the system. However, this does not happen the instant a breakpoint is encountered. The illegal instruction fault must be handled by building a fault frame and then calling a routine which eventually invokes the debugger process. If the breakpoint is in ring 3 there is even more code to pass through. While this code is being executed, the breakpointed process must compete with the rest of the processes for system resources as it does ordinarily. Thus other processes can potentially run after a certain process has hit a breakpoint.

5.1.2 Stack Implications

As noted in the previous section, when a process hits a breakpoint, it will push two new stack frames onto the user's stack. The first stack frame is a fault frame for the illegal instruction fault. The second frame is built when the illegal instruction fault handler calls a routine which will invoke the Ring Zero Debugger process. These facts are mostly irrelevant to a debugger user except for the following cases.

- o The extra stack frames can sometimes be seen. When one is in the debugger and examines a process which has just hit a breakpoint, the added stack frames to service the breakpoint will not be seen. This is because the debugger takes this into consideration by looking back down the stack to get the values of the process's registers at the time of the breakpoint. However, if another process is also hitting a breakpoint but has not yet invoked the debugger, one will see the extra frames on this process's stack. In addition, if the user examines register values for this process, the values shown will not be the values at the time of the breakpoint. This won't cause any problems but is mentioned here to avoid confusion.

- o Primos stacks are more likely to overflow. Pushing extra stack frames to service breakpoints means that Primos will use more stack space than it otherwise would. In most cases this doesn't matter since the stacks are relatively large. However, the page fault stack is quite small and the addition of extra frames can sometimes cause it to overflow. When this happens, the system halts at R0OVR__. There is little that can be done in this case other than trying to limit the number of breakpoints in routines that will concurrently share this stack.

- o A corrupt stack can ruin debugging. The Ring Zero Debugger needs to use a process's stack to service breakpoints. If this stack has been corrupted, breakpoints will not work correctly. This may even cause fatal errors for the debugger. A common way to allow the stack to become corrupted is to fail to allocate enough stack space for an assembly language routine.

- o Deadly embrace in the page fault handler. Breakpoints can be set in most of the ring zero fault handlers, but there can be a few obscure problems. An example of one relates to a small section of code in the page fault handler. A breakpoint in this section may cause one to hang the system. The section referred to is the code that preceeds the CALF instruction in the page fault handler. The root of the problem is not issuing a CALF immediately. This means that the code up to the CALF instruction is not using the page fault stack. Thus, if a breakpoint is set in this code and the page fault happens to be for the next page of the current stack,

the system will be in an infinite loop. When it hits the breakpoint, the illegal instruction fault handler will try to push a fault frame on the current stack. However, this will just cause another page fault and the cycle begins again. This is a relatively unlikely event, but it can happen.

- o WAIT's on the interrupt stack are a problem. There is one stack in the system shared by all the interrupt processes known as the interrupt stack. All these processes can share this stack because none of the processes leave any stack history before executing a wait instruction. However, if a breakpoint is set on one of these wait instructions, it would leave two extra stack frames before waiting. This would cause the stack to become corrupted. The debugger can detect this condition. It will clear the breakpoint and print an error message when this happens.

5.1.3 PCL Instructions

There is no instruction on Prime machines that is more complex than procedure call (PCL). Breakpoints for this instruction proved to be very difficult to implement. The way that was chosen is different from the method described earlier for all other breakpoints. Instead of replacing the PCL instruction with an illegal instruction, the ECB of the routine that the PCL is calling is overwritten with new values. These new values cause the debugger to be invoked rather than calling the actual routine. When the debugger is later exited, the process's stack is manipulated so that control will pass to the routine originally called by the PCL. This method seems to work fairly well but has some quirks that should be explained.

- o The ECB is modified. The program counter and keys of the routine called by the PCL will be modified when a breakpoint is installed. Don't be alarmed.

- o The PCL has already executed. With most breakpoints, the instruction breakpointed has not yet executed. This is not true for PCL instructions because of the way that they are implemented. With PCL's, the instruction has mostly completed but one has not started to execute in the called routine. This means that the new stack frame has been built and the argument pointers have been put into it. The debugger tries to make this situation transparent by looking in the stack frame to get the register values before the PCL executed. It mostly succeeds except that the program counter (PB) has been advanced past the PCL instruction.

- o A breakpoint on a PCL won't be seen if the instruction fails to complete. This point is sort of a corollary of the previous point. Because the pcl instruction has nearly completed before the breakpoint is encountered, anything which prevents this instruction from completing also prevents the breakpoint from being seen. A common reason for failing to complete the instruction is some sort of fault which cannot be serviced. Page faults and pointer faults are frequent events when executing a pcl instruction because of the transfer of the argument pointers. If, however, the pointer fault cannot be resolved, the instruction will not complete and the breakpoint will never be seen.

5.1.4 Single Stepping

The ability to do single step's is clearly a useful ability to have for debugging and thus this capability is part of the Ring Zero Debugger. However, this ability is of much less utility for users of the Ring Zero Debugger than it would be for other types of debuggers due to certain limitations. These limitations relate to the designs of the processor architecture, Primos, and the Ring Zero Debugger itself. These limitations are discussed below.

- o Can only step from breakpoints. A user of the debugger can only issue the step command if the debugger was entered from a previously set breakpoint (or step). Thus if the debugger is entered due to a console interrupt, one must set a breakpoint, leave the debugger, and have the breakpoint be hit before a single step can be issued. The reason for this is related to the debugger design for doing next-instruction prediction. The debugger will issue an error message should a user attempt a step without having entered the debugger from a breakpoint.

- o Only one process can step at a time. When one issues the step command it refers to a single process. If the debugger is entered before this process completes its step operation (for whatever reason), another step command cannot be issued without getting an error message. The first step command must complete or be cleared before another one can be issued. Supporting many, actively stepping processes in the system at the same time would have made the debugger design very much more complex.

- o Step aborts for various reasons. Step is a relatively useful command when used with small step counts. However, if one gives it a large count, one sees before very long that it will probably abort with an error message. This is due to the fact that there are certain conditions that occur frequently in code that force the step command to abort. The reasons for this relate primarily to the design of Primos. Most of the reasons for aborting are listed below.

  - . Interrupt inhibited code. As previously discussed in section 5.1.1, a breakpoint inside interrupt inhibited code can break the critical region. Therefore the debugger will abort a step operation if it encounters an interrupt inhibit instruction.

  - . Private address space code. The debugger was not designed to allow breakpoints in a process's private address space. Thus if one tries to step past an instruction that calls or jumps into a process's private address space, the step command will abort with an error message.

  - . Machine mode changes. The Ring Zero Debugger is written for V-mode only. If one attempts to change the mode of the system with mode changing instructions or by calling a routine with a mode other than V-mode, the step command will abort with an error message.

  - . Unable to set breakpoints. If the debugger cannot set a breakpoint on a particular instruction, then the step command must abort. Some of the reasons

for not being able to set a breakpoint are described in section 5.1.1.

5.2 The Issue of Non-resident Memory

One of the important features of Prime systems is the ability to support virtual memory. There is a great deal of code in Primos to support this functionality. This code makes non-resident memory become resident if it is referenced. In order to have this ability in the Ring Zero Debugger, the debugger would either have to use the functionality in Primos or duplicate this functionality itself. If Primos was used by the debugger to reference non-resident memory, then the debugger would no longer be stand-alone. This would mean that it could not be used to debug certain parts of Primos. Duplicating virtual memory support in the debugger is by no means feasible. Therefore it was decided that the Ring Zero Debugger will not have the ability to access non-resident memory.

The effect of the decision to not have the ability to reference non-resident memory is that most Ring Zero Debugger commands will abort with a page fault error should non-resident memory be referenced. To lessen the impact of this restriction, the debugger is designed so that breakpoints can be set in non-resident memory and the let command can modify non-resident memory.

This ability is achieved by having the Primos page fault handler invoke the Ring Zero Debugger on every page fault when non-resident memory prevents these two commands from completing. When the correct page is made resident, the commands are completed. Of course, all this manipulation is transparent to the user except when errors are involved.

The advantage of adding this "pending" ability for installing non-resident breakpoints is that if everything goes well the user can't even tell whether the breakpointed memory was resident. The disadvantage of pended breakpoints is that only limited error checking can take place at the time the command is issued. If the specified location does not represent a valid instruction, this can only be determined when the non-resident page is made resident. However, this will not be when the breakpoint command was issued but some time later after the debugger has been exited.

Errors during pended breakpoints may be somewhat surprising to the user. First the error will be shown possibly long after the breakpoint command has been issued but before any process has hit the breakpoint. Secondly, the stated reason for entering the debugger will be because of a page fault. This is the only way for the user to ever see the debugger entered this way.

5.3 Using a Separate Process

The Ring Zero Debugger is implemented as a separate process which has the highest priority of any process in the system. This fact together with the inhibiting of interrupts means that the debugger will never be preempted by another process or even by phantom interrupt code. There are a couple of implications of this design.

- o The debugger will fail if process exchange data bases are corrupted. Process exchange is implemented in firmware but relies on the integrity of data bases such as the ready list, PCB's, and semaphores. If any of these are corrupted, the debugger may be unable to run. In fact it is common in such circumstances for the machine to be hung in a microcode loop.

- o The debugger cannot be entered without Primos running. With stand-alone VPSD, one can enter VPSD after a machine halts. This is because it does not use process exchange. Invoking VPSD simply means jumping into some code which turns on segmentation and lets one exmaine memory. The debugger does need to have process exchange on and thus Primos must be running. If the machine halts or is halted while in Primos, Primos must be restarted by either warmstarting, coldstarting , or issuing a run command (if manually halted) before the debugger can be used.

- o The VCP commands display and displayc won't work while in the debugger. Display and displayc are VCP commands that allow one to examine resident memory locations while Primos is running. In order to accomplish this, the VCP relies on the Primos clock process to read the sense switches and output values in the "lights". While the debugger is running, the clock process is not. Therefore display and displayc will not work correctly.

5.4 Warmstart

Very little is written, or for that matter known, about warmstart. It is a means of attempting to restart the system right where it left off when it was stopped. The Ring Zero Debugger can be used in many cases where warmstarts are attempted, yet there are some clear restrictions.

Stating that the debugger can be used with warmstarts refers to two basic abilities. The first ability is that breakpoints can be set in warmstart code after process exchange has been turned on. This code can be debugged just like any other code. The second is the ability to warmstart while in the debugger. This means that if the debugger process is running when the warmstart occurs, a warmstart procedure can be done successfully. One can be in the debugger process either by just "passing through it" to service a breakpoint or by being in the debugger command level. In the first situation, one will see the warmstart happen immediately. In the second case, one will resume execution back in the debugger and will not see the warmstart until the debugger is exited.

There are, however, some situations where warmstarts will not work with the Ring Zero

Debugger. These are listed below.

- o Breakpoints executed by interrupt processes may fail after a warmstart. An unfortunate aspect of the warmstart code is the way that it deals with the interrupt stack. This code resets the interrupt stack, effectively destroying any current state information. Because breakpoints need to use extra stack frames (see section 5.1.2), any process which is servicing a breakpoint and thus using the interrupt stack, may fail after a warmstart.

- o Warmstarts don't work at all on the P850. The reason for this is currently unknown.

5.5 Effect on the Primos Load

The Ring Zero Debugger is loaded as part of the Primos ring zero load, but in a rather unusual way. The debugger is loaded first before the rest of Primos. The unusual part is that after the debugger is loaded, nearly all the names of debugger routines must be removed. This means that the debugger can see any name in Primos but Primos can only see a few, explicitly stated names in the debugger. The main reason for this is that the debugger uses a few Primos routines including the PLP libraries. Because the debugger must reside in wired memory, wired copies of these routines must exist. The best way to accomplish this is to load separate copies of these routines with the debugger. However, this dictates that the debugger have its own, separate namespace and thus the just described loading sequence is used. This design gives rise to a few issues.

- o Changing certain Primos routines can break the debugger. There are a few Primos routines that are loaded both with the debugger and with Primos. Primarily these include the PLP libraries, IOAS, conversion routines such as CHSFX1, and some page map primitives. (A complete list of these routines can be found in the load.) If any of these routines are modified incorrectly it will affect the operation of the debugger, possibly breaking it. If these routines need to be changed and debugged, one could modify the load so that the debugger picks up the unmodified version and Primos uses the new version.

- o Many debugger global entrypoints appear to be unresolved. If one examines the output from the Primos load, one will see many unresolved names immediately after the debugger is loaded (but before the rest of Primos has been). Unfortunately, this is a normal situation brought about by the peculiar way that the debugger must be loaded. These unresolved names concern Primos data bases that the debugger must know the locations of. After the rest of Primos is loaded, all these names should be resolved.

- o A separate load map for the debugger is not generated. Most Ring Zero Debugger symbols do not appear in the Primos load map. Basically the debugger has its own, separate namespace. What would be desireable, then, is two load maps: one for the debugger and one of Primos. Unfortunately, it is not possible to do this

with SEG without generating an error. Therefore, it was decided that no load map would be generated for the debugger. However, one can generate a map for the debugger by making changes to the ring zero load file. (See appendix F.3).

Appendix A
Finding Variable Information from Listings

Program variables must be manually defined before they can be referenced (see section 3.6). In order to do this, one must first know what the actual definition of the variable is. The definition of a variable relates to where it is in memory and what the data type is. This information can be found in the cross reference of a listing of the program. The following discussion describes how this information can be extracted from a listing. Before describing how to get program variable information from a listing, some general comments about the way variables are defined should be helpful.

Variables in programs generally fall into two classes based on how they are allocated. Either they are statically allocated in memory (e.g. PL1 static variables or Fortran common areas) or they are dynamically allocated on the current stack frame (e.g. PL1 automatic variables). Variables that are static are usually defined in terms of the linkage area of a program. At Prime, this means that when the program is being executed, the static variables are referenced as offsets from the linkage base register (LB). Dynamically allocated variables on the other hand are usually expressed as offsets from the stack base register (SB). Lastly, variables in common areas are referenced as an offset from the start of the common area.

In order to define program variables in the Ring Zero Debugger, a program listing must be consulted. The listing will indicate the type of the variable and how it is defined. The key is to know how to get this information from the cross reference listing for the langauge that the code is written in. The following examples will show how to get this information from listings in plp, pma, and Fortran. Each example will show how certain variables were actually defined in a program and then what was produced in the cross reference listing.

PLP Listings

The following lines were removed from the module prwfSS in Primos. They show the procedure definition and some variable declarations within the program.

```
prwf$$:
       proc (xkey, xunit, xbuf_ptr, xbuf_len, xposition,
             xretwords, xcode) options (nocopy, gate);
  dcl xunit fixed bin(15),
      xbuf_ptr ptr options (short),
      xbuf_len bit(16) aligned,
      xposition fixed bin (31),
      xretwords fixed bin,
      xcode fixed bin;
  dcl key_action fixed bin,
      old_cur_ro fixed bin (31),
      ldptr ptr options (short),
      transaction_locked bit(1) aligned,
      nrecs fixed bin(31);
```

The corresponding lines from the cross reference listing are shown below.

```
001 000035S  KEY_ACTION  bin(15) automatic
001 000154S  LDPTR  pointer automatic
001 000160S  NRECS  bin(31) automatic
001 000126S  OLD_CUR_RA  bin(31) automatic
001 000042S  TRANSACTION_LOCKED bit(1) aligned automatic
001 000101S  XBUF_LEN  bit(16) parameter
001 000076S  XBUF_PTR  pointer parameter
001 000112S  XCODE  bin(15) parameter
001 000070S  XKEY  parameter
001 000104S  XPOSITION  bin(31) parameter
001 000107S  XRETWORDS  bin(15) parameter
001 000073S  XUNIT  bin(15) parameter
```

The key to reading the listing is that there is the letter S following the second column of numbers. This means that the variable is stack based and the number is the offset from the base of the current stack frame. For example, variable "key_action" is at offset 35 in the stack frame. Thus, one could describe the location of key_action by the address-expression "SB%+35". In fact this would be the way that key_action would be defined using the define_command command.

Another point about the cross reference listing concerns variables that are labeled as "parameter". These variables represent arguments passed to the current program. However, the actual value of the argument does not exist at the given stack location. This location contains the address of the argument (since the pcl instruction passes pointers to the arguments not the arguments). Thus one would have to go through one level of indirection to determine the true value of the variable.

