MiniMON29K™
User Interface
MONDFE

MiniMON29K™ User Interface: MONDFE, Release 3.0

Advanced Micro Devices, Inc.
5204 E. Ben White Blvd.
Austin, TX 78741

Contents

About MONDFE

Chapter 1

MONDFE Command-Line Syntax

Chapter 2

MONDFE Commands

Chapter 3

MONDFE Tutorial

Appendix A

Error Messages

Index

Figures and Tables

Figures

Tables

About MONDFE

The Advanced Micro Devices (AMD®) MiniMON29K™ user interface, **mondfe**, provides a simple user interface to develop, debug, and execute application programs on a 29K Family-based target. **mondfe** is *UDI-conformant*, which means that the back-end of **mondfe** conforms to AMD's *Universal Debugger Interface (UDI) Specification*.

mondfe provides an interactive command interface, which is designed for non-graphics terminals. **mondfe** interactive commands provide the user with the ability to download programs in AMD's Common Object File Format (COFF), set breakpoints, display and modify registers, assemble instructions, and trace program execution.

This chapter first describes the MONDFE documentation, then discusses the features and modules of the MONDFE software, and, finally, lists the MONDFE documentation conventions.

MONDFE Documentation

This documentation is written for programmers using **mondfe** to develop applications based on the 29K Family of microprocessors and microcontrollers. For more information on these microprocessors and microcontrollers, see the list of suggested reference materials that follows.

About This Manual

Chapter 1: “MONDFE Command-Line Syntax” describes how to invoke **mondfe**. The command-line syntax and descriptions of all the command-line options are discussed, followed by a description of the UDI configuration file and command-file format.

Chapter 2: “MONDFE Commands” describes each **mondfe** command in alphabetical order.

Chapter 3: “MONDFE Tutorial” explains how to use **mondfe** to debug an example program containing intentional errors. The tutorial presents a methodical approach to compiling, linking, and debugging programs prepared by the user.

Appendix A: “Error Messages” contains MONDFE and UDI error messages and descriptions.

“Index” provides an index to the MONDFE manual.

Suggested Reference Material

The following reference documents may be of use to the **mondfe** software user:

- *Am29000™ and Am29005™ User's Manual and Data Sheet*
  Advanced Micro Devices, order number 16914A.
- *Am29030™ and Am29035™ Microprocessors User's Manual and Data Sheet*
  Advanced Micro Devices, order number 15723B
- *Am29050™ Microprocessor User's Manual*
  Advanced Micro Devices, order number 14778A
- *Am29050™ Data Sheet*
  Advanced Micro Devices, order number 15039A.
- *Am29200™ RISC Microcontroller User's Manual and Data Sheet*
  Advanced Micro Devices, order number 16362B
- *Am29205™ RISC Microcontroller Data Sheet*
  Advanced Micro Devices, order number 17198A
- *Am29240™, Am29245™, and Am29243™ RISC Microcontrollers User's Manual and Data Sheet*
  Advanced Micro Devices, order number 17741A
- *High C® 29K™ User's Manual*
  Advanced Micro Devices
- *High C® 29K™ Reference Manual*
  Advanced Micro Devices
- *Host Interface (HIF) Specification, Version 2.0*
  Advanced Micro Devices, order number 11014B
- *MiniMON29K™ Target Interface Process: MONTIP*
  Advanced Micro Devices
- *Processor Initialization and Run-Time Services: OSBOOT*
  Advanced Micro Devices
- *Programming the 29K™ RISC Family*
  by Daniel Mann, P T R Prentice-Hall, Inc. 1994
- *RISC Design-Made-Easy Application Guide*
  Advanced Micro Devices, order number 16693A
- *Universal Debugger Interface (UDI) Specification, Version 1.2*
  Advanced Micro Devices

MONDFE Software

The features of the **mondfe** software are discussed below, followed by a description of the three modules of the program.

MONDFE Features

The MiniMON29K Debugger Front End, **mondfe**, provides non-symbolic debugging facilities to set breakpoints, display and modify registers, assemble instructions, and trace execution. It has a line-oriented user interface that is designed for a non-graphics terminal. **mondfe** downloads application programs (in their entirety or selected portions) in COFF file format onto the target.

The log file capability of **mondfe** can create a log file of all the commands of a debug session. This log file can be directly given as an input command file at invocation to execute the same sequence of commands. The echo file capability of **mondfe** can capture the screen displays into a file or capture a history of what was done during a particular debug session. The echo file created can be used for testing purposes or for comparing two automated debug sessions.

mondfe can be invoked in quiet mode using the **–q** option, which suppresses all descriptive messages. The program also can be invoked in either interactive or non-interactive mode with the **–D** command-line option. When invoked in non-interactive mode, an executable file must be specified on the command line. This executable file is downloaded onto the target and executed until completion.

mondfe provides a set of functions to display and modify memory and register contents. Commands do not support symbolic references to program variable names and addresses. The registers can be referenced using their standard mnemonics.

Commands in **mondfe** can display data in bytes, half-words, words, floating-point values, or in double words. An ASCII representation of the data displayed also is provided when the display format is in words. Commands in **mondfe** can write to memory in bytes, half-words, words, floating-point values, double words, or in character strings. The **FILL** (**F**) command fills memory with a string of characters.

mondfe can disassemble instructions in memory. Standard opcode mnemonics are used for disassembly.

MONDFE Modules

MONDFE is comprised of three modules: User Interface, MONDFE Command Interpreter, and MONDFE Command-to-UDI Procedural Call Converter. Figure 0–1 shows these modules.

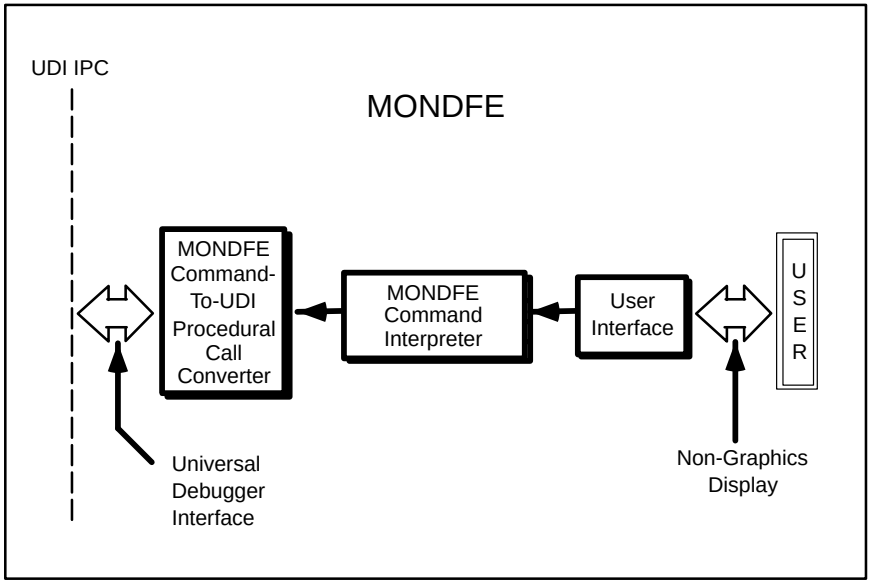

*Figure 0–1. MiniMON29K Debugger Front End, **mondfe***

User-Interface Module

The User-Interface Module is a non-symbolic user interface designed for non-graphics terminals. This module provides an interactive command interface, where it prompts the user to enter commands at the **MONDFE.EXE>** prompt. The commands are passed to the Command Interpreter Module. The status (error code) and results, if any, of the command executed are displayed on the terminal. The interactive command interface can be suppressed at invocation on the command line. This module also implements the control functions to perform terminal input and output operations of the application program running on the 29K Family-based target.

MONDFE Command Interpreter Module

The MONDFE Command Interpreter Module parses the input string entered by the user and checks for a valid **mondfe** command. For each user command, this module executes the appropriate command handler. When the command is completely executed, the module receives the next user command. The status (error code) and any results from the executed command are then returned to the User-Interface Module. The appendix of this manual describes the error messages that **mondfe** generates.