Another segment of the same cross reference listing is shown below to illustrate another point. This example shows the way that a common area is represented (in this case pudcom). If one wanted to define a variable that existed in a common area, one could express it as an offset from the global symbol for the common area. For example, the address-expression for variable "cusr" would be pudcom+10.

```
001          1 PUDCOM  external
    000000+00   2 FREE_PTR  pointer
    000002+00   2 EXT_PTR  pointer
    000004+00   2 STK_RES(1:2)  bin(15)
    000006+00   2 PGFSPB  pointer
    000010+00   2 CUSR  bin(15)
    000011+00   2 PCBUSR  bin(15)
    000012+00   2 UTBLPTR  pointer
```

PMA Listings

To illustrate program variables in PMA, the following lines were taken from the Primos routine pgmapa. Most of the lines demonstrate the use of the dynm pseudo-op to define stack based variables.

```
ENT    SDWADR
DYNM   XPTR(3)
DYNM   XUSER(3)
DYNM   XBSAVE(2)
DYNM   TEMP(2)
DYNM   SDW_ADDR(2)
DYNM   PPN
DYNM   MAP_OFF
DYNM   PME_OFF
DYNM   MAP_PME(2)

SDWADR ECB    SDWX0,XPTR,2
```

The corresponding lines from the cross reference listing for pgmapa are shown below. It should be easy to see that all the stack based variables show an offset followed by an S. Thus an address-expression to define the location of the variable "map_off" would be SB%+27. However, some of the stack based variables represent argument pointers and thus only contain the address of the value of the argument. This is true of variables "xptr" and "xuser". Also shown in this listing are some variables in global common areas. These are listed as an offset followed by the letter C. In this particular example, the actual common is, again, pudcom. Thus an address-expression to reference variable "absave" is pudcom+20.

```
ABSAVE     000020C 0051  0051
HMAPSK     000062C 0051  0051
MAP_OFF    000027S 0045  0215  0221
MAP_PME    000031S 0047  0229  0233
PME_OFF    000030S 0046  0222  0237
PPN        000026S 0044  0208  0227
SDW2       000202C 0051  0051
SDW_ADDR   000024S 0043  0194  0198
TEMP       000022S 0042  0136  0142  0155  0157
XBSAVE     000020S 0041  0081  0090  0123  0143
XPTR       000012S 0038  0074  0091  0114  0160
XSAVE      000065C 0051  0051
XUSER      000015S 0039  0148  0154
```

Fortran Listings

Finally, the following lines will show how Fortran indicates variable definitions. These code lines were taken from the Primos program ta$.

```
 INTEGER FUNCTION TA$ (XLINE,XSTATE,UKEY,FNAME,FNAMEL,
X                     ATSW,CODE)
 INTEGER  XLINE(41), XSTATE(2), UKEY, FNAME (16), FNAMEL,
X         CODE
 LOGICAL  ATSW
 INTEGER I,FLEVEL,ICODE,CHARPT(5),BUFF(BUFSIZ),
X        INFO(8),JNKNAM(16),LTB,RTB,TPARS$,TYPE,LEVEL,
X        LDISK,PASSWD(16)
 INTEGER*4 TNPTR
 DATA LTB,RTB/:274,:276/
```

The corresponding lines from the cross reference listing are shown below. The second column shows the variable type (e.g. I=integer, L=logical, J=integer*4). The third column shows the

basis for the definition. Both "argument" and "stack" mean that the displacement given in the fourth column is relative to the stack. The value "linkage" means the displacement is relative to the linkage area. Finally, if a name appears inside slashes, the offset is relative to a common area whose name is in the slashes.

```
ATSW    L ARGUMENT 000051   0032S  0035S  0075M
CODE    I ARGUMENT 000054   0032S  0034S  0074M
FNAME   I ARGUMENT 000043   0032S  0034S  0103M
FNAMEL  I ARGUMENT 000046   0032S  0034S  0076M
INFO    I STACK    000136   0065S  0088A  0090
JNKNAM  I STACK    000146   0065S  0123A  0125A
LDISK   I STACK    000027   0065S  0101M  0130M
LEVEL   I STACK    000023   0065S  0073M  0139M
LTB     I LINKAGE  000400   0065S  0069I  0159
MDVNO   I /LSMCOM/ 002232   0052S  0189
PASSWD  I STACK    000166   0065S  0104M  0117A
RTB     I LINKAGE  000401   0065S  0069I  0138
TA$     I STACK    000031   0032S  0258M
TNPTR   J STACK    000206   0067S  0087M  0096M
UKEY    I ARGUMENT 000040   0032S  0034S  0072
VDNAM   I /LSMCOM/ 002233   0052S  0191A
XLINE   I ARGUMENT 000032   0032S  0034S  0088A
XSTATE  I ARGUMENT 000035   0032S  0034S  0088A
```

A few examples from the previous cross reference may be helpful. An address-expression for the variable "info" is sb%+136. The variable "ltb" is defined in the linkage area so it would be lb%+400. For a variable in a common area, "mdvno" would be defined as lsmcom+2232. As with all languages on a Prime system, any variables that are arguments contain pointers to the true argument values. Thus a pointer to the value for argument "code" is at sb%+54.

Appendix B
Command Syntax

The syntax for the Ring Zero Debugger is listed below. Both upper and lower case input is allowed. The convention in the specification below is that commands and arguments may be abbreviated to the substring of those letters which are capitalized.

```
<command>      ::= : <variable> <variable-type-option> |
                   Access <address-expression> |
                   Access_REGister <access-registers> |
                   Access_TYPE <access-option> |
                   ARGumentS |
                   BReaKpoint <break-expression> <proceed-count> |
                   CLeaR <break-expression> |
                   CLeaRAll |
                   Continue |
                   DEFine_COMmand <command-name> <command-list>
                   DEFine_VARiable <qualified-variable>
                        <address-expression> <variable-type-option> |
                   DELete_COMmand <command-name-option> |
                   DELete_VARiable <variable-option> |
                   DISplay_COMmand <command-name-option> |
                   DISplay_VARiable <variable-option> |
                   Dump <address-expression> <address-expression> |
                   Help <help-option> |
                   LET <variable> = <new-value> |
                   LIST <break-expression> |
                   LiSTAll |
                   LOOKAT <process-option> |
                   Lookup_Address <address-expression> <symbol-type> |
                   Lookup_Symbol <symbol> |
                   Pcb <process-option> |
                   Print_LOCKS |
                   ReaDY_LiST |
                   SeaRCH <address-expression> <address-expression>
                          <search-pattern> |
                   STATus <status-options> |
                   Step <step-count> |
                   STRUCture <structure-option> |
                   SYStem_REGisters |
                   Trace <trace-options> |
                   Translate_to_PHYSical <address-expression> |
                   Translate_to_VIRtual <physical-page-number> |
                   TTybuf <process-option> |
                   WHere
```

```
<variable>     ::= PL1 identifier

<variable-type-option> ::= <variable-type-specification> | <empty>

<address-expression> ::= <virtual-address> | <symbolic-expression> |
                         <base-register-expression>

<access-registers> ::= A | B | L | E | X | Y | PB | SB | LB | XB |
                       DTAR0 | DTAR1 | DTAR2 | DTAR3 | KEYS |
                       MODALS | OWNER | FCODE | FADDR | TIMER |
                       FAR0 | FLR0 |FAR1 | FLR1 | 0 | 1 | ... | 77

<access-option> ::= <access-type> | <empty>

<break-expression> ::= <process-number> : <breakpoint-address> |
                       <breakpoint-address>

<proceed-count> ::= <decimal-number> | <empty>

<command-name>  ::= string

<command-list>  ::= <command> ; <command-list> | <command>

<qualified-variable> ::= <procedure-name>\<variable> | <variable>

<command-name-option> ::= <command-name> | <empty>

<variable-option> ::= <variable> | <empty>

<help-option> ::= <command> | <empty>

<new-value>    ::= a value expressed in the type of its target

<process-option> ::= <process-number> | <empty>

<symbol-type> ::= ANY | PB | ECB | LB | LBN | COMMON | OTHER |
                  <empty>

<symbol>       ::=  names from the Primos load maps

<search-pattern> ::= 'string' | <octal-list> <search-mask>

<status-options> ::= <process-number> | ALL | USer |
                     INTerrupt | <empty>

<step-count>  ::= <decimal-number> | <empty>

<structure-option> ::= <definition> <address-expression> |
                       <definition> | <empty>

<trace-options> ::= <process-number> <address-expression> |
                    <process-number> | <empty>

<physical-page-number> ::= <octal-number>


<variable-type-specification> ::= <variable-type> |
                                  <variable-type> <variable-length>

<virtual-address> ::= <segment>/<octal-number> |
                      <segment>/<octal-number> <addop> <octal-number>
```

```
<symbolic-expression> ::= <symbol> | <symbol> <addop> <octal-number>

<base-register-expression> ::= <base-reg> |
                               <base-reg> <addop> <octal-number>

<access-type> ::= Ascii | Bit | Decimal | Hex | Octal | Symbolic

<process-number> ::= <decimal-number>

<breakpoint-address> ::= <virtual-address> | <symbolic-expression>

<decimal-number> ::= -32768 to 32767

<procedure-name> ::= procedure from Primos load map

<octal-list>  ::= <octal-number> <octal-list> | <octal-number>

<search-mask> ::= & <octal-list> | <empty>

<definition>  ::= CLDATA | DISK_QUEUE_BLOCK | ECB | FIGCOM |
                  PUDCOM | PUSTAK | SUPCOM | UPCOM

<octal-number> ::= 16 bit octal number

<variable-type> ::= Ascii | Bit | char_Vary | Decimal | Octal |
                    Pointer

<variable-length> ::= <decimal-number>

<segment>     ::= 12 bit octal number

<addop>       ::= + | -

<base-register> ::= SB% | LB% | XB% | .

<empty>       ::=
```

Appendix C
Assembly Language Syntax

The syntax for the assembly language accepted by the Ring Zero Debugger is listed below. The definition is for V-mode only as this is the only mode that the debugger supports. Symbolic assembly language input is only an issue for the access command.

```
<symbolic-input>  ::= <generics> | <skips> | <decimal> |
                      <character> | <generic_ap> | <branches> |
                      <shifts> | <field> | <mem_refs> |
                      <arg_ptrs>

<generics>  ::= <generic-mnemonics>

<skips>  ::= <skip_mnemonics> | <skip-mnemonics> <bit-number>

<bit-number>  ::= 1 to 16

<decimal>  ::= <decimal-mnemonics>

<character>  ::= <character-mnemonics>

<generic-ap>  ::= <generic-ap-mnemonic> <generic-ap-value>

<generic-ap-value>  ::= <base-reg-expression> <bit-expression>
                        <generic-ap-options>

<base-reg-expression>  ::= <base-register> <addop> <word-number> |
                           <base-register> | <word-number>

<base-register>  ::= PB% | SB% | LB% | XB%

<addop>  ::= + | -

<word-number>  ::= 16 bit unsigned number

<bit-expression>  ::= + <bit-number> B | <empty>

<generic-ap-options>  ::= ,* | <empty>

<branches >  ::= <branch-mnemonic> <word-number>

<shifts>  ::= <shift-mnemonic> <shift-count>

<shift-count>  ::= a number from 0 to 63

<field>  ::= <field-mnemonic> <field-operands>

<field-operands>  ::= <far> | <flr>

<far>  ::= 0 | 1

<flr>  ::= 0 | 1

<mem-refs>  ::= <mem-ref-mnemonic> <length-specifier>
                <mem-ref-operand>

<length-specifier>  ::= # | % | <empty>
```

```
<mem-ref-operand>  ::= <pc-relative> | <base-reg-relative>

<pc-relative>  ::= <pc-displacement> |
                   <pc-displacement> <pc-options>

<pc-displacement>  ::= * <oddop> <pc-bounds> | <valid-displacement>

<pc-bounds>  ::= a number between -223 and 255

<valid-displacement>  ::= a number - (current program counter) is
                          within <pc-bounds>

<pc-options>  ::= ,* | ,X | ,*X

<base-reg-relative>  ::= <base-reg-expression> <base-reg-options>

<base-reg-options>  ::= ,X | ,Y | ,* | ,*X | ,*Y | ,X* | ,Y*

<arg-ptrs>  ::= AP <base-reg-expression> <bit-expresion> <ap-options>

<ap-options>  ::= ,S | ,SL | ,* | ,*S | ,*SL | <empty>

<generic-mnemonics>  ::= A1A | A2A | ACA | ADLL | ARGT | . . .

<skip-mnemonics>  ::=  DRX | IRX |SAR | SAS | SGT | . . .

<decimal-mnemonics >  ::= XAD | XBTD | XCM | XDTB | XDV | . . .

<character-mnemonics>  ::= ZCM | ZED | ZFIL | ZMV | ZMVD | . . .

<generic-ap-mnemonics>  ::= ABQ | ATQ | CALF | INBC | . . .

<branch-mnemonics>  ::= BCEQ | BCGE | BCGT | BCLE | . . .

<shift-mnemonics>  ::= ALL | ALR | ALS | ARL | ARR | . . .

<field-mnemonics>  ::= ALFA | EAFA | LFLI | . . .

<mem-ref-mnemonics>  ::= ADD | ADL | ANA | ANL | CAS | . . .
```

Appendix D
Error Messages

Various types of errors can be encountered in the course of using the Ring Zero Debugger. The different classes of errors are described in section 2.9. This appendix describes the errors known as user errors. It also includes warnings.

A breakpoint already exists at specified address.
An attempt was made to install a breakpoint at a location where one already exists.

A command definition has not been specified.
The define_command command has been issued but is missing arguments which will define the action of the new user-defined command.

A command name has not been specified.
The define_command command requires arguments which specify the command name and the definition of this user-defined command.

A Primos symbol is required as an argument.
This command expects a symbol from the Primos load maps but none was given.

A private address space does not exist for the active process.
The current command is referencing a private address space but the active process is an interrupt process. The key is that interrupt processes don't have their own private address spaces.

A segment offset is not a valid address expression.
A number representing an offset within some "current" segment is not a valid form of an address-expression. See the definition of address-expression.

A valid address expression must be specified.
The given command expected an address-expression argument but none was given.

A valid register name must be specified.
The access_register command requires a valid register name as an argument. See the definition of this command.

An octal number is expected.
The current command expected an octal number but didn't get one.

An unpend operation may be erroneously ignored.
This warning indicates that an operation that was "pended" due to non-resident memory, may never become properly "unpended" when the page becomes resident. Pended operations usually refer to breakpoints. The expectation is that this warning will seldom happen, but if it does, the breakpoint should be cleared and re-installed.

Bit numbers must be between 1 and 20 in octal.
The bit number operand specified is not in the correct range or radix.

Branch addresses must be 16 bit octal numbers.
The operand to a branch instruction must be an offset within the current segment. It cannot be a relative address (e.g. *-5).

Breakpoint table is full. Command ignored.
The number of breakpoints currently defined equals the maximum number allowed. To define another breakpoint, one must be deleted from the table. The attempt to set another breakpoint did not cause one to be installed.

Breakpoints are not allowed for this instruction.
Breakpoints are not implemented for certain instructions. The list of instructions is given in section 5.1.1.

Breakpoints are not allowed in debugger code.
Breakpoints cannot be set within the debugger code. They also cannot be set in code which is needed to invoke the debugger when a breakpoint is hit. Code in this category includes the gate added to support ring 3 breakpoints and all illegal instruction fault handlers.

Breakpoints are not allowed in private segments.
The debugger was not designed to allow breakpoints in private segments.

Breakpoints are not supported for this form of memory reference instruction.
The form of memory reference instruction refers to the base register relative form that are indexed with a displacement of 7 or less. These are one word instructions that use the contents of the x register to determine how to form the effective address! It is truly amazing that any form so cryptic was ever defined in the architecture. Breakpoints for such a ridiculous form are impossible.

Breakpoints cannot be set on an ARGT instruction.
An argt instruction is not an executable instruction. Any breakpoint should be set on the next instruction instead.

Breakpoints not allowed on LDLR/STLR KEYS.
Due to an architectural problem, neither the ldlr nor the stlr instructions can support breakpoints if the register being referenced by these instructions is the keys/modals register. Otherwise breakpoints can be set on these instructions.

Can't continue from breakpoints set on WAIT instructions that use the interrupt stack. Breakpoint/step cleared.
One of the basic rules of using the interrupt stack is that no process which uses it should ever leave any stack history on it. In other words a wait instruction should not be executed if the process has active stack frames on the interrupt stack. Setting a breakpoint on a wait instruction would cause debugger stack frames to be left on the interrupt stack. This would eventually corrupt the interrupt stack. Breakpoints on wait instructions are allowed as long as the current stack is not the interrupt stack.

Can't modify breakpointed memory. Operation aborted.
The basic design of the debugger does not allow one to change the values of memory locations that contain breakpoints.