MONDFE Command-to-UDI Procedural Call Converter Module

The MONDFE Command-to-UDI Procedural Call Converter Module implements each of the command handlers and sub-handlers called by the MONDFE Command Interpreter Module. It initializes the UDI data structures from **mondfe** data structures and calls the appropriate UDI procedure. The actual implementation of the UDI procedure depends on the UDI interprocess communication (IPC) mechanism. Any results returned by the UDI procedure are converted and placed into **mondfe** data structures. Thus, this module allows **mondfe** to operate with any UDI-conformant target interface process. Figure 0–2 shows how **mondfe** can be used for debugging with the UDI-conformant TIP, **montip**.

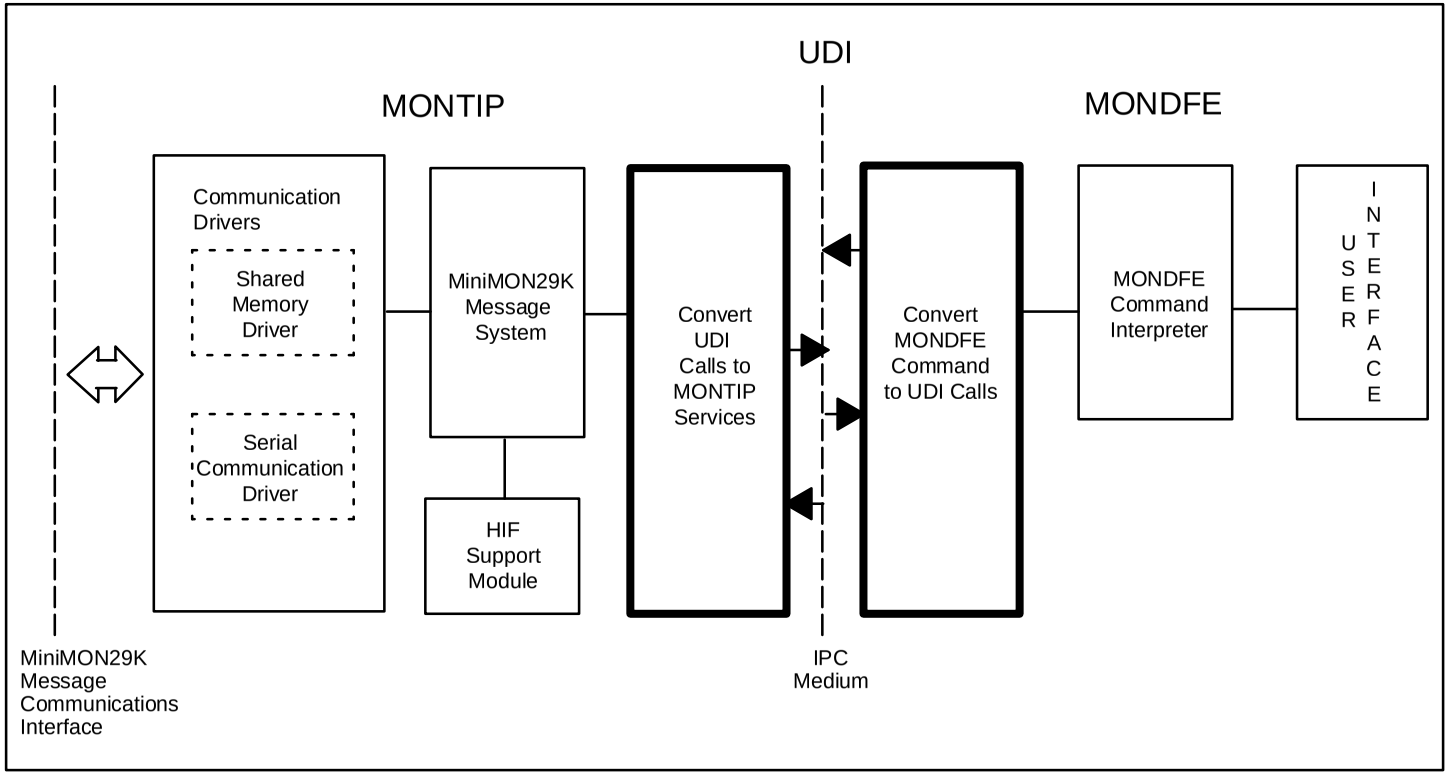

*Figure 0–2. MiniMON29K **mondfe** with UDI-Conformant TIP, **montip***

MONDFE Documentation Conventions

The Advanced Micro Devices manual *MiniMON29K User Interface: MONDFE* manual uses the conventions shown in the following table (unless otherwise noted). These same conventions are used in all the 29K Family support product manuals.

Table 0–1. Notational Conventions

Symbol	Usage
Boldface	Indicates that characters must be entered exactly as shown. The alphabetic case is significant only when indicated.
Italic	Indicates a descriptive term to be replaced with a user-specified term.
`Typewriter face`	Indicates computer text input or output in an example or listing.
[ ]	Encloses an optional argument. To include the information described within the brackets, type only the arguments, not the brackets themselves.
{ }	Encloses a required argument. To include the information described within the braces, type only the arguments, not the braces themselves.
..	Indicates an inclusive range.
...	Indicates that a term can be repeated.
|	Separates alternate choices in a list — only one of the choices can be entered.
:=	Indicates that the terms on either side of the sign are equivalent.

Chapter 1

MONDFE Command-Line Syntax

The syntax for using **mondfe** is described on the following pages, followed by a description of the UDI configuration files and the **mondfe** command-file format.

MONDFE Command-Line Syntax

Syntax: mondfe [–D] –TIP *tip_id_from_udi_config_file* [–q]
[–e *echo_filename_for_script*][–c *input_cmd_filename*]
[–ms *mem_stack_size_in_hex*]
[–rs *reg_stack_in_size_in_hex*] [–le]
[–log *logfile_name*] [–w *wait_time*]
[*pgm_name* [*pgm_args_list*] [<*input_data_file*]]

where:

–D
Specifies that an *interactive* debug session is requested. When this option is not specified, a program *must* be specified on the command line. The program is downloaded and executed. The default is *non-interactive*; that is, a program must be specified on the invocation line. Both **mondfe** and the TIP exit after completion of the program's execution.

–TIP *tip_id_from_udi_config_file*
Specifies the ID of the TIP defined in the UDI configuration file with which to establish connection for that debug session. (The UDI configuration file is **udiconfs.txt** on MS-DOS© systems, and **udi_soc** on UNIX© systems. See the "UDI Configuration Files" section on page 1–5 for more information.) This ID is matched with the first field of the line entries in the UDI configuration file. The TIP corresponding to the matched ID entry is started (if necessary) and a connection is requested. This option must be specified.

–q
Indicates that downloading messages must be suppressed (quiet mode). This is useful when a script of the session is being saved in an echo file. The default is non-quiet mode. The quiet mode can be turned on or off from the **mondfe** command prompt using **qon** and **qoff** commands.

–e *echo_filename_for_script*
Turns on the echo mode and specifies the name of the echo file in which to save the entire session that appears on the screen or terminal. This option scripts a debug session. There is no default filename. The echo mode can be turned on or off from the **mondfe** command prompt using the **EON** and **EOFF** commands.

–c *input_cmd_filename*

Specifies the name of the input command file. **mondfe** reads all its input (commands and program input) from this file until the end of file is reached. At that point, **mondfe** switches back to reading from standard input, usually the terminal. There is no default. The command file can be specified or changed at the **mondfe** command prompt using the **ZC** command. For more information on using a command file, see page 1–7.

–ms *mem_stack_size_in_hex*

Specifies the memory stack size to be used for the debug session. This information is passed to the TIP before a process begins execution. The default is 0x6000 bytes.

–rs *reg_stack_size_in_hex*

Specifies the register stack size to be used for the debug session. This information is passed to the TIP before a process begins execution. The default value is 0x2000 bytes.

–le Specifies that the target system is little endian. The default is big endian.

–log *logfile_name*

Turns on the log mode and specifies the name of the logfile. The logfile contains all **mondfe** commands entered by the user in that particular session. This logfile can be used as the input command file to reproduce a debug session. There is no default filename. Log mode can be turned on or off from the **mondfe** command prompt using the **LOGON** or **LOGOFF** command. The logfile name can be specified or changed using the **ZL** command.