Can't set breakpoints on calls to non-64V routines.
The debugger only supports Prime V-mode. Because of the way that breakpoints are implemented on pcl instructions, the debugger must disallow any attempt to set breakpoints on pcl instructions which call a routine whose mode is other than V-mode.

Can't set breakpoints on calls to private segments. Operation aborted.
The debugger does not support breakpoints in private segments. Because of the way that breakpoints are implemented on pcl instructions, the debugger must also disallow any breakpoints on pcl instructions which call routines in private segments.

Command line is too long. Any input ignored.
The command line just entered exceeds the maximum length of 256 characters.

Current instruction doesn't allow steps.
The current step operation must abort since the next instruction does not allow breakpoints.

Current procedure's name not in Primos load maps.
As part of printing out the arguments to a procedure, an attempt was made to determine the name of the procedure by using the value of the lb register. However, the look up failed.

Define commands table is full.
The debugger internal table used to store user-defined commands is full. Some user-defined commands must be deleted before more can be added.

Given option is invalid with this command.
The current command does not recognize a given option as valid.

Given physical page does not exist for this system.
The physical page number given for the current command specifies a physical page that does not exist on the current system.

Given physical page is free or voided.
The physical page number given to the translate_to_virtual command does not correspond (at this moment) to any process's virtual address. In other words, no process currently owns this physical page.

Given structure name is not known.
The date base name specified in the structure command is not recognized by the command. A list of the valid names can be seen by issuing the structure command with no arguments.

Given virtual address corresponds to non-resident physical memory.
The translate_to_physical command has been issued with an address that is not currently resident. Therefore there is no corresponding physical address.

Given virtual address is undefined.
The segment specified by the virtual address does not exist (i.e. it has never been allocated).

Index registers cannot be used with this instruction.
Certain memory reference instructions do not allow indexing. This is a function of the Prime processor architecture.

Invalid access type specified.
The description of the access_type command specifies the valid access types.

Invalid addressing mode. Check indirection or indexing.
The operand given for a memory reference instruction has tokens in the fields that usually indicate indirection or indexing, but the tokens cannot be recognized.

Invalid base register expression given as address.
A base register expression is a specific type of address-expression. It specifies a base register plus or minus an octal offset. The debugger recognized a base register but the values following it do not seem to be valid offsets.

Invalid displacement in memory reference instruction.
The displacement given for a one word pc-relative form of a memory reference instruction exceeds the allowable limits (-223 to 255).

Invalid input for current access type.
An attempt was made to modify memory, but the new value is of the wrong access type (e.g. an ascii value for a decimal access type).

Invalid input type for this defined variable.
The data item being assigned to a program variable is not of the same type as the program variable.

Invalid interrupt process number specified.
Non-positive process numbers specify interrupt processes. However, the number specified for this command is beyond the range of those interrupt processes that are currently defined.

Invalid length specified for printing a field.
The length given for printing a field is not consistent with the type of a field (e.g. a decimal number that is 5 words long). A common way to get this error message is to add or modify a structure definition with an incorrect field definition.

Invalid mask. It must be a 16 bit octal number(s).
The mask or masks specified for the search command are not 16 bit octal numbers.

Invalid operand for a memory reference instruction.
The operand specified for the memory reference instruction cannot be parsed into a valid instruction.

Invalid operand for the instruction.
In attempting to parse an assembly language instruction, an operand was given that was invalid for the given instruction.

Invalid pcb link address encountered.
In attempting to sequence through the linked lists of pcb's that make up the system ready list, a pcb link address was found that is clearly invalid.

Invalid physical page number specified.
The physical page number argument is not a 16 bit octal number.

Invalid ring number in pointer.
A pointer was specified with an invalid ring number. The correct form for a pointer with a ring number is SSSS(R)/WWWWWWW where SSSS is a 12 bit segment number in octal, WWWWWWW is a 16 bit word number in octal, and R is the ring number. The ring number can only be 0 or 3.

Invalid status command option specified.
See the description of the status command for a list of the valid status command options.

Invalid symbolic address.
A symbolic address is a specific type of address-expression. It specifies a Primos load map symbol plus or minus an octal offset. The debugger interpreted the address-expression as a symbolic address but did not recognize the tokens as being in the valid format.

Invalid variable length.
The value being specified for the length of the program variable is not valid for the given type. Valid ranges for the lengths depend on the specified type.

Invalid variable name.
Program variable names must be like variable names in PL1G. They must be start with letters and consist of only letters, numbers, '_', and '$'.

Invalid variable type.
A type is given for a program variable that is unknown. See the description of the define_variable command for a list of the defined types.

Invalid word number in pointer.
A pointer was specified with an invalid word number. A 16 bit octal number is expected for the word number portion of the pointer.

Missing or invalid search pattern. A character string or 16 bit octal

number(s) is expected.
The search pattern given for the search command is not valid. See the search command description for more details.

Nesting of arrays in the definition of a structure is not supported.
The structure command works from internally defined tables in the debugger. These definitions allow a field to be defined as an array. However, arrays cannot be defined within arrays. See module ds>struc_dcl.pma.

No breakpoint exists at specified address.
An attempt was made to clear a breakpoint at a location were none was set.

Offsets from symbols must be 16 bit octal numbers.
All forms of address-expression's can have an optional offset. The given offset could not be converted into a 16 bit octal number.

Operation aborted due to faulted pointer.
In attempting to compute the effective address of an instruction, a faulted pointer (linkage fault) was encountered. This happens most often with either breakpoints or steps on pcl instructions. The debugger cannot resolve the address so it must abort.

Procedure name is required for variables defined in terms of base register expressions.
If a program variable is defined in terms of a base register, a check is made in the debugger to ensure that any references to this variable are made while executing in the procedure for which the variable was defined. The procedure name is needed for this check.

Proceed count must be between 1 and 32767.
The proceed count attribute given in the breakpoint command was not in the proper range of 1 to 32767.

Process numbers must be decimal numbers less than 256.
The process number specified is greater than the number of processes that Primos can support. Currently the maximum number supported is 255.

Process specified does not have terminal buffers.
The ttybuf command was issued with a process number that does not have terminal buffers. Processes such as slaves and phantoms do not have terminal buffers.

Referenced variable is not defined for current procedure.
A program variable has been referenced which was defined with a procedure name that does not match the procedure currently being executed by the active process.

Search pattern size must not be greater than size of the memory to search.
The pattern specified with the search command is larger than the area of memory that is to be searched.

Segment numbers must be 12 bit octal numbers.
A 12 bit octal number representing a segment number was expected as an argument by the current command.

Shift counts must be between 0 and 77 in octal.
The operand to a shift instruction was not in the valid range or radix.

Specified memory reference operand requires a 2 word instruction.
The is no one word form of this memory reference instruction given the specified operand.

Specified process is logged out.

The process number specified references a process which is logged out. This is a problem for the current command since there is limited information about a logged-out process.

Specified process is not configured.
The process number specified is greater than the number of processes configured on this system.

Specified register name is unknown.
The register name given does not match any known to the debugger. See the access_register command for a list of valid names.

Step presumes entry from a breakpoint/single step.
The ability to issue the step command is limited to the situation where the debugger must have been entered from a previously set breakpoint or single step. (See section 5.1.4).

Stepping through a critical region is not allowed.
In single stepping through code, an inhibit instruction was encountered. Setting breakpoints in interrupt inhibited code breaks the critical region and thus the step is aborted. (See section 5.1.1)

Structure definition is missing an array begin.
The structure command works from internally defined tables in the debugger. These tables have been incorectly modified to indicate the end of an array without a beginning. See the module containing the structure definitions for a complete description of modifying these data bases. (ds>struc_dcl.pma)

Structure definition is missing an array end.
The structure command works from internally defined tables in the debugger. These tables have been incorectly modified to indicate the start of an array without an end. See the module containing the structure definitions for a complete description of modifying these data bases. (ds>struc_dcl.pma)

Structure entry has an invalid type.
The structure command works from internally defined tables in the debugger. Each entry specifies a type and length for a particular field. An entry has just been encountered that does not have a defined type. See the module containing the structure definitions for a list of the valid types. (ds>struc_dcl.pma)

Symbol not found in Primos load maps.
An unsuccessful attempt was made to look up a symbol name or address in the Primos load maps.

The active process does not own a register set yet the specified register only exists in user register sets.
Certain registers, such as fcode and faddr, only exist for a process when that process owns a register set. When the process gets swapped out of a register set, values for these registers are not saved and thus are undefined.

The ending address is not a valid address expression.
The command just issued requires a valid address-expression as an argument. This argument represents the ending address. See the definition of address-expression.

The size limit on the search pattern is 8 octal numbers or a string of 20 characters.
The search pattern given for the search command is not valid. See the search command description for more details.

The specified name has not been defined as a command.
An attempt was made to either display or delete a user-defined command that had not

previously been defined.

The starting address is not a valid address expression.
The command just issued requires a valid address-expression as an argument. This argument represents the starting address. See the definition of address-expression.

The step count must be a number between 1 and 32767.
The value given for the step count was either not a valid number or not in the allowed range.

There is already an actively stepping process.
The design of the debugger only allows one process system-wide to be actively stepping at any one time. (See section 5.1.4).

Too many arguments specified.
The given command uses fewer arguments than were given on the command line.

Too many masks specified.
The mask specified for the search command is not the same length as the specified pattern to search for. The mask and the pattern must be the same number of words.

Unknown command.
The command typed is neither a valid debugger command nor a user-defined command.

Unknown V-mode instruction.
The current command required a look up of an instruction by either opcode or mnemonic, but no corresponding value was found in the internal debugger instruction set table for V-mode instructions.

Variable is not defined.
A reference has been made to a program variable which has not been previously defined.

Variable table is full.
The internal debugger table used to store information about program variables is full. Some variables must be deleted before more can be added.

Virtual address must be specified in octal.
The argument typed is not a valid virtual address. Valid input must be an octal segment number and then an octal word number with a / separating the 2 fields.

Appendix E
Summary of Functionality Limitations

Due to the complexity of Prime processors and Primos, it is not possible to develop a debugger without some caveats or limitations on its functionality. This appendix summarizes some of these limitations. Complete descriptions of the limitations are given in chapter 5.

1. Breakpoints cannot be set on the following instructions: ARGT, CALF, E16S, E32I, E32R, E32S, E64R, E64V, IRTC, IRTN, LPSW, STEX, and SVC. The quad floating point instructions are also not supported.

2. Breakpoints will not work if placed in the following places: check handling code, phantom interrupt code, the tape dump program, coldstart and warmstart code before process exchange is turned on, code executed by the frontstop process, illegal instruction fault handlers, Ring Zero Debugger code.

3. Breakpoints should not be set in interrupt inhibited code.

4. Memory can't be modified where breakpoints are set.

5. Breakpoints won't work on self-modifying code.

6. Breakpoints make it more likely that a stack might overflow by pushing on extra stack frames.

7. A corrupt stack can make the debugger fail.

8. Breakpoints in a few places in the ring zero fault handlers can sometimes cause deadly embrace.

9. Breakpoints on pcl instructions won't be seen if the instruction fails to complete.

10. The step command can only be issued if the debugger was entered by encountering a previously set breakpoint.

11. The step command cannot be issued again if a previous step command has not completed.

12. The step command will abort for various reasons: an inhibit interrupts instruction has been encountered, the process is about to enter a private segment, the code is attempting to change modes, a breakpoint cannot be set on the next instruction.

13. Non-resident memory cannot be examined.

14. Corrupting a process exchange data base may make the debugger fail.

15. The debugger cannot be restarted without running Primos.

16. The VCP commands display and displayc do not work while in the debugger.

17. Warmstarts on breakpointed processes which use the interrupt stack may fail.

18. Warmstarts while in the debugger do not work at all on the P850.

19. Modifying Primos routines that are also separately loaded with the debugger can make the debugger fail.

Appendix F
Maintenance Notes

Like any other piece of software, the debugger is expected to evolve over time. In many cases, these changes may be made by engineers who are not very familiar with the structure of the debugger. The purpose of this appendix is to pass along some information that may be useful to those people who need to know more about the structure of the debugger. If more information is desired, a complete design specification was written and can be found in osdoc (the document system for the Primos group).

F.1 Changes to Primos for the Ring Zero Debugger

Most of the Ring Zero Debugger consists of newly written code. This new code consists of about 100 modules and resides in a new Primos subdirectory know as ds (i.e. debugger source). This code is for the most part, independent of the rest of Primos.

The actual number of existing Primos modules which were changed for the debugger is very small. The number is 8 , excluding minor insert file changes, and half of these changes consist of only a few modified lines. The basic changes are listed below.

1. A new debugger process
2. A modified system console driver to detect the debugger key sequence and invoke it -if it is configured
3. Modifications to the illegal instruction fault handlers to invoke the debugger if it is configured (basis of breakpoints)
4. A modified frontstop process to loop while the debugger is running (P850 only)
5. The addition of a new private gate so that breakpoints can be supported in ring 3

One other area where the debugger will be noticed is the ring zero load. The load file load>begin_load is very much larger because this is where the debugger was added to the load sequence. The final phase of the ring zero load is also different. Just before the mapgen program is invoked, a new program (dump_maps) will be invoked. The purpose of this program is to write the newly created Primos load maps into one of the Primos run images (i.e. PRXXXX files) for later use by the debugger.

F.2 Areas Most Likely to Change

There are two major reasons for changing the debugger. One reason is to enhance the existing functionality. An example of such an enhancement is to allow conditional breakpoints. The other reason for changing the debugger is if Primos changes. This is true because the debugger knows about, and makes certain assumptions about, the structure of a few Primos data bases. This latter reason for changing the debugger will be the topic of this section.

The basic operation of the debugger (e.g. breakpoints, accessing memory and registers), has very little connection with Primos. This code depends more on the processor architecture than on Primos. In other words, it will be a relatively rare occasion that a change to Primos will require a change in these routines. What is more likely to require changes are certain commands that are rather closely related to the current structure of Primos. These commands include structure, status, ready_list, print_locks, and ttybuf.

The structure command will undoubtedly be the major reason for changing the debugger. The price of getting the desireable functionality of this command is that there is a debugger data base that can very easily become out-of-date. When this happens, the structure command will fail to work properly. It will be the job of every engineer to update the debugger structure data base when a change is made to a Primos data base that the debugger knows about.

One more reason for modifying the structure command will be to add new definitions for new or existing Primos data bases. The original set of structure definitions was just to demonstrate the capability. The hope is that as the use of the debugger becomes more widespread, engineers will populate this debugger data base.

Another likely command that will require modifications is the status command. It returns a fair amount of information on each process, much of which can change over time. Some examples of things that can change in Primos that will require a corresponding change in this command are new interrupt processes, new ready list levels, new user types, new n1locks, and new abort flags.

Other commands that may easily require changes are print_locks and ttybuf. If any n1locks in the system are added or modified, then print_locks must reflect the change. The ttybuf is able to display the contents of terminal buffers by knowing the data base for these buffers. Any changes in the format of these buffer data bases must be reflected in the ttybuf command of the debugger.

One situation where one needn't change the debugger is if new interrupt processes are being added at an existing ready list level. This is somewhat surprising since the Autopsy program requires modification in this circumstance and the debugger does not. The difference is due to the fact that the names of interrupt processes are looked up in the load maps by the debugger.

In summary, the following changes to Primos will require changes to debugger code. The commands or modules affected by the changes are shown in parenthesis.

- o modifications to Primos data bases that have a debugger structure defined for them (structure)
- o changing or adding new user types (status)
- o ready list level changes (ready_list, status)
- o changing or adding new nllocks (print_locks, status)
- o changing or adding new process aborts (status)
- o changing terminal buffer data bases (ttybuf)
- o elimination of the hideous refalt mechanism in Primos (ds>refalt_fix.pma)

F.3 Getting a Load Map

Getting loader information about the Ring Zero Debugger is unfortunately somewhat involved. Because the debugger is loaded first and then all symbols are expunged, there is virtually no information about the debugger in the Primos ring zero map. There is a way to get a separate map of the debugger, but because of limitations in the seg loader, the standard load is not set up to generate one.

The problem with seg is that it cannot generate two load maps in the same load sequence. The way to get around this is to issue the seg subcommand "return" to leave the load sequence and then restart it with the subcommand "load *". This allows one to get a map of just the debugger, separate from the Primos ring zero map. However the bad consequence of doing this is that when seg completes, it will indicate an error. This is at best confusing to people, at worst it will prevent cold.cpl program from running. For this reason, a map is not automatically generated.

In the actual load file for the start of Primos ring zero load, the seg commands to generate a map for the debugger are included but are commented out. To get a debugger load map these lines should be replaced with the real commands. It may also be necessary to manually run cold.cpl.