–w *wait_time*

Specifies the length of time (in loop counts) that **mondfe** requires the TIP to wait when expecting a response from the target. The *wait_time* parameter is an integer value. The default value is 10. A value of minus 1 (–1) indicates wait forever.

pgm_name [*pgm_args_list*]] [<*input_data_file*]

Allows the user the option to specify the program to be executed and its command-line arguments when invoking **mondfe**. This program is downloaded onto the target system. If the **–D** option is *not* used, the program executes immediately after download. If a **–D** option *is* used, **mondfe** enters interactive mode and prompts the user for commands or reads from the input command file, if one was specified. **mondfe** then waits for a **G** command before executing the program. If the input data is redirected using **<input_data_file**, the program's request for input is satisfied by transmitting the contents of the specified input file. Otherwise, input from the keyboard is expected.

NOTE: This standard input redirection is a new feature and allows the separation of data input from the command file.

UDI Configuration Files

udiconfs.txt

This file is the UDI configuration file for MS-DOS systems. The UDICONF environment variable can be set to the complete pathname of this file if the file is not in the current working directory. When UDICONF is *not* set, the **udiconfs.txt** file *must* be present in the current working directory. The UDICONF variable is checked before using the file in the current directory. If the DFE does not find this file, an error is reported.

For more information about the UDI configuration and its related files, see the *Universal Debugger Interface (UDI) Specification, Version 1.2*.

udi_soc

This file is the UDI configuration file for UNIX systems. The UDICONF environment variable can be set to the complete pathname of this file, if it is not in the current working directory of the Debugger Front End (DFE). When UDICONF is *not* set, this file *must* be present in the current working directory. The UDICONF variable is checked before using the file in the current directory. If this file is not found, an error is reported.

Sample File Entries

The following set of sample entries in the UDI configuration file are used in the examples below.

MS-DOS Host (udiconfs.txt file entries)

```
eb29k_id montip.exe –t eb29k –r eb29k.os
isstip_id isstip.exe –r osboot
serial_38400 montip.exe –t serial –baud 38400
```

UNIX Host (udi_soc file entries)

```
isstip_id AF_UNIX sockiss isstip –r osboot
serial_38400 AF_UNIX sockser montip –t serial –baud 38400
```

Example

```
mondfe –D –TIP isstip_id
```

In the example above, **mondfe** is invoked in interactive mode (**–D** option) and **isstip_id** is used as the ID of the TIP. This ID is matched with the first field of the line entries specified in the UDI configuration file, which is **udi_soc** on UNIX hosts and **udiconfs.txt** on MS-DOS hosts.

Example

```
mondfe –D –TIP serial_38400 –ms 10000 –rs 3000
```

In the example above, **mondfe** is invoked in interactive mode and **serial_38400** is specified as the ID of the TIP. This example also shows the memory stack and the register stack sizes for that debug session specified using the **–ms** and the **–rs** options, respectively.

Example

```
mondfe –TIP eb29k_id hello.eb arg1 arg2
```

In this example, **mondfe** is invoked in non-interactive mode. The ID of the TIP is given as **eb29k_id** (for MS-DOS hosts only). Since **hello.eb** is an invalid option, **mondfe** interprets it as the name of the program to be executed. Anything following **hello.b** is interpreted as program arguments. Therefore, **hello.eb** is downloaded and executed until completion.

MONDFE Command-File Format

Command files contain sequences of the **mondfe** commands. The **–c** command-line option specifies a single input file, whose commands are to be executed immediately when **mondfe** is invoked.

Command files can improve efficiency when debugging a program. For example, a command file can load and execute a program automatically when **mondfe** is invoked. An example of the commands this file might contain is shown below:

```
Y program.out
B 10020i
G
```

These commands (one per line) load the absolute COFF file **program.out** into memory, set a breakpoint at location 10020i, and begin execution.

If these commands are stored in a file **startup.cmd,** the command line for invoking **mondfe** would look like the following:

```
C:> mondfe –D –TIP eb29k_id –c startup.cmd
```

All the housekeeping chores associated with loading the program, setting breakpoints, and displaying initial program or data information can be written to a command file.

Chapter 2

MONDFE Commands

This chapter provides a detailed description of each **mondfe** command, including syntax, options, specific information on how to use each command, and examples of command usage.