F.4 Reporting Errors

For problem reporting purposes, the Ring Zero Debugger is just a particular part of Primos. Therefore, if there are any problems with the debugger, the appropriate means of resolving the problem is to enter a spar into Polaris just like any other Primos problem. On this spar, the product will be Primos, the subproduct will be debugger.

Index

Carl P. Underwood is attempting to emulate a CPU.
- He works 24 hours a day, 7 days a week, 52 weeks a year.
- He can do in 1 second what a Prime 9950 can do in 1 nano-second.

Here is a comparison which will show how Carl is doing.

TASK	PRIME 9950	CARL
I Execution of a generic instruction:		
- STLB & cache hit	90 ns	1 min 30 sec
- STLB hit, cache miss	900 ns	15 min
- STLB miss, cache hit	2120 ns	35 min
- STLB miss, cache miss	3000 ns	50 min
- Page fault (disk I/O)	30 - 100 ms	1 - 3.1 years
II Interrupts:		
- Clock process (250 hertz)	4 ms	46 days
- AMLC clock (110 baud)	100 ms	3.2 years
- AMLC clock (9600 baud)	1.041 ms	12 days
- Minor time slice	300 ms	9.5 years
- Major time slice	2 sec	63.5 years
- DMx trap (DMA - DMQ)	900 - 7000 ns	15 min - 2 hours
- Burst mode DMA transfer	4 words 900 ns every 6900 ns for 1.8 ms	4 words 15 mins every 2 hrs for 21 days
III Wait Times:		
- Process Exchange	1.9 - 7.6 us	32 mins - 126 mins
- Average disk seek	9.0 - 25.0 ms	3.5 - 10 months
- Random disk seek	45.0 ms	1.5 years
- Average latency time	8 ms	2.5 months
IV Misl.		
- 1 second of wall clock	1 sec	31.7 years
- Rewind tape	1 min	1902 years
- Coffee break	15 min	28,530 years

Ring Zero Debugger Examples (10/10/85)

Demonstration 1

```
CP>

CP> sysclr

••• CPU VERIFIED •••

CP> boot 74114
```

Configure the debugger and enter it during coldstart

```
Debugger entered due to coldstart request.
-> status
```

status command.

```
Process 1  SYSTEM
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    14(0)/4667 (SUPPB)
   LB:    0(0)/0 (Unknown)
   SB:    6003(0)/164       XB:    0(0)/0
   L:     000000 000000     E:     000000 000000
   X:     000000            Y:     000000
   FAR0:  000000 000000     FLR0:  000000 000000
   FAR1:  000000 000000     FLR1:  000000 000000
   Keys:  014000
```

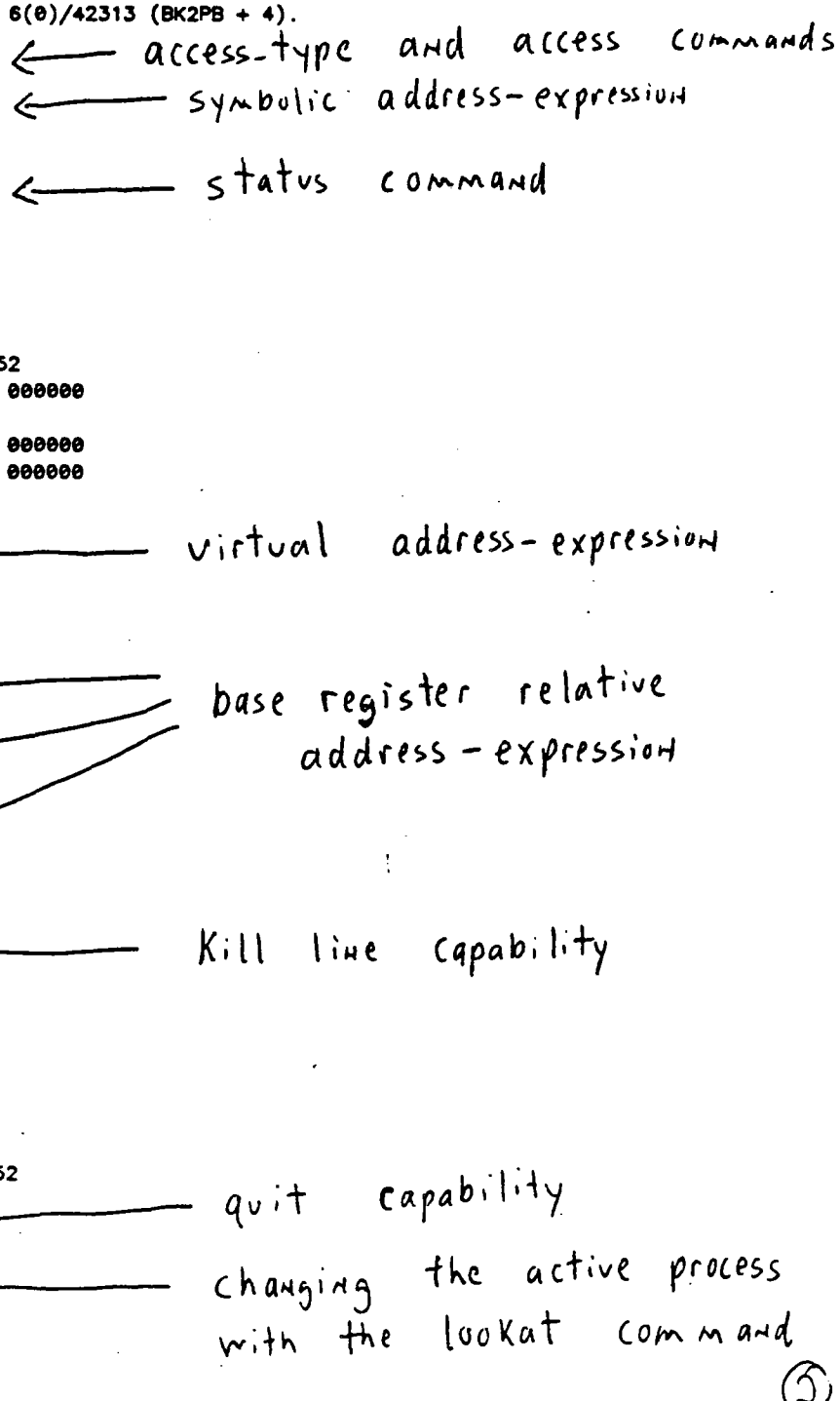
first instruction executed by user 1

```
-> continue
Leaving the debugger.
CONFIG -DATA CONFIG
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> continue
Leaving the debugger.
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> continue
Leaving the debugger.

PRIMOS 20.1.0OS

(c) Prime Computer, Inc., 1985

2048K BYTES MEMORY IN USE

Starting up revision 19 partition "DOLFIN".
(Quota system may be incorrect; run FIX_DISK.)
\
Debugger entered due to console interrupt.
     Process 1 was executing at 6(0)/46576 (PGRESI + 4).
-> continue
Leaving the debugger.
418K bytes wired.
Please enter date and time.
OK, COMO -NTTY
OK, START_NET NETCON.EN.D11 -NODE EN.D11
Beginning Network Initialization.
OK, ADD -ON ENS OSGRP4 SYSENS INTEG OSGRP2 NEWENS
OK, ADD -ON ENM SYSENM OSGRP1 SOFTM DUMPM DSAG
OK, ADD -ON S35 OSGRP0 OSGRP3
OK, event_log -net -off
OK, RDY -LONG
OK 00:02:42  47.566  94.230
/* Enter time and type CO -CONTINUE.
```

Console interrupts

Much more wired memory for the debugger

```
OK 00:02:43  0.087  0.000
CO -PAUSE
OK 00:02:43  0.066  0.093
se -100385 -1502
OK 15:02:04  0.330  0.727
co -continue
OK 15:02:11  0.054  0.000
MAX ALL
OK 15:02:11  0.100  0.000
COMO -NTTY
OK 15:04:11  0.060  0.057
CO -END
OK 15:04:11  0.054  0.000
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> access_type
Current access type is symbolic.
-> access prwf$$
11(0)/33046  ARGT
11(0)/33047  LDA% SB%+ 70 ,*
11(0)/33051  STA# SB%+ 34   ?
-> status
Process -20  BK1PCB
   Level: Backstop process
   State: Ready
   PB:    6(0)/42313 (BK2PB + 4)
   LB:    6(0)/42110 (SCHED)
   SB:    4(0)/163750 *      XB:     4(0)/552
   L:     000000 000000      E:      000000 000000
   X:     000400             Y:      000000
   FAR0:  000000 000000      FLR0:   000000 000000
   FAR1:  000000 000000      FLR1:   000000 000000
   Keys:  034101

-> access 11/33046
11(0)/33046  ARGT
11(0)/33047  LDA% SB%+ 70 ,*
11(0)/33051  STA# SB%+ 34   ?
-> access_type octal
-> access lb%+402
6(0)/42512  006000
6(0)/42513  000130  ?
-> access sb%+2
4(0)/163752  000004
4(0)/163753  163747  ?
-> access_type symbolic
-> access *
6(0)/42313  BDX 42313
6(0)/42315  IRS%  42466 ,Y  ?
-> abcdefghijkl?access_type
Current access type is symbolic.
-> status
Process -20  BK1PCB
   Level: Backstop process
   State: Ready
   PB:    6(0)/42313 (BK2PB + 4)
   LB:    6(0)/42110 (SCHED)
   SB:    4(0)/163750        XB:     4(0)/552
   L:     000000 00
quit.
   lookat
  tive process is -20.
   lookat 1
-> status
Process 1  SYSTEM
```



```
   Level: System process
   Type:  Supervisor
   State: Waiting at 6(0)/13350 (ASRSEM)
   PB:    6(0)/34235 (WAITA + 74)
   LB:    6(0)/55706 (C1IN$)
   SB:    6003(0)/164        XB:     4(0)/100100
   L:     000000 000000      E:      000000 000000
   X:     000000             Y:      000000
   FAR0:  000000 000000      FLR0:   000000 000000
   FAR1:  000000 000000      FLR1:   000000 000000
   Keys:  014001

-> access_type octal
-> dump vpdev vpdev+15

6(0)/23757  004060 000001 000001 000001 000001 000001 000001 000001
6(0)/23767  000001 000001 000001 000001 000001 100000
-> breakpoint pagtur+1
-> list pagtur+1
Type   Address         Procedure          Process  Count  Mnemonic
brkpt  6(0)/45205      PAGTUR + 1           Any        1    CRA
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 29 was executing at 6(0)/45205 (PAGTUR + 1).
-> status
Process 29  (Login name is not resident)    *** Owns register set 1 ***
   Level: Network process
   Type:  Network process
   State: Ready
   PB:    6(0)/45205 (PAGTUR + 1)
   LB:    6(0)/46002 (PAGTUR)
   SB:    6000(0)/1264       XB:     30(0)/61460
   L:     000010 100077      E:      000000 000000
   X:     000000             Y:      177777
   FAR0:  000000 000000      FLR0:   000000 000000
   FAR1:  031070 000000      FLR1:   000000 000000
   Keys:  034100             Modals: 100077
   Fcode: 045206 000040      Faddr:  6(0)/1703
   Locks owned: NETLCK

-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 29 was executing at 6(0)/45205 (PAGTUR + 1).
-> list *
Type   Address         Procedure          Process  Count  Mnemonic
brkpt  6(0)/45205      PAGTUR + 1           Any        1    CRA
-> clear pagtur+1
-> list *

*** Debugger user error:
    No breakpoint exists at specified address.
-> foobar

*** Debugger user error:
    Unknown command.
-> access 7777/0
29:7777(0)/0
  * Fault while in debugger:
    Segment fault (type 60) encountered at 55(0)/15227.
    Attempt to reference 7777(0)/0.
-> access nlogin
15(0)/2267
```

← waiting for character input

← dump command

← breakpoint and list commands

← location of breakpoint

← NI Lock held

← User errors

← segment fault in debugger

```
*** Fault while in debugger:
    Page fault (type 10) encountered at 55(0)/15227.
    Attempt to reference 15(0)/2267.
-> continue
Leaving the debugger.
date
   Oct 85 15:07:56 Thursday
^ 15:07:59  0.166  0.157
```

← page fault in debugger

Demonstration 2

```
date
03 Oct 85 15:09:08 Thursday
OK 15:09:11  0.136  0.000
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> access_type symbolic
-> access pagtur+1
6(0)/45205  CRA   ?
-> breakpoint pagtur+1
-> continue
Leaving the debugger.
avail

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 6(0)/45205 (PAGTUR + 1).
-> access_register a
A: 000010
-> status
Process 1  SYSTEM                          *** Owns register set 1 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    6(0)/45205 (PAGTUR + 1)
   LB:    6(0)/46002 (PAGTUR)
   SB:    6000(0)/1264        XB:     6002(3)/3073
   L:     000010 100077       E:      000000 000000
   X:     000000              Y:      177777
   FAR0:  066002 002754       FLR0:   000004 000000
   FAR1:  066002 002754       FLR1:   000016 000000
   Keys:  034100              Modals: 100077
   Fcode: 045206 000040       Faddr:  6(0)/1703

-> access_register a
A: 000010  12345
-> access_register a
A: 012345
-> access_register lb
LB (high order): 000006
LB (low order): 046002
-> access_register fcode
FCODE (high order): 045206
FCODE (low order): 000040
-> search 55/0 55/177777 '%11:'
55(0)/14203  %11:2

*** Fault while in debugger:
    Page fault (type 10) encountered at 55(0)/120507.
    Attempt to reference 55(0)/174000.
-> access_type ascii
-> access 55/14203
55(0)/14203  %1
55(0)/14204  1:
55(0)/14205  2z  ?
-> access_type octal
-> dump 600/2600 600/2677

600(0)/2600  140000 140001 000077 140003 140004 141023 140275 000000
600(0)/2610  000000 000000 000000 000000 000000 000000 000000 000000
600(0)/2620  000000 000000 000000 000000 000000 140272 000000 000000
  9(0)/2630  000000 140750 140751 000000 000000 000000 000000 000000
 .0(0)/2640  000000 000000 000000 000000 000100 000100 000100 000100
600(0)/2650  000101 000101 000101 000101 000101 000101 003777 003777
600(0)/2660  003777 003777 003777 003777 003777 003777 003777 003777
```

access_register command

"a" register is modified

search command (for a string)

```
600(0)/2670  003777 003777 003777 003777 003777 003777 003777 023777
-> search 600/2600 600/2677 140001 & 140001
600(0)/2601  140001
600(0)/2603  140003
600(0)/2605  141023
600(0)/2606  140275
600(0)/2632  140751
-> listall
Type   Address        Procedure          Process  Count  Mnemonic
brkpt  6(0)/45205     PAGTUR + 1           Any        1    CRA
-> clearall
-> listall
No breakpoints are set.
-> continue
Leaving the debugger.
     Volume DOLFIN
     118512 total records
      11296 records available
      90.5% full

OK 15:09:31  1.036  2.975
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> breakpoint p$cidx+1
-> continue
Leaving the debugger.
date

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(3)/133503 (P$CIDX + 1).
-> access_type symbolic
-> access p$cidx
 1(0)/133502  ARGT
 1(0)/133503  LDA# 133616
 (0)/133504  STA# SB%+ 31
41(0)/133505  EAFA 0, SB%+ 12 ,*
41(0)/133510  LDA# SB%+ 26
41(0)/133511  BLGE 133515   ?
-> clearall
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(3)/133504 (P$CIDX + 2).
-> step 2

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(3)/133510 (P$CIDX + 6).
-> step 1000

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(3)/1134 (HASH_UID + 134).
-> step 2000

\
Debugger entered due to console interrupt.
     Process 1 was executing at 55(0)/615 (DBGSRT + 74).
-> listall
Type   Address        Procedure          Process  Count  Mnemonic
step   32(0)/7733     CL$PIX + 53            1     1659   LDA
-> continue
  aving the debugger.

  ugger entered due to breakpoint/single step.
     Process 1 was executing at 32(3)/10432 (CL$PIX + 552).
-> access_type octal
```

search for a pattern using a mask

listall command

clearall command

step command

can enter debugger with a console interrupt while single stepping

```
-> access pfcn
6(0)/622  000000
6(0)/623  007041  ?
-> breakpoint 1:pagtur+467 5
-> listall
Type   Address          Procedure          Process  Count  Mnemonic
brkpt  6(0)/45673       PAGTUR + 467           1      5     STL
-> continue
Leaving the debugger:
03 Oct 85 15:14:52 Thursday
OK 15:14:53  2.130  0.039
avail
     Volume DOLFIN
     118512 total records
      11296 records available
      90.5% full

OK 15:15:04  0.578  0.215
ld x

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 6(0)/45673 (PAGTUR + 467).
-> access pfcn
6(0)/622  000000
6(0)/623  007046  ?
-> clearall
-> continue
Leaving the debugger.

<DOLFIN>CMDNC0 (ALL access)
3965 records in this directory, 3965 total records out of quota of 0.

No entries selected.

OK 15:15:17  2.721  0.954
```

← page fault counter

← using a proceed count with the breakpoint command

=

← 5 page faults since last breakpoint.