All commands are executed that are entered on the host computer's keyboard at the **mondfe** prompt (**MONDFE.EXE>**) or read from a command input file specified on the command line at **mondfe** invocation are executed. The results, if any, are displayed on the host computer's screen. Commands are case-insensitive and can be entered in either upper or lower case. For example, **BC** and **bc** are identical commands for clearing one or more breakpoints. In this manual, the command names appear in upper case for emphasis only.

Command List

The commands, their abbreviations, and the pages on which they can be found are listed below.

29K Family Memory Address Format: *29K_MEM_ADDR*

The 29K Family memory address format,*29K_MEM_ADDR*, recognized by **mondfe** commands is as follows:

address_value[*space_id_suffix*]

where:

address_value Is a numerical value interpreted as a hexadecimal number.

space_id_suffix Is an optional character suffix immediately following *address_value* with no spaces in between. The suffix character can be one of the following:

- **i** Instruction RAM space
- **r** Instruction ROM space
- **p** I/O port address
- **u** Generic memory space, and to access 29K Family microcontroller peripherals
- **m** Data RAM space

All commands interpret the specified 29K Family memory address value as a hexadecimal number. When an address space suffix is not given, a default memory space is assumed. The default memory space differs among **mondfe** commands. For example, memory space defaults to the Instruction RAM space for the **DISASSEMBLE** (**L**) command, whereas it defaults to Data RAM space for the memory **DISPLAY** (**D**) command.

29K Family Register Name Format: *29K_REG_NAME*

The 29K Family register name format, *29K_REG_NAME*, recognized by **mondfe** commands is as follows:

LR*nnn* | AR*nnn* | GR*nnn* | SR*nnn* | TR*nnn* | XR*nnn*

where:

nnn	Is a decimal number within the valid range (0 to 255).
LR*nnn*	Refers to processor's LOCAL register number *nnn*. Valid entries are **lr0–lr127**.
AR*nnn*	Refers to processor's ABSOLUTE register number *nnn*. Valid entries are **ar0–ar255**.
GR*nnn*	Refers to processor's GLOBAL register number *nnn*. Valid entries are **gr0–gr1**, **gr64–gr127**.
SR*nnn*	Refers to processor's SPECIAL register number *nnn*. Valid entries are: **sr0–sr25**, **sr128–sr135**.
TR*nnn*	Refers to processor's TLB register number *nnn*. Valid entries are **tr0–tr127**.
XR*nnn*	Refers to COPROCESSOR register number *nnn*. For example: **xr0**.

Standard mnemonics also can be used to specify *special* register names.

A — Assemble Instruction

Syntax: A *29K_MEM_ADDR mnemonic* [*para1* [*para2...*]]
or
A *29K_MEM_ADDR*

where:

29K_MEM_ADDR
Specifies the hexadecimal memory address value and the associated memory space suffix as described earlier under the *29K_MEM_ADDR* format. If no suffix is given, the memory space assumed is instruction RAM.

mnemonic Specifies a legal 29K Family of microprocessors alphabetic instruction mnemonic.

para1, para2, etc.
Are the parameters required by the specific microprocessor instruction being assembled.

The **ASSEMBLE** (**A**) command allows the user to assemble instructions in memory, using the standard instruction mnemonics of the AMD 29K Family of RISC processors. Each **ASSEMBLE (A)** command assembles one instruction in memory.

The parameters (*para1*, *para2*, etc.) to the instructions and their numbers depend on the type of instruction being assembled. For example, a **NOP** instruction does not require any parameters, while a **LOAD** instruction requires four parameters. Parameters can be separated with commas or spaces. However, parameters cannot be separated with semicolons. The semicolon (;), which is the assembler comment character, is not recognized and it produces an illegal syntax error.

Multiple instructions can also be assembled by invoking the assembler in interactive mode. When **A** *29K_MEM_ADDR* is entered, the assembler displays the address and prompts for the instruction to be assembled. It continues to prompt for instructions until a period (**.**) is entered. Relative jumps can be assembled by specifying the target as an offset from the address where the instruction is being assembled. For example:

```
jmp     .+4
```

where **.** (period) refers to the address where the instruction is being assembled.

Example

```
MONDFE.EXE> A 10004 SUB GR96 GR96 1
MONDFE.EXE> L 10004 10004
00010004 25606001 sub      gr96,gr96,0x1
```

In the example above, an instruction to subtract constant 1 from global register **gr96** has been assembled into instruction RAM at location 0x10004. The **ASSEMBLE** (**A**) command does not produce any output, but the **DISASSEMBLE** (**L**) command in the example shows the assembled instruction at location 0x10004.

B — Breakpoint Command

Syntax:

B	[*29K_MEM_ADDR* [*pass_count*]]
B050	*29K_MEM_ADDR* [*pass_count*]
BC	[*29K_MEM_ADDR*]

where:

B — Sets or displays breakpoints.

B050 — Sets a breakpoint using the Am29050 Instruction Breakpoint Control Register.

BC — Clears a specific breakpoint or all breakpoints.

29K_MEM_ADDR — Specifies the hexadecimal address value and the associated memory space suffix where the breakpoint is to be set or cleared. Instruction RAM is assumed when no suffix is used.

pass_count — Specifies the number of times minus 1 that the instruction at the breakpoint address should be executed before honoring the breakpoint for the first time. The *pass_count* parameter is interpreted as a decimal value. The *pass_count* value can be positive or negative. Positive values define *sticky* breakpoints, and negative values define *non-sticky* breakpoints. Non-sticky breakpoints are immediately removed when they are hit. Sticky breakpoints remain set even after they are hit. If no *pass_coun*t is given, a default value of 1 is used.

To accomplish the following breakpoint tasks, use the indicated **BREAKPOINT** command:

- Set a breakpoint

To set a breakpoint, use the **B** command and specify the address at which to set the breakpoint. Specify any *pass_count* as required. Sticky breakpoints are set by using a positive, non-zero *pass_count* value. Non-sticky breakpoints are set by using a negative *pass_count* value.

- Clear a breakpoint

To clear a breakpoint, use the **BC** command and specify the address at which the breakpoint is to be cleared.

- Display all valid breakpoints

To display all valid breakpoints, use the **B** command without any parameters. All the current breakpoints will be displayed.

- Clear all breakpoints

To clear *all* breakpoints, use the **BC** command without any parameters.

- Set a breakpoint using the Am29050 Instruction Breakpoint Control register

Using the **B050** command instead of the **B** command forces the MiniMON29K debugger core to use an Am29050 processor's Instruction Breakpoint Control Register to implement the breakpoint. If none of these registers are available, the breakpoint is not set and an error is returned. This command is recognized only by the MiniMON29K target interface process, **montip**. The **B050** command can be used to set breakpoints in ROM, where the ROM space is not writable. This is useful in debugging freeze mode code.

Example

```
MONDFE.EXE>B 10004I
MONDFE.EXE>G
```

In the example above, a breakpoint is set at address 0x10004, in instruction RAM. No output is produced by executing this command; however, upon execution of the following **GO** (**G**) command, the output that follows is displayed when the program stops at the breakpoint:

```
MONDFE.EXE>Breakpoint hit at 00010004.
00010004  25010118  sub  gr1,gr1,0x18
```

Notice that the location of the breakpoint hit, the hexadecimal value of the instruction word at that location, and its disassembled value are displayed. Further note that the instruction at the hit breakpoint has not been executed. Resumption of the program by a **TRACE** (**T**) or **GO** (**G**) command resumes with this instruction unless the PC registers have been changed.

C — Display Target Configuration

Syntax: C

The **C** command displays the 29K Family-based target's configuration, including the following:

- Processor type
- Whether a coprocessor is present in the system
- Instruction memory range
- Data memory range
- Instruction ROM range
- Version of the MiniMON29K Debugger Front End (**mondfe**) being used

Example

```
MONDFE.EXE> C

      MINIMON29K 3.0 User Front End
    Copyright 1993 Advanced Micro Devices, Inc.
        Version 3.0 – 0 (07/15/93)
    Processor type:          Am29000 (revision D)
    Coprocessor:             None
    ROM range:               0x0 to 0x7ffff (512K)
     Instruction memory range:  0x0 to 0x7ffff (512K)
    Data memory range:       0x0 to 0x7ffff (512K)


      (Enter 'h' or '?' for help)
```

CAPS — Display DFE and TIP Capabilities

Syntax: CAPS

The **CAPS** command displays the DFE version number, TIP version number, and UDI version.