Demonstration 3

```
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> brk gpath$+10
-> continue
Leaving the debugger.
!d

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62044 (GPATH$ + 10).
-> arguments
Current routine: GPATH$

6 arguments at SB%+56:
# 1 at 41(3)/124207  : 000001.140040
# 2 at 6002(3)/4715  : 000037.000001
# 3 at 6002(3)/10433 : 006742.066002
# 4 at 41(3)/124530  : 000200.140040
# 5 at 6002(3)/6523  : 035045.000010
# 6 at 6002(3)/4141  : 000000.000000
-> pcb 1

Process:   1
Level:  622                      Link: 000000
Wait list: 0(0)/106717           Abort flags: 0000000000000000
PB: 6(0)/35703                   SB: 6000(0)/1456
LB: 6(0)/41334                   XB: 6(0)/106716
L: 000000 000000                 E: 000000 000000
X: 000000                        Y: 000000
FAR0: 000000 000000              FLR0: 000000 000000
FAR1: 000000 000000              FLR1: 000000 000000
Interval timer: 177774 000000    Elapsed timer: 000002 012761
DTAR2: 140002 167101             DTAR3: 176302 167064
Keys: 014001

Concealed stacks:
            PB                KEYS          FCODE(high)          FADDR
            11(0)/62044       014100        062045               11(0)/1703
            13(3)/55722       014000        000000               13(3)/60760
-> status

Process 1  SYSTEM                          *** Owns register set 0 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/62044 (GPATH$ + 10)
   LB:    11(0)/62432 (GPATH$)
   SB:    6003(0)/164        XB:     6(0)/27052
   L:     000000 062044      E:      017777 017777
   X:     000000             Y:      177777
   FAR0:  066002 004412      FLR0:   000000 000000
   FAR1:  066002 006435      FLR1:   000000 000000
   Keys:  014100             Modals: 100037
   Fcode: 062045 000040      Faddr:  11(0)/1703
   Locks owned: FSLOK

-> status all
Process -21  DBGPCB                        *** Owns register set 1 ***
   Level: Debugger process
   State: Ready
   PB:    55(0)/0 (DBGSEG)
   LB:    55(0)/11552 (Unknown)
   PCB abort flags: TSEALM

Process -20  BK1PCB
```

arguments command

pcb command

status command

```
   Level: Backstop process
   State: Ready
   PB:    6(0)/42313 (BK2PB + 4)
   LB:    6(0)/42110 (SCHED)

Process -19  BK2PCB
   Level: Backstop process
   State: Ready
   PB:    6(0)/42307 (BK2PB)
   LB:    6(0)/42110 (SCHED)

Process -18  CLKPCB
   Level: Clock process
   State: Waiting at 4(0)/500 (SEMCOM)
   PB:    6(0)/4043 (CLKINS + 650)
   LB:    6(0)/5366 (UNLOAD (et al))

Process -17  FNTPCB
   Level: Clock process
   State: Waiting at 4(0)/572 (FNTSEM)
   PB:    6(0)/4057 (FNTPB)
   LB:    6(0)/5366 (UNLOAD (et al))

Process -16  SLCPCB
   Level: SMLC process
   State: Waiting at 4(0)/502 (SLCSEM)
   PB:    12(0)/1005 (SLCPB)
   LB:    12(0)/4020 (SLCINI (et al))

Process -15  AMLPCB
   Level: AMLC process
   State: Waiting at 4(0)/504 (AMLSEM)
   PB:    16(0)/4427 (AMLCI_ + 141)
   LB:    16(0)/10200 (Unknown)

Process -14  MPCPCB
   Level: MPC process
   State: Waiting at 4(0)/506 (MPCSEM)
   PB:    6(0)/130002 (MPCDIM)
   LB:    6(0)/130732 (MPINIT (et al))

Process -13  MP2PCB
   Level: MPC process
   State: Waiting at 4(0)/510 (MP2SEM)
   PB:    6(0)/131376 (MP2DIM)
   LB:    6(0)/132326 (M2INIT (et al))

Process -12  GP1PCB
   Level: MPC4/Versatec process
   State: Waiting at 4(0)/512 (GP1SEM)
   PB:    11(0)/20623 (GP1PB)
   LB:    11(0)/23633 (Unknown)

Process -11  GP2PCB
   Level: MPC4/Versatec process
   State: Waiting at 4(0)/514 (GP2SEM)
   PB:    11(0)/20623 (GP1PB)
   LB:    11(0)/23634 (Unknown)

Process -10  VERPCB
   Level: MPC4/Versatec process
   State: Waiting at 4(0)/516 (VERSEM)
   PB:    6(0)/135352 (VERDIM)
   LB:    6(0)/135720 (GTSTAT (et al))

Process -9  PNCPCB
```

```
   Level: Disk/Ringnet process
   State: Waiting at 4(0)/520 (PNCSEM)
   PB:    12(0)/25314 (PNCDIM + 6)
   LB:    12(0)/30366 (PNCNIT (et al))

Process -8  SP1PCB
   Level: Disk/Ringnet process
   State: Waiting at 4(0)/522 (SP1SEM)
   PB:    4(0)/163747 (INTRT_)
   LB:    4(0)/1532 (Unknown)

Process -7  DK1PCB
   Level: Disk/Ringnet process
   State: Waiting at 4(0)/534 (DSKSEM)
   PB:    6(0)/37537 (DMA_ERR + 224)
   LB:    6(0)/106316 (Unknown)

Process -6  DK2PCB
   Level: Disk/Ringnet process
   State: Waiting at 4(0)/536 (DSKSEM + 2)
   PB:    6(0)/37537 (DMA_ERR + 224)
   LB:    0(0)/1335 (Unknown)

Process -5  DK3PCB
   Level: Disk/Ringnet process
   State: Waiting at 4(0)/540 (DSKSEM + 4)
   PB:    6(0)/37537 (DMA_ERR + 224)
   LB:    0(0)/1472 (Unknown)

Process -4  DK4PCB
   Level: Disk/Ringnet process
   State: Waiting at 4(0)/542 (DSKSEM + 6)
   PB:    6(0)/37537 (DMA_ERR + 224)
   LB:    0(0)/1627 (Unknown)

Process -3  ASYPCB
   Level: AMLC process
   State: Waiting at 4(0)/3335 (ASYSEM)
   PB:    16(0)/11412 (ASYNDM)
   LB:    16(0)/12734 (Unknown)

Process -2  SLXPCB
   Level: SMLC process
   State: Waiting at 4(0)/3337 (SLXSEM)
   PB:    12(0)/4446 (SLXPB)
   LB:    12(0)/5444 (Unknown)

Process -1  IPQPCB
   Level: IPQ process
   State: Waiting at 4(0)/3341 (IPQSEM)
   PB:    16(0)/32314 (IPQPB)
   LB:    16(0)/31734 (Unknown)

Process 0  IBSPCB
   Level: IPQ process
   State: Waiting at 4(0)/3343 (IBSSEM)
   PB:    16(0)/31330 (IBSPB)
   LB:    16(0)/30742 (Unknown)

Process 1  SYSTEM                          *** Owns register set 0 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/62044 (GPATH$ + 10)
   LB:    11(0)/62432 (GPATH$)
   Locks owned: FSLOK
```

```
Process 29  NETMAN
   Level: Network process
   Type:  Network process
   State: Waiting at 12(0)/25302 (PNTSEM)
   PB:    6(0)/34454 (WAIT + 4)
   LB:    6(0)/34156 (SETSWI (et al))

-> trace 1

Level 1:  GPATH$
Root: 6003  SB: 6003/164  Size: 174 words  Type: 000000 (PCL)
Keys: 034200
Call at 41(3)/124254 (EPF$MAP (et al) + 372) SB: 6002(3)/6436 LB: 41(0)/125032

(trace)> arguments
Current routine: GPATH$

6 arguments at SB%+56:
# 1 at 41(3)/124207  : 000001.140040
# 2 at 6002(3)/4715  : 000037.000001
# 3 at 6002(3)/10433 : 006742.066002
# 4 at 41(3)/124530  : 000200.140040
# 5 at 6002(3)/6523  : 035045.000010
# 6 at 6002(3)/4141  : 000000.000000

(trace)> father

Level 2:  EPF$MAP
Root: 6002  SB: 6002/6436  Size: 1090 words  Type: 000000 (PCL)
Keys: 034100
Call at 13(3)/15071 (STD$CP + 2313) SB: 6002(3)/4622 LB: 13(0)/20564

(trace)> arguments
Current routine: EPF$MAP

4 arguments at SB%+111:
# 1 at 13(3)/20721   : 000000.177240
# 2 at 6002(3)/4715  : 000037.000001
# 3 at 13(3)/15223   : 000006.020000
# 4 at 6002(3)/4141  : 000000.000000

(trace)> father

Level 3:  STD$CP
Root: 6002  SB: 6002/4622  Size: 908 words  Type: 010000 (PCL)
Keys: 134200
Call at 13(3)/7364 (LISTN_ (et al) + 1042) SB: 6002(3)/4046 LB: 13(0)/11364

(trace)> arguments
Current routine: STD$CP

6 arguments at SB%+115:
# 1 at 6002(3)/4410  : 000002.166344
# 2 at 6002(3)/4141  : 000000.000000
# 3 at 6002(3)/4142  : 000000.066002
# 4 at 6002(3)/4150  : 000000.000000

(trace)> stack

(SB: 6002/4622)
1:6002/>10  134200    7366    7777       0   40000       0       0       3
  6002/>20   47053       4  177777   66002   60013   15035   66002    6303
1:6002/>30   60013   21645   60013   21213   66002    6410   60013   21645
1:6002/>40    7777       0   66002    6410   66002    6422     212     210
1:6002/>50  100000       3       1       0       0       0  140000       0
```

trace commands and subcommands

(11)

```
1:6002/>60   11345      1   66002      0      0      2   4411  100003
1:6002/>70    4622      3       4     37      1      1  14000   41732
1:6002/>100      1      0       0      0 140000      0  60013   46204
1:6002/>110   7777     10  177400   4652      1  66002   4410       0
1:6002/>120  66002   4141       1  66002   4142      0  66002    4150
1:6002/>130      0 100000    4650      1 100000  66002   4666   66002
1:6002/>140  60013  46204   66002  66002   4664  60013  66002    4634
1:6002/>150 100000  66002    4650 100000     41  52674 100000   53162
1:6002/>160  66002 100000   66002  66002 100000  66002  66002  100000
1:6002/>170  60041  44055  100000     10 177400 100000  66002    4761
1:6002/>200 100000  64377   75266 100000  64377 101264 100000  100000
1:6002/>210  66002 100000       0    120 100000   4525      0  100000
1:6002/>220  60041 130356  100000   4672     41 100000  66002     403
1:6002/>230 100000 100000       0 100000  60041  43747 100000    4770
1:6002/>240     41 100000   14000  43737 100000 130460      1  100000
1:6002/>250   4667  66002  100000  20556  64377 100000      2   66002
1:6002/>260 100000      2       0 100000   7777      0 100000   44055
--More--n

(trace)> father 2

Level 5:  COMLV$
Root: 6002  SB: 6002/4022  Size: 20 words  Type: 000000 (PCL)
Keys: 134100
Call at 13(3)/113353 (DF_UNIT_ + 6403) SB: 6002(3)/2136 LB: 13(0)/116524

(trace)> son

Level 4:  LISTN_
Root: 6002  SB: 6002/4046  Size: 364 words  Type: 010000 (PCL)
Keys: 014000
Call at 13(3)/137230 (COMLV$ (et al) + 0) SB: 6002(3)/4022 LB: 13(0)/136642

(trace)> father 2

Level 6:  DF_UNIT_
Root: 6002  SB: 6002/2136  Size: 948 words  Type: 000000 (PCL)
Keys: 014000
Call at 13(3)/100577 (RAISE_ + 437) SB: 6002(3)/2036 LB: 13(0)/100300

(trace)> arguments
Current routine: DF_UNIT_

1 arguments at SB%+122:
# 1 at 6002(3)/2116  : 066002.001704

(trace)> father

Level 7:  RAISE_
Root: 6002  SB: 6002/2036  Size: 64 words  Type: 000000 (PCL)
Keys: 014000
Call at 13(3)/77633 (SIGNL$ + 261) SB: 6002(3)/1704 LB: 13(0)/77440

(trace)> father

Level 8:  SIGNL$
Root: 6002  SB: 6002/1704  Size: 90 words  Type: 040000 (PCL)
Keys: 014000
Call at 13(3)/132307 (SWFIM_ (et al) + 5) SB: 6002(3)/1524 LB: 13(0)/131732

(trace)> father

Level 9:  C1IN$
Root: 6002  SB: 6002/1524  Size: 112 words  Type: 000000 (PCL)
Keys: 014000
Call at 13(3)/46146 (CL$GET + 266) SB: 6002(3)/1412 LB: 13(0)/45604.
```

(12)

```
(trace)> goto 15

*** End of stack at level 12 (start + 3)

Level 12:  INFIM_
Root: 6002  SB: 6002/630  Size: 6 words  Type: 000000 (PCL)
Keys: 060013
Call at 4(0)/171752 SB: 7777(0)/0 LB: 0(0)/6002(6)

(trace)> quit
-> ttybuf 1

User 1 message buffer (600 bytes long) at 7(0)/17224:
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..................User 30: Phantom requested terminal input.


Input buffer (400 bytes long) for user 1 at 7(0)/7142:
    a 2267
    ?q
    avail
    avail
    ld x
    date
    a wsh>tests
    r ph_file_copy 1 ring0.map
    x.print fc1.como
    a wsh>tests
    r ph_file_copy 1 ring0.map
    ph pager.ph
    a billh
    ph pager.ph
    vpsd
    sh 15
    a 2267
    ?q
    lo all
    max

    date
    se -100285 -1920
    date
    stat us
    date
    stat us
    .
    ll
    stat us
    ld


Output buffer (600 bytes long) for user 1 at 7(0)/0:
    . 20
Maximum number of program invocations: 20
Maximum number of private static segments: 128
Maximum number of private dynamic segments: 64

  09:57:20  6.727  0.033  level 2

User                              No  Line Devices
SYSTEM                             1   asr <DOLFIN>
```

← ttybuf command

```
NETMAN                          29   nsp <DOLFIN>

OK 09:57:32  0.109  0.000  level 2

-> print_locks

SLOK: Locked for reading by 1 user(s).
 0 reader(s) waiting
No writer(s) waiting

UFDLOK: Unlocked.
No reader(s) waiting
No writer(s) waiting

BLKLOK: Unlocked.
No reader(s) waiting
No writer(s) waiting

SDLOK: Unlocked.
No reader(s) waiting
No writer(s) waiting

TRNLOK: Unlocked.
No reader(s) waiting
No writer(s) waiting

UTLOK: Unlocked.
No reader(s) waiting
No writer(s) waiting

RATLOK: Unlocked.
No reader(s) waiting
No writer(s) waiting

EVLCK: Unlocked.
 reader(s) waiting
No writer(s) waiting

SP1LCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

NETLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

NMMLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

SLCLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

MOVLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

SEGLCK: Unlocked.
No reader(s) waiting
No writer(s) waiting

 LCK: Unlocked.
 eader(s) waiting
 writer(s) waiting
```

← print_locks command

```
-> ready_list

START -> USR001
           |
           v
        BK1PCB -> BK2PCB
-> sysreg
PSWPB: 6(0)/42313    PSWKEYS: 014100 100077

DMA channel     I/O address      Word Count
      0         0(0)/172110        000000
      2         0(0)/176000        000000
      6         0(0)/1000          000000
     14         0(0)/151005        007405
     16         0(0)/145005        000000
     20         0(0)/171000        000000
     22         0(0)/175000        000000
     24         0(0)/100255        000300
     26         0(0)/72001         000200
     30         0(0)/10003         002200
     32         0(0)/41272         002200
     34         0(0)/201           002027
     36         0(0)/4003          003500
-> clra

-> lookup_address 6/11675 ecb

   ROUTINE:  LOGEV1 + 3434 from ECB
-> lookup_address  37/100 common

   COMMON:  LSMCOM + 100
-> lookup_address 13/11364 lbn

   ROUTINE:  LISTN_
   ROUTINE:  LISTEN_
   ROUTINE:  LISTEN_C
-> lookup_address 13/7000 any

   ROUTINE:  LISTEN_C + 371 from PB



-> lookup_symbol  prwf$$

 ECB of routine: 11(0)/35543
 PB of routine: 11(0)/33046
 LB of routine: 11(0)/35136
-> lookup_symbol lsmcom

   Address of common: 37(0)/0
-> lookup_symbol r0dbg_on

   Address of other: 14(0)/770
->
```

← ready_list command

← system_registers command

← lookup_address command

← lookup_symbol command

Demonstration 4

```
-> clearall
-> brk gpath$+33
-> continue
Leaving the debugger
vpsd

 = 1
4000/ 1 STA#  177740,•X   ?
$quit

OK ld

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62067 (GPATH$ + 33).

-> define_variable gpath$\code sb%+44 decimal 1
-> define_variable msg 4000/300 ascii 20
-> define_variable ptr 4000/100 pointer
-> define_variable bit_str 4000/200 bit

-> display_variable code
Procedure           Variable        Address         Type        Length
GPATH$              CODE            SB%+44          DECIMAL          1

-> display_variable
Procedure           Variable        Address         Type        Length
GPATH$              CODE            SB%+44          DECIMAL          1
                    MSG             4000(0)/300     ASCII           20
                    PTR             4000(0)/100     POINTER          2
                    BIT_STR         4000(0)/200     BIT              1
-> : code
    0
-> let msg = 'It''s only a test'
-> let ptr = 55(0)/12000
-> let bit_str = 10110
-> : msg
It's only a test
-> : ptr
55(0)/12000
-> : bit_str
1011000000000000
-> : bit_str decimal
 -20480
-> delete_variable code
-> delete_variable

OK to delete all defined variables? yes

-> define_command com1 clearall; breakpoint prwf$$+1; continue; com2
-> define_command com2 dump • •; step 1; com2

-> display_command

  Defined command:
  COM1          : CLEARALL;BREAKPOINT PRWF$$+1;CONTINUE;COM2
  COM2          : DUMP • •;STEP;COM2

-> com1
-> CLEARALL;BREAKPOINT PRWF$$+1;CONTINUE;COM2
-> BREAKPOINT PRWF$$+1;CONTINUE;COM2
-> CONTINUE;COM2
  ving the debugger.

Debugger entered due to breakpoint/single step.
```

get from "xref" listing

← define_variable command

← display_variable command

← : command

← let command

← delete_variable command

← define_command command

← recursive definition

← display_command command

```
     Process 1 was executing at 11(0)/33047 (PRWF$$ + 1).
-> COM2
-> DUMP * *;STEP;COM2

11(0)/33047  LDA% SB%+ 70 ,*
-> STEP;COM2

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/33051 (PRWF$$ + 3).
-> COM2
-> DUMP * *;STEP;COM2

11(0)/33051  STA# SB%+ 34
-> STEP;COM2

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/33052 (PRWF$$ + 4).
-> COM2
-> DUMP * *;STEP;COM2

quit.
-> clearall
-> delete_command

OK to delete all defined commands? yes
```

expansion of previously defined command

recursive "sub" step

delete_command command

```
-> help
:                       Access                  Access_REGister
Access_TYPE             ARGumentS               BReaKpoint
CLeaR                   CLeaRAll                Continue
DEFine_COMmand          DEFine_VARiable         DELete_COMmand
DELete_VARiable         DISplay_COMmand         DISplay_VARiable
Dump                    Help                    LET
LIST                    LiSTAll                 LOOKAT
Lookup_Address          Lookup_Symbol           Pcb
Print_LOCKS             ReaDY_LiST              SeaRCH
STATus                  Step                    STRUCture
SYStem_REGisters        Trace                   Translate_to_PHYSical
Translate_to_VIRtual    TTybuf                  WHere

-> help trace
Command name:   Trace  (T)

   Command description:
   Print stack frames for the given process.

   Command line arguments:
   [<process-number> [<address-expression>]]

   where address-expression may be one of the following:
   { <segno>/<wordno> | <symbol-name> | LB% | SB% | XB% | * }
   [ + <offset> | - <offset>]

   where the following commands can be used within TRACE:
   Quit | Father [<levels>] | Son [<levels>] | Current | Goto [<levels>] |
   Arguments | STack [<start> [<end>]] | User <process-number> [<address>]
```

help command

```
-> structure

PRIMOS data bases known to the Debugger:
```

structure command

```
CLDATA                  DISK_QUEUE_BLOCK        ECB
FIGCOM                  PUDCOM                  PUSTAK
SUPCOM                  UPCOM
-> structure ecb 11/35543
Structure ECB at 11(0)/35543.

   Offset | Field name          | Value
   -------|---------------------|-----------------
   035543 | pb                  | 11(0)/33046
   035545 | frame size          | 000202
   035546 | stack root          | 000000
   035547 | args displac        | 000070
   035550 | num of args         |      7
   035551 | lb                  | 11(0)/35136
   035553 | keys                | 014000

-> structure figcom
Structure FIGCOM at 14(0)/700.

   Offset | Field name          | Value
   -------|---------------------|-----------------
   000700 | LOUTQM              | 32767
   000701 | LOTLIM              |      3
   000702 | DONSTP              |      0
   000703 | DLOGOT              |      0
   000704 | SPCH1/DERA          | 000210
   000705 | SPCH2/DEKL          | 000277
   000706 | PRI500              |      1
   000707 | VERSIO              | 20.1.1osdbg
   000720 | NLGPRT              |      1
   000721 | LOGOVR              |      0
   000722 | LRQUOT              |      0
   000723 | DMQMSK              | 1111111111111111
   000724 | CPUID               |      6
   000725 | INSTLB              |      0
   000726 | APCNFG              |      0
   000727 | UPSSW               |     -1
   000730 | CPUREV              |      2
   000731 | STAMP               | 09/23/85**15:15:43**19.2000000
   000750 | RWLOCK              |      1
   000751 | ABBRSW              |      1
   000752 | SLVRUN              |      0
   000753 | DTRDRP              |      0
   000754 | ZCPU                |      0
   000755 | STTMCP              |      0
   000756 | MAPREV              |      0
   000757 | RGSETS              |      2
   000760 | RGSET0              |      3
   000761 | ECCTRL              |      0
   000762 | BCLOCK              |      0
   000763 | SENSOR              |      0
   000764 | MEMHLT              |      1
   000765 | DISPCH              |      0
   000766 | LOGBAD              |      0
   000767 | DEFMEL              |     16
   000770 | R0DBG_ON            | 100000
   000771 | SUSPEND_SLAV        |      0

 translate_to_physical 13/22000

 rtual address 13(0)/22000 translates to physical address 1514000.
This address is on physical page 646.
```

← initial set of defined structures

← translate_to_physical command

```
-> translate_to_physical 6/1000

Virtual address 4000(0)/100 translates to physical address 63000.
This address is on physical page 31.

-> translate_to_virtual 646

Corresponding virtual address is 13(0)/22000 for process 1.

-> translate_to_virtual 31

Corresponding virtual address is 6(0)/0 for process 1.
```

← translate_to_virtual command

Demonstration 5.1

```
date
03 Oct 85 15:19:52 Thursday
OK 15:19:55  0.169  0.027
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> breakpoint gpath$+1
-> continue
Leaving the debugger.
id x

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62035 (GPATH$ + 1).
-> arguments
Current routine: GPATH$

6 arguments at SB%+56:
# 1 at 41(3)/124207  : 000001.140040
# 2 at 6002(3)/1505  : 000037.000001
# 3 at 6002(3)/5223  : 136304.100000
# 4 at 41(3)/124530  : 000200.140040
# 5 at 6002(3)/3313  : 035045.000010
# 6 at 6002(3)/731   : 000000.000000
-> define_variable  xkey 41/124207
-> defvar xunit 6002/1505
-> defvar xpathname 6002/5223  ascii 128
-> defvar xmax_chars 41/124530
-> defvar xpath_len 6002/3313 decimal 1
-> defvar xcode 6002/731
-> defvar gpath$\key sb%+42
-> defvar gpath$\valid_segment sb%+36
-> defvar gpath$\uteptr sb%+100 pointer 2
 > defvar gpath$\code sb%+44
-> defvar gpath$\unit_open sb%+35
   display_variable
rocedure          Variable        Address          Type          Length
                  XKEY            41(0)/124207     OCTAL              1
                  XUNIT           6002(0)/1505     OCTAL              1
                  XPATHNAME       6002(0)/5223     ASCII            128
                  XMAX_CHARS      41(0)/124530     OCTAL              1
                  XPATH_LEN       6002(0)/3313     DECIMAL            1
                  XCODE           6002(0)/731      OCTAL              1
GPATH$            KEY             SB%+42           OCTAL              1
GPATH$            VALID_SEGMENT   SB%+36           OCTAL              1
GPATH$            UTEPTR          SB%+100          POINTER            2
GPATH$            CODE            SB%+44           OCTAL              1
GPATH$            UNIT_OPEN       SB%+35           OCTAL              1
-> where

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62035 (GPATH$ + 1).
-> : key
100000
-> : xkey
000001
-> step 3

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62041 (GPATH$ + 5).
-> : key
  \001
   : valid_segment
  \00
   tep 2
```

Testing GPATH$ (one path)

← breakpoint at start of module

← check input arguments

← define program variables

← start sequencing through the code

```
Debugger entered due to breakpoint/single step.
     Process 1 was executing at 6(0)/26637 (LOCKFS).
-> : valid_segment
000000
-> breakpoint gpath$+10
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62044 (GPATH$ + 10).
-> status user
Process 1  SYSTEM                          *** Owns register set 0 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/62044 (GPATH$ + 10)
   LB:    11(0)/62432 (GPATH$)
   Locks owned: FSLOK

Process 29  (Login name is not resident)
   Level: Network process
   Type:  Network process
   State: Waiting at 12(0)/25302 (PNTSEM)
   PB:    6(0)/34454 (WAIT + 4)
   LB:    6(0)/34156 (SETSWI (et al))

-> clearall
-> : key
000001
-> breakpoint gpath$+22
-> continue
Leaving the debugger.

 ebugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62056 (GPATH$ + 22).
-> access_type symbolic
-> access *
11(0)/62056  PCL% LB%+ 424 ,*
11(0)/62060  AP SB%+ 61 ,*S
11(0)/62062  AP SB%+ 100 ,S
11(0)/62064  AP SB%+ 44 ,SL
11(0)/62066  STA# SB%+ 35   ?
-> : xunit
000037
-> : uteptr
*177776/000000
-> : code
000001
-> clearall
-> breakpoint gpath$+33
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62067 (GPATH$ + 33).
-> : xunit
000037
-> : uteptr
717(0)/6376
-> : code
000000
   step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62070 (GPATH$ + 34).
```

FS lock has been taken

call to open_chk

arguments passed to open_chk

arguments after call to open_chk

```
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62071 (GPATH$ + 35).
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62077 (GPATH$ + 43).
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62206 (GPATH$ + 152).
-> : unit_open
100000
-> clearall
-> breakpoint gpath$+155
-> continue
Leaving the debugger..

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62211 (GPATH$ + 155).
-> access_type symbolic
-> access *
11(0)/62211  CRA
11(0)/62212  STA% SB%+ 72 ,*
11(0)/62214  EAL% LB%+ 432 ,*
11(0)/62216  JSXB% LB%+ 434 ,*
11(0)/62220  PCL% LB%+ 436 ,*  ?
-> breakpoint 11/62220
-> listall
Type   Address        Procedure          Process  Count  Mnemonic
brkpt  11(0)/62211    GPATH$ + 155         Any        1    CRA
brkpt  11(0)/62220    GPATH$ + 164         Any        1    PCL
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62220 (GPATH$ + 164).
-> status user
Process 1  SYSTEM                          *** Owns register set 1 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/62240 (GPATH$ + 204)
   LB:    11(0)/62432 (GPATH$)
   Locks owned: FSLOK  UFDLOK
   PCB abort flags: TSEALM

Process 29  (Login name is not resident)
   Level: Network process
   Type:  Network process
   State: Waiting at 12(0)/25302 (PNTSEM)
   PB:    6(0)/34454 (WAIT + 4)
   LB:    6(0)/34156 (SETSWI (et al))

-> clearall
-> : uteptr
717(0)/6376
-> access_type octal
-> access 717/6376+2
717(0)/6400  000000
  '(0)/6401  000264  ?
  ccess 717/6376+6
 (0)/6404  000000  ?
-> : xpathname
<D00FI0000'05a100+0040805Y0000001 00000'050100'0040005 >E000NpaV/000000V.006200
```

end of select statement

call to lockr

call to ra2pth

UFD lock taken

arguments to ra2pth

```
  ●●●●●●●●●●.  ●●●●●●●●●●●●●●y●c●●● ●●●●●● d●●●●●●
-> : xmax_chars
000200
-> : xpath_len
    0
-> : code
000000
-> clearall
-> breakpoint gpath$+201

*** Debugger user error:
    Unknown V-mode instruction.
-> access_type symbolic
-> access gpath$+164
11(0)/62220  PCL% LB%+ 436 ,*
11(0)/62222  AP SB%+ 100 ,*
11(0)/62224  AP XB%+ 2 ,S
11(0)/62226  AP XB%+ 6 ,S
11(0)/62230  AP SB%+ 64 ,*S
11(0)/62232  AP SB%+ 67 ,*S
11(0)/62234  AP SB%+ 72 ,*S
11(0)/62236  AP SB%+ 44 ,SL
11(0)/62240  EAL% LB%+ 432 ,*  ?
-> breakpoint 11/62240
-> listall
Type   Address         Procedure          Process  Count  Mnemonic
brkpt  11(0)/62240    GPATH$ + 204         Any        1    EAL
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62240 (GPATH$ + 204).
-> : xpathname
:DOLFIN>CMDNC0>LD.RUN●8●5Y●●●●●●!●●●●●'●5●!●●'●●4●●●5 >E●●●NpeV/●●●●●●V.●●6200
  ●●●●●●●●●●.  ●●●●●●●●●●●●●<DOLFIN>CMDNC0>LD.RUN
-> : xpath_len
   21
-> access_type symbolic
-> access *
11(0)/62240  EAL% LB%+ 432 ,*
11(0)/62242  JSXB% LB%+ 440 ,*
11(0)/62244  JMP#  62475   ?
-> clearall
-> breakpoint 11/62244
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62244 (GPATH$ + 210).
-> status user
Process 1  SYSTEM                          *** Owns register set 0 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/62244 (GPATH$ + 210)
   LB:    11(0)/62432 (GPATH$)
   Locks owned: FSLOK
   Outstanding abort flags: TSEALM

Process 29  (Login name is not resident)
   Level: Network process
   Type:  Network process
   State: Waiting at 12(0)/25302 (PNTSEM)
   PB:    6(0)/34454 (WAIT + 4)
   LB:    6(0)/34156 (SETSWI (et al))
```

invalid offset in listing

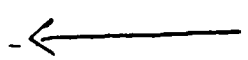
call to ra2pth

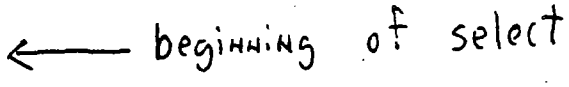
arguments after call to ra2pth

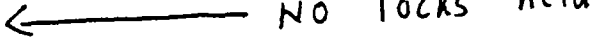
call to UNLKN to free UFD lock

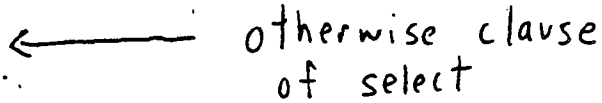
UFD lock released

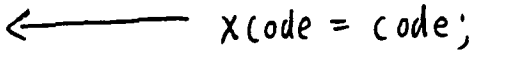
```
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62475 (GPATH$ + 441).
-> : code
000000
-> access .
11(0)/62475  JSXB% LB%+ 456 ,.
11(0)/62477  LDA# SB%+ 44
11(0)/62500  CAS#  63002
11(0)/62501  JMP#  62503
11(0)/62502  JMP#  62507
11(0)/62503  CAS#  63003   ?
-> breakpoint 11/62477
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62477 (GPATH$ + 443).
-> status user
Process 1  SYSTEM                          *** Owns register set 0 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/62477 (GPATH$ + 443)
   LB:    11(0)/62432 (GPATH$)

Process 29  (Login name is not resident)
   Level: Network process
   Type:  Network process
   State: Waiting at 12(0)/25302 (PNTSEM)
   PB:    6(0)/34454 (WAIT + 4)
   LB:    6(0)/34156 (SETSWI (et al))

-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62500 (GPATH$ + 444).
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62503 (GPATH$ + 447).
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62506 (GPATH$ + 452).
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/63013 (GPATH$ + 757).
-> access .
11(0)/63013  LDA# SB%+ 44
11(0)/63014  STA% SB%+ 75 ,.
11(0)/63016  PRTN   ?
-> : code
000000
-> : xcode
000000
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/63014 (GPATH$ + 760).
-> step

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/63016 (GPATH$ + 762).
```











```
-> : xpathname
<DOLFIN>CMDNC0>LD.RUN@8@5Y@@@@@@!@@@@@'@5@!@@'@@4@@@5 >E@@@Np@V/@@@@@@V,@@62@@
  @@@@@@@@@@@.  @@@@@@@@@@@@@<DOLFIN>CMDNC0>LD.RUN
-> : xcode
000000
-> : xpath_len
   21
-> clearall
-> continue
Leaving the debugger.

<DOLFIN>CMDNC0 (ALL access)
3965 records in this directory, 3965 total records out of quota of 0.

No entries selected.

OK 15:20:12  2.603  0.000
```

valves returned by the call to GPATH$

Forcing an error

```
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> access gpath$+160
11(0)/62214  EAL% LB%+ 432 ,*
11(0)/62216  JSXB% LB%+ 434 ,*
11(0)/62220  PCL% LB%+ 436 ,*
11(0)/62222  AP SB%+ 100 ,*
11(0)/62224  AP XB%+ 2,S
11(0)/62226  AP XB%+ 6 ,S
11(0)/62230  AP SB%+ 64 ,*S
11(0)/62232  AP SB%+ 67 ,*S
11(0)/62234  AP SB%+ 72 ,*S
 1(0)/62236  AP SB%+ 44 ,SL
 1(0)/62240  EAL% LB%+ 432 ,*  ?
-> breakpoint 11/62240
-> continue
Leaving the debugger.
ld x

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/62240 (GPATH$ + 204).
-> : code
000000
-> let code=43
-> : code
000043
-> continue
Leaving the debugger.
Buffer too small.  LD (std$cp)
ER 15:21:34  0.166  0.033
```

call to ra2pth

after call to ra2pth

modify code to force an error

result of simulated error

Demonstration 5.2

Analyzing a hung system

```
a wsh>tests
OK 15:27:07  0.178  0.269
r ph_file_copy 1 ring0.map
PHANTOM is user 31
OK 15:27:20  0.393  0.042

 debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
->
CP> stop
HALTED AT 000056/001750: 005103

CP> sysclr

*** CPU VERIFIED ***

CP> run 1001
lookat
Active process is -20.
-> continue
Leaving the debugger.

***** WARM START *****
date
Improper command name. "DATE" (std$cp)
ER 15:27:44  0.206  1.096
date
03 Oct 85 15:27:48 Thursday
OK 15:27:50  0.121  0.000
avail
     Volume DOLFIN
     118512 total records
      11306 records available
      90.5% full

OK 15:28:03  0.581  0.260
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> breakpoint amlpb
-> continue
Leaving the debugger.

Debugger entered due to breakpoint/single step.
     Process -15 (AMLPCB) was executing at 16(0)/4253 (AMLDIM).
-> clearall
-> ready_list
START -> AMLPCB
           |
           V
         DK1PCB -> PNCPCB
           |
           V
         BK1PCB -> BK2PCB
-> step 100

Debugger entered due to breakpoint/single step.
     Process -15 (AMLPCB) was executing at 16(0)/6055 (TTHOUT + 110).
-> ready_list
START -> AMLPCB
           |
           V
         DK4PCB -> DK3PCB -> DK2PCB -> DK1PCB -> PNCPCB
           |
```

← warmstart while in debugger

← after warmstart still in debugger

← set breakpoint in AMLDIM

← step of AMLDIM causes ready_list to fill up

```
          v
        USR029
          |
          v
        BK1PCB -> BK2PCB
->
CP> stop
HALTED AT 000056/002136: 013074

CP> sysclr

*** CPU VERIFIED ***

CP> run 1001

*** Fault while in debugger:
    Segment fault (type 60) encountered at 55(0)/1205.
    Attempt to reference 3403(0)/0.
-> ready_list
START -> CLKPCB
          |
          v
        USR029
          |
          v
        BK1PCB -> BK2PCB
-> continue
Leaving the debugger.

***** WARM START *****
avail
\
Debugger entered due to console interrupt.
     Process 29 was executing at 12(0)/50633 (RNGRCV + 201).
-> ready_list
START -> CLKPCB
          |
          v
        USR029
          |
          v
        BK1PCB -> BK2PCB
-> status user
Process 1  SYSTEM
   Level: System process
   Type:  Supervisor
   State: Waiting at 6(0)/107167 (DKRQB + 1167)
   PB:    6(0)/35703 (DKTWO_ + 165)
   LB:    6(0)/41334 (RREC (et al))
   Locks owned: FSLOK
   PCB abort flags: MINALM

Process 29  (Login name is not resident)
   Level: Network process
   Type:  Network process
   State: Ready
   PB:    12(0)/50633 (RNGRCV + 201)
   LB:    12(0)/51112 (RNGRCV)
   Locks owned: NETLCK

Process 31  SYSTEM
   Level: Priority 1 user
   Type:  CPL phantom
   State: Waiting at 6(0)/106645 (DKRQB + 645)
   PB:    6(0)/35703 (DKTWO_ + 165)
   LB:    6(0)/41334 (RREC (et al))
```

warmstart while in debugger

?

all interrupt processes except the clock process are removed

system is hung

waiting for disk requests to complete

```
   Locks owned: FSLOK
   Outstanding abort flags: TSEALM
   PCB abort flags: QUTALM

-> status -7
Process -7  DK1PCB
   Level: Disk/Ringnet process
   State: Ready
   PB:    6(0)/40233 (DMA_ERR + 720)
   LB:    6(0)/106244 (Unknown)
   SB:    4(0)/164070        XB:     0(0)/1200
   L:     000000 000000      E:      000000 000000
   X:     000000             Y:      000000
   FAR0:  000000 000000      FLR0:   000000 000000
   FAR1:  000000 000000      FLR1:   000000 000000
   Keys:  134201

-> ready_list
START -> CLKPCB
           |
           V
         USR029
           |
           V
         BK1PCB -> BK2PCB
-> pcb -7

Process:   -7
Level:  616                        Link: 076700
Wait list: 0(0)/534                Abort flags: 0000000000000000
PB: 6(0)/40233                     SB: 4(0)/164070
LB: 6(0)/106244                    XB: 0(0)/1200
L: 000000 000000                   E: 000000 000000
   000000                          Y: 000000
FAR0: 000000 000000                FLR0: 000000 000000
FAR1: 000000 000000                FLR1: 000000 000000
Interval timer: 120273 000000      Elapsed timer: 000000 100000
DTAR2: 177700 000000               DTAR3: 177700 000000
Keys: 134201
->
```

disk process is ready but ...

it is NOT on the ready list

⇒ disk process is IN limbo

Demonstration 6

```
OK 17:21:06  0.057  0.000
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> breakpoint pagtur

*** Debugger user error:
    Breakpoints cannot be set on an ARGT instruction.
-> access_type symbolic
-> access pagtur
6(0)/45204  ARGT   ?
-> breakpoint fault+60

*** Debugger user error:
    Breakpoints are not allowed for this instruction.
-> access fault+60
6(0)/31112  CALF 32420   ?
-> breakpoint pagtur+1
-> a pagtur+1
6(0)/45205  CRA   sze

*** Debugger user error:
    Can't modify breakpointed memory. Operation aborted.
6(0)/45206  STA# SB%+ 22   ?
-> clearall
-> access prwf$$+35
11(0)/33103  PCL% LB%+ 430 ,*
11(0)/33105  AP SB%+ 73 ,*S
11(0)/33107  AP SB%+ 150 ,S
11(0)/33111  AP SB%+ 44 ,SL
11(0)/33113  SAS 1   ?
-> breakpoint prwf$$+35
-> listall
Type   Address        Procedure          Process  Count  Mnemonic
brkpt  11(0)/33103    PRWF$$ + 35          Any        1    PCL
-> continue
Leaving the debugger.
avail

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 11(0)/33103 (PRWF$$ + 35).
-> status
Process 1  SYSTEM                          *** Owns register set 0 ***
   Level: System process
   Type:  Supervisor
   State: Ready
   PB:    11(0)/33113 (PRWF$$ + 45)
   LB:    11(0)/35136 (PRWF$$)
   SB:    6003(0)/252        XB:     6(0)/27052
   L:     000000 033103      E:      000000 000006
   X:     000000             Y:      177777
   FAR0:  006000 016000      FLR0:   000040 000000
   FAR1:  000006 003124      FLR1:   000006 000000
   Keys:  014100             Modals: 100037
   Fcode: 001100 000040      Faddr:  55(0)/1703
   Locks owned: FSLOK

-> clearall
-> continue
'eaving the debugger.
     Volume DOLFIN
     118512 total records
      11301 records available
      90.5% full
```

breakpoints cannot be set on certain instructions

can't change code where breakpoints are installed

breakpoint on a pcl instruction

PB has been advanced since PCL has executed

```
OK 17:26:53  1.115  3.206
\
Debugger entered due to console interrupt.
     Process -20 (BK1PCB) was executing at 6(0)/42313 (BK2PB + 4).
-> step

*** Debugger user error:
    Step presumes entry from a breakpoint/single step.
-> breakpoint p$cidx+1
-> continue
Leaving the debugger.
avail

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(3)/133503 (P$CIDX + 1).
-> step 10000

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 41(3)/133503 (P$CIDX + 1).
-> listall
Type   Address         Procedure          Process  Count  Mnemonic
brkpt  41(0)/133503    P$CIDX + 1           Any        1    LDA
step   41(0)/133504    P$CIDX + 2             1     9904    STA
-> step

*** Debugger user error:
    There is already an actively stepping process.
-> clearall
-> listall
No breakpoints are set.
-> breakpoint pagtur+1
-> continue
 eaving the debugger.

 ugger entered due to breakpoint/single step.
     Process 1 was executing at 6(0)/45205 (PAGTUR + 1).
-> clearall
->.step 2000

*** Debugger user error:
    Stepping through a critical region is not allowed.

Debugger entered due to breakpoint/single step.
     Process 1 was executing at 6(0)/26223 (LKPRV_ + 7).
-> access_type symbolic
-> access *
6(0)/26223  INHP   ?
-> access nlogin
15(0)/2267
*** Fault while in debugger:
    Page fault (type 10) encountered at 55(0)/15227.
    Attempt to reference 15(0)/2267.
-> clearall
-> breakpoint nlogin
-> listall
Type   Address         Procedure          Process  Count  Mnemonic
brkpt  15(0)/2267      NLOGIN               Any        1
-> continue
Leaving the debugger.
     Volume DOLFIN
     118512 total records
      11301 records available
      90.5% full

OK 17:27:11  0.660  0.245
```

can't step from a console interrupt

can't step while any process is still actively stepping

step aborts because of hardware interrupt inhibit instructions

"pended" breakpoint

can't show mnemonic

```
vpsd

$sn 15

$a 2267

*** Debugger user error:
    Breakpoints cannot be set on an ARGT instruction.

Debugger entered due to page fault.
     Process 1 was executing at 4000(3)/62450 (Unknown).
-> clearall
-> continue
Leaving the debugger.
15/ 2267 ARGT  ?
$q
OK 17:27:46  0.287  0.263
```

← cause the page to be brought IN.

← deferred - error message from a pended breakpoint

```
000000: (0001)   /* GPATH$.PLP, PRIMOS>FS, PRIMOS GROUP, 06/25/84
000000: (0002)      Return a pathname given a unit, attach point or segment number.
000000: (0003)      Copyright (c) 1981, Prime Computer, Inc., Natick, MA 01760 */
000000: (0004)
000000: (0005)   /* Description:
000000: (0006)   /* Returns the pathname of the unit, attach point or segment specified.
000000: (0007)   /*
000000: (0008)   /* Abnormal conditions:  None.
000000: (0009)   /*
000000: (0010)   /* Implementation:  Calls Ra2pth if local, R$call if remote.
000000: (0011)   /*
000000: (0012)   /* Gate Gpath$: return pathname of unit, attach point or segment.
000000: (0013)   /* Calling sequence:
000000: (0014)   /*
000000: (0015)   /*     dcl Gpath$ entry (fixed bin, fixed bin, char (*), fixed bin, fixed bin, fixed bin);
000000: (0016)   /*     call Gpath$ (key, unit, pathname, max_path_len, actual_path_len, code);
000000: (0017)   /*
000000: (0018)   /*     key:              May be any of the following (input):
000000: (0019)   /*                       K$UNIT — use passed unit number.
000000: (0020)   /*                       K$CURA — get pathname of current a.p.
000000: (0021)   /*                       K$INIA — get pathname of initial a.p.
000000: (0022)   /*                       K$SEGN — use passed segment number.
000000: (0023)   /*     unit:             unit number of unit to check or segment number
000000: (0024)   /*                       if key = K$SEGN. (input)
000000: (0025)   /*     pathname:         returned pathname of unit, attach point or segment (output).
000000: (0026)   /*     max_path_len:     size of pathname buffer in characters (input).
000000: (0027)   /*     actual_path_len:  length of returned pathname in characters (output).
000000: (0028)   /*     code:             standard error code (output).
000000: (0029)   /*
000000: (0030)   /* Logical structure of module Gpath$:
000000: (0031)   /*
000000: (0032)   /* If (unit open and local)
000000: (0033)   /*   Then
000000: (0034)   /*     (get idev, bra from AST entry and then get pathname with Ra2pth)
000000: (0035)   /* If (segment number)
000000: (0036)   /*   Then
000000: (0037)   /*     (get pathname with Ra2pth)
000000: (0038)   /* Select (error code)
000000: (0039)   /*   When (illegal remote reference)
000000: (0040)   /*     (go remote with R$call)
000000: (0041)   /*   When (unit not open)
000000: (0042)   /*     If (checking attach point)
000000: (0043)   /*       Then
000000: (0044)   /*         (map to not attached for user)
000000: (0045)   /* Return
000000: (0046)   /*
000000: (0047)   /* ********************************************************
000000: (0048)   /* * This module takes the FS and UFD locks for reading. *
000000: (0049)   /* ********************************************************
000000: (0050)   /*
000000: (0051)   /* Modifications:
000000: (0052)   /*   Date    Programmer     Description of modification
000000: (0053)   /* 06/25/84 Sadigh         Return E$UNOP if key is K$COMO and COMO is not open.
000000: (0054)   /* 02/18/84 Sadigh         Added k$como option to return pathname of COMOUTPUT
```

(32)

Selected portions of a listing of GPATH$

```
000000: (0055)   /*                          file.
000000: (0056)   /* 01/24/84 Slutz           Added init for valid_segment.
000000: (0057)   /* 11/29/83 Slutz           Fixed for Dynamic File Units.
000000: (0058)   /* 08/09/83 Goggin          Added new key K$SEGN.
000000: (0059)   /* 10/05/82 Swartzendruber Use slave ID instead of NPX node.
000000: (0060)   /* 11/17/81 Slutz           Return correct error code for k$unit.
000000: (0061)   /* 11/09/81 Slutz           Fixed r$call coding to handle spare byte at end.
000000: (0062)   /*                          Buffer length is in characters!
000000: (0063)   /* 08/28/81 Weinberg        Initial coding.
000000: (0064)      */
000000: (0065)
000000: (0066)   gpath$:
000000: (0067)           proc (xkey, xunit, xpathname, xmax_chars, xpath_len, xcode) options (
000000: (0068)                    gate, nocopy);
000001: (0069)
000001: (0070)   dcl xkey fixed bin,                          /* Determines whose pathname to get */
000001: (0071)       xunit fixed bin,                         /* Unit number if key = k$unit
000001: (0072)                                                   or segment number if key = k$segn*/
000001: (0073)       xpathname char (128),                    /* Name returned here */
000001: (0074)       xmax_chars fixed bin,                    /* Length of xpathname buffer in characte
000001: (0075)       xpath_len fixed bin,                     /* Length of returned name in characters
000001: (0076)       xcode fixed bin;                         /* Standard error code */
000001: (0077)
000001: (0101)
000001: (0102)   /* External entry points */
000001: (0103)
000001: (0104)   dcl ra2pth entry (fixed bin (31), fixed bin, char (*), fixed bin, fixed
000001: (0105)            bin, fixed bin),
000001: (0106)       open_chk entry (fixed bin, ptr, fixed bin) returns (bit (1)),
000001: (0107)       r$call entry options (variable),
000001: (0108)       ti$sgs entry () returns (fixed bin(15)),
000001: (0109)       sdwadr entry (fixed bin(15), fixed bin(15)) returns(ptr options(
000001: (0110)            short)),
000001: (0111)
000001: (0112)   /* Gets pointer to sdw for
000001: (0113)      given user and segment. */
000001: (0114)
000001: (0115)       pgmapa entry (ptr options(short), fixed bin(15)) returns(ptr options(
000001: (0116)            short));
000001: (0117)
000001: (0118)   /* Gets pointer to page map
000001: (0119)      entry for given user and segment. */
000001: (0120)
000001: (0121)   /* Local declarations */
000001: (0122)
000001: (0123)   %replace fam_i_gpath$_key by 235,
000001: (0124)            set_high_order_bit by 32768,        /* used to detect high order
000001: (0125)                                                   bit being set in SDW */
000001: (0126)            page_size by 1024,                  /* record or page size */
000001: (0127)            full_seg by 65536,                  /* segment size */
000001: (0128)            dtar2 by 2;
000001: (0129)
000001: (0130)   dcl rcode fixed bin,                         /* Remote error code */
000001: (0131)       uteptr ptr,                              /* Pointer to unit table entry */
```

```
000001: (0132)       unit_open bit (1) aligned,                    /* True if unit open and local */
000001: (0133)       valid_segment bit(1) aligned,                 /* True if valid segment given
000001: (0134)                                                        with use segment key */
000001: (0135)       runit fixed bin,                              /* Unit may be different if going remote
000001: (0136)       odd_length bit(1) aligned,                    /* True if odd max length */
000001: (0137)       odd_byte char(1) aligned,                     /* The odd byte to keep in remote case ON
000001: (0138)       key fixed bin,                                /* Local copies */
000001: (0139)       rkey fixed bin,                               /* key for remote system */
000001: (0140)       code fixed bin,
000001: (0141)       sdw_ptr ptr options (short),                  /* Pointer to SDW */
000001: (0142)       null_sdw fixed bin (31) static init (set_high_order_bit),
000001: (0143)       idx fixed bin,                                /* Index */
000001: (0144)       nvmfs fixed bin static external,              /* Number of AST entries */
000001: (0145)       ast_addr ptr options (short),                 /* Pointer to AST */
000001: (0146)       fb15 fixed bin based,
000001: (0147)       h_map_virt_addr ptr options (short),          /* Pointer to page map for EPF seg*/
000001: (0148)       h_map_ptr ptr options (short),                /* Pointer to page in which
000001: (0149)                                                        page map for EPF seg resides */
000001: (0150)       mapped_phys_page_addr fixed bin (31),         /* Physical address of HMAP of EPF segmen
000001: (0151)       user_phys_page_addr fixed bin (31),           /* Physical address of HMAP from SDW */
000001: (0152)       devno fixed bin,                              /* Logical device of file */
000001: (0153)       1 bra_bit based,                              /* Copy of BRA of the file */
000001: (0154)          2 h8 bit(8),
000001: (0155)          2 m8 bit(8),
000001: (0156)          2 l16 bit(16),
000001: (0157)       bra fixed bin (31),                           /* BRA of the file */
000001: (0158)       not_found bit(1) aligned,                     /* Boolean */
000001: (0159)       segno fixed bin,                              /* Segment */
000001: (0160)       dtar2_top fixed bin(15);                      /* top segment in dtar2 */
000001: (0161)
000001: (0162)    /* Validate parameters */
000001: (0163)
000001: (0164)            key = xkey;                              /* Make local copy */
000004: (0165)            valid_segment = '0'b;                    /* Init before checking.... */
000006: (0166)            call lockfs;                             /* No interruptions, please */
000010: (0167)
000010: (0168)            select (key);
000022: (0169)               when (k$unit)
000022: (0170)                  do;
000022: (0171)                     unit_open = open_chk (xunit, uteptr, code);
000033: (0172)                     if code = e$bkio
000033: (0173)                        then do;
000037: (0174)                        code = 0;
000041: (0175)                        unit_open = '1'b;
000043: (0176)                        end;
000043: (0177)                  end;
000043: (0178)               when (k$cura)
000043: (0179)                  unit_open = open_chk (current_ap_unit, uteptr, code);
000055: (0180)               when (k$homa)
000055: (0181)                  unit_open = open_chk (home_ap_unit, uteptr, code);
000067: (0182)               when (k$inia)
000067: (0183)                  unit_open = open_chk (initial_ap_unit, uteptr, code);
000101: (0184)               when (k$como)
000101: (0185)                  unit_open = open_chk (como_unit, uteptr, code);
```

(53)

```
000113: (0186)              when (k$segn)
000113: (0187)                 do;
000114: (0188)                    unit_open = '0'b;              /* Remember a unit has not
000116: (0189)                                                      been passed.*/
000116: (0190)                    segno = xunit;                 /* Parameter is a segment no */
000121: (0191)
000121: (0192)    /* Calculate the top segment number in DTAR2 for this user.  Check the
000121: (0193)       bit "pudcom.flagbt.big_dtar2" to determine whether 256 or 512 (dec). */
000121: (0194)
000121: (0195)                    dtar2_top = tl$sgs();
000124: (0196)                    if ((segno < dynsgs(dtar2)) | (segno > dtar2_top))
000124: (0197)                       then do;
000135: (0198)                       code = e$bkey;              /* Segment not in range. */
000137: (0199)                       valid_segment = '0'b;
000141: (0200)                       end;
000141: (0201)                    else valid_segment = '1'b;
000144: (0202)                    end;
000144: (0203)              otherwise
000144: (0204)                 do;
000145: (0205)                    unit_open = '0'b;
000147: (0206)                    valid_segment = '0'b;
000150: (0207)                    code = e$bkey;
000152: (0208)                    end;
000152: (0209)              end;                                 /* Select */
000152: (0210)
000152: (0211)           if unit_open
000152: (0212)              then do;                             /* Go get pathname from unit. */
000155: (0213)              call lockr (ufdlok);                 /* Must lock dirs to do it */
000161: (0214)              call ra2pth (uteptr -> utcme.bra, uteptr -> utcme.idevno, xpathname,
000161: (0215)                   xmax_chars, xpath_len, code);
000201: (0216)              call unlkn (ufdlok);
000205: (0217)              end;                                 /* Go get pathname from unit. */
000205: (0218)
000205: (0219)           else if valid_segment
000205: (0220)              then do;                             /* Go get pathname from segno. */
000211: (0221)              not_found = '1'b;
000213: (0222)              sdw_ptr = sdwadr (segno, pudcom.cusr);   /* Obtain a pointer to SDW */
000223: (0223)
000223: (0224)    /* Form physical address of page table from SDW  */
000223: (0225)
000223: (0226)              if (addr(sdw_ptr) -> long_fb ^= 0)    /* valid segno */
000223: (0227)                 then if (sdw_ptr -> long_fb ^= null_sdw)
000223: (0228)                    then do;                       /* Good SDW */
000233: (0229)                    user_phys_page_addr = sdw_ptr -> sdw.phys_low + full_seg *
000233: (0230)                          sdw_ptr -> sdw.phys_high;
000253: (0231)                    do idx = 1 to nvmfs while (not_found);   /* Search ASTs */
000266: (0232)                       ast_addr = addr (ast(idx));
000277: (0233)                       if (addr(ast_addr -> aste.nrnw) -> fb15 ^= 0)
000277: (0234)                          then do;                 /* This entry in use. */
000305: (0235)
000305: (0236)    /* Form physical address of page table for EPF segment */
000305: (0237)
000305: (0238)                             h_map_virt_addr = ast_addr -> aste.ppmap;
000311: (0239)                             h_map_ptr = pgmapa (h_map_virt_addr, pudcom.cusr);
```

```
000321: (0240)                         mapped_phys_page_addr = page_size * pgppn(h_map_ptr) + addr
000321: (0241)                                 (h_map_virt_addr) -> pointerb.w.wn;
000340: (0242)
000340: (0243)   /* If two addresses are equal get devno and bra */
000340: (0244)
000340: (0245)                         if mapped_phys_page_addr = user_phys_page_addr
000340: (0246)                            then do;
000344: (0247)                            devno = ast_addr -> aste.dev_bra.dvno;
000352: (0248)                            bra = 0;
000355: (0249)                            addr (bra) -> bra_bit.m8 = ast_addr -> aste.dev_bra.brah;
000363: (0250)                            addr (bra) -> bra_bit.l16 = ast_addr -> aste.dev_bra.bral;
000366: (0251)                            not_found = '0'b;
000370: (0252)                            end;
000370: (0253)                         end;
000370: (0254)                      end;                              /* Search ASTs */
000402: (0255)                   end;                                 /* Good SDW */
000402: (0256)
000402: (0257)             if not_found
000402: (0258)                then code = e$fntf;
000407: (0259)             else do;
000410: (0260)                call lockr (ufdlok);                    /* Must lock dirs to do it */
000414: (0261)                call ra2pth (bra, devno, xpathname, xmax_chars, xpath_len, code);
000432: (0262)                call unlkn (ufdlok);
000436: (0263)                end;
000436: (0264)             end;                                       /* if valid segment */
000436: (0265)
000436: (0266)          call unlkfs;                                  /* Done with file system */
000440: (0267)
000440: (0268)   /* Process according to error code */
000440: (0269)
000440: (0270)          select (code);
000450: (0271)             when (e$irem)
000450: (0272)                do;                                     /* Unit is remote, handle that */
000450: (0273)                   xpath_len = 0;                       /* Clear in case of error */
000453: (0274)                   odd_length = (mod(xmax_chars, 2) = 1); /* Remember state */
000467: (0275)                   if odd_length
000467: (0276)                      then odd_byte = substr(xpathname, xmax_chars + 1, 1);
000515: (0277)                   if (key = k$unit & (xunit >= sysun | xunit = como_unit)) |
000515: (0278)                      (key = k$como)
000515: (0279)                      then runit = uteptr -> rem_ute.master_to_slave;   /* Map unit number */
000547: (0280)                   else runit = xunit;    /* Attach point */
000556: (0281)                   if key = k$como
000556: (0282)                      then rkey = k$unit;
000564: (0283)                   else rkey = key;
000567: (0284)                   rcode = fam_i_gpath$_key;            /* For FAM I */
000571: (0285)                   call r$call (iamfs, uteptr -> rem_ute.slave_id, 'GPATH$', 6, rcode,
000571: (0286)                           rkey, 1, k$i2 + k$in,
000571: (0287)                           runit, 1, k$i2 + k$in,
000571: (0288)                           xpathname, xmax_chars, k$out + k$char + k$ref + 5,
000571: (0289)                           xmax_chars, 1, k$i2 + k$in,
000571: (0290)                           xpath_len, 1, k$i2 + k$out,
000571: (0291)                           xcode, 1, k$i2 + k$out,
000571: (0292)                           xunit, 1, k$i2 + k$in);   /* Needed for FAM I */
000672: (0293)                   if rcode ^= 0
```

```
000672: (0294)                      then xcode = rcode;          /* Network errors take precedence */
000677: (0295)                   if odd_length                   /* Restore odd byte to keep Tracy smiling
000677: (0296)                      then substr(xpathname, xmax_chars + 1, 1) = odd_byte;
000726: (0297)                end;
000726: (0298)             when (e$unop)
000726: (0299)                if key ^= k$unit & key ^= k$como
000726: (0300)                   then xcode = e$natt;           /* Map not open to not attached */
000741: (0301)                else xcode = code;                /* Correct for units */
000753: (0302)             otherwise
000753: (0303)                do;
000754: (0304)                   xcode = code;
000757: (0305)                   end;
000757: (0306)             end;                                 /* Select */
000757: (0307)
000757: (0308)          return;
000760: (0309)
000760: (0310)          end;                                    /* Gpath$ */
```

0000 ERRORS (PL/P rev 19.2)

PROCEDURE SIZE = 507 WORDS, LINKAGE FRAME SIZE = 50 WORDS
2111 SOURCE LINES, 193 STATEMENTS, COMPILATION TIME = 43.49 CPU SECONDS
52.2% DATA POOL UTILIZATION

(37)

SB%+44

```
001 000044S  CODE  bin(15) automatic
      130D                171A            172             174M            178A
                          180A            182A            184A            198M
                          207M            214A            257M            261A
                          270             301             304
001 000042S  KEY  bin(15) automatic
      130D                164M            168             277(2)          281
                          283             298(2)
001 000035S  UNIT_OPEN  bit(1) aligned automatic
      130D                171M            175M            178M            180M
                          182M            184M            188M            205M
                          211
001 000000X  1 UTCME  based
      82<137>D
     000000+00    2 VSTAT
      82<137>D
     000000+00        3 MODIFIED  bit(1)
      82<137>D
     000000+01        3 SYSUSE  bit(1)
      82<137>D
     000000+02        3 SHTBIT  bit(1)
      82<137>D
     000000+03        3 NO_CLOSE  bit(1)
      82<137>D
     000000+04        3 DISK_ERROR  bit(1)
      82<137>D
     000000+05        3 FILE_TYPE  bit(3)
      82<137>D
     000000+08        3 OPEN_MODE  bit(8)
      82<137>D
     000001+00    2 MORE_VSTAT
      82<137>D
     000001+00        3 REMOTE_UNIT  bit(1)
      82<137>D
     000001+01        3 NO_DTA_UPDATE  bit(1)
      82<137>D
     000001+02        3 BACKUP_USE  bit(1)
      82<137>D
     000001+03        3 SPARE  bit(13)
      82<137>D
     000002+00    2 BRA  bin(31)
      82<137>D          214A
     000004+00    2 CUR_RA  bin(31)
      82<137>D
     000006+00    2 LDEVNO  bin(15)
      82<137>D          214A
     000007+00    2 REL_WORDNO  bin(15)
      82<137>D
     000010+00    2 REL_RECNO  bin(31)
      82<137>D
     000012+00    2 RWLOCK  bit(8)
      82<137>D
     000012+08    2 ACCESS
      82<137>D
```

```
     000012+08      3 PROTECT  bit(1)
       82<137>D
     000012+09      3 DELETE  bit(1)
       82<137>D
     000012+10      3 ADD  bit(1)
       82<137>D
     000012+11      3 LIST  bit(1)
       82<137>D
     000012+12      3 USE  bit(1)
       82<137>D
     000012+13      3 EXECUTE  bit(1)
       82<137>D
     000012+14      3 WRITE  bit(1)
       82<137>D
     000012+15      3 READ  bit(1)
       82<137>D
     000013+00   2 POS_IN_PARENT  bin(15)
       82<137>D
     000014+00   2 PARENT_BRA  bin(31)
       82<137>D
     000016+00   2 HASH_THREAD  pointer
       82<137>D
     000020+00   2 QUOTA_BLK_PTR  pointer
       82<137>D
     000022+00   2 DIR_BLK_PTR  pointer
       82<137>D
     000024+00   2 DAM_IDX_RA  bin(31)
       82<137>D
     000026+00   2 EX_MAP_PTR  pointer
       82<137>D

001          UTCME_CHARS  char(48) based
       82<189>D
001 000100S  UTEPTR  pointer automatic
       130D              171A          178A          180A          182A
                         184A          214(2)        277           285
001          UTHASH(1:257)  pointer external
       82<84>D
001 000036S  VALID_SEGMENT  bit(1) aligned automatic
       130D              165M          199M          201M          206M
                         219
001          VDNUDG  entry constant shortcall external
       90<33>D
001          WAITA  entry constant shortcall returns(bin(15)) external
       90<33>D
001          WORD(1:1)  bin(15) based
       98<6>D
001 000075S  XCODE  bin(15) parameter
       70D               285A          293M          298M          301M
                         304M
001 000056S  XKEY  bin(15) parameter
       70D               164
001 000067S  XMAX_CHARS  bin(15) parameter
       70D               214A          261A          274           275
```

```
                    285(2)A         295

001 000000X  1 XP  based
      98<52>D
    000000+00   2 R_SN  bit(16)
      98<52>D
    000001+00   2 XREL  bin(15)
      98<52>D

001 000064S  XPATHNAME  char(128) parameter
      70D               214A            261A            275             285A
                        295M
001 000072S  XPATH_LEN  bin(15) parameter
      70D               214A            261A            273M            285A

001 000000X  1 XPB  based
      98<55>D
    000000+00   2 R_SN  bit(16)
      98<55>D
    000001+00   2 XRELB  bit(16)
      98<55>D

001 000061S  XUNIT  bin(15) parameter
      70D               171A            190             277(2)          280
                        285A
```