CH0 — Transfer Terminal Control to Target

Syntax: CH0

The **CH0** transfers control of the terminal to the 29K Family target program. The characters entered by the user are sent to the underlying TIP for processing. Control is transferred to **mondfe** when **Ctrl–U** is entered.

CON — Connect to UDI Session

Syntax: CON *session_id*

where:

session_id Is a UDI session number.

The **CON** command requests connection to the UDI TIP corresponding to the UDI debug session referred to by *session_id*.

CP — Create UDI Process

Syntax: CP

The **CP** command sends a request to the TIP to create a new UDI process.

D — Display Memory/Registers

Syntax:

D [W | H | B | F | D][*starting_resource* [*ending_resource*]]

where:

W	Formats the display in words (default).
H	Formats the display in half-words.
B	Formats the display in bytes.
F	Formats the display in floating-point values.
D	Formats the display in double-precision floating-point format.
starting_resource	Specifies the starting location of the data to be displayed. This can be either a memory address specified in *29K_MEM_ADDR* format, or a register name specified in the *29K_REG_NAME* format. When a *starting_resource* is not specified, the display begins at the *ending_resource* value of the previously executed **DISPLAY** (**D**) command. The *29K_MEM_ADDR* and the *29K_REG_NAME* formats are described on pages 2–4 and 2–5. The default memory space is data memory.
ending_resource	Specifies the ending location of the data to be displayed. This can be either a memory address specified in *29K_MEM_ADDR* format, or a register name specified in *29K_REG_NAME* format. If an *ending_resource* is not specified, a default value is assumed depending on the type of the resource accessed (memory or registers). The *29K_MEM_ADDR* and the *29K_REG_NAME* formats are described on pages 2–4 and 2–5.

The **DISPLAY** (**D**) command displays 29K Family target memory contents or register values. The *starting_resource* and *ending_resource* together define the range of data to be displayed. The display format is specified by the suffix character of the **DISPLAY** (**D**) command. All the format suffixes above can be used to display either memory contents or register contents. When no suffix is given, that is, when **D** is used alone, the display format defaults to words.

The ASCII representation of the data also is displayed when the display format is either words (**D** or **DW**), half-words (**DH**), or bytes **(DB**).

When an *ending_resource* value is not specified, a default range of values applicable to the resource type (memory or registers) is used. The range between the *starting_resource* and the *ending_resource* must belong to the same resource type when specified.

Display Formats

In the display formats shown below, the word, half-word, and byte displays show the addresses and data in hexadecimal format. The single-precision and double-precision displays show the data in floating-point format. The data also is shown in ASCII format if the characters are printable ASCII characters.

Word

```
MONDFE.EXE>D 10000 1000F
00010000 7261643d 256c662c 2073696e 28726164 rad=%lf sin(rad
```

Half-Word

```
MONDFE.EXE>DH 10000 1000F
00010000 7261 643d 256c 662e 2073 696e 2872 6164 rad=%lf sin(rad
```

Byte

```
MONDFE.EXE>DB 10000 1000F
00010000 72 61 64 3d 25 6c 66 2e 20 73 69 6e 28 72 61 64 rad=%lf sin(rad
00010000 72 61 64 3d 25 6c 66 2e 20 73 69 6e 28 72 61 64 rad=%lf sin(rad
```

Single-Precision Floating-Point

```
MONDFE.EXE>DF 14000 1400F
00014000 +1.147944e-040 +1.148224e-040 +6.157305e-041 +6.166834e-041
```

Double-Precision Floating-Point

```
MONDFE.EXE>DD 14000 1400F
00014000 +9.338479079051290e-310 +9.338479079054255e-310
```

Example

```
MONDFE.EXE>D gr96 gr99
gr096 0000a000 0000e5bc 00010000 000163ab .........c.
```

In the example above, the contents of global registers **gr96** through **gr99** are dumped in word format.

Example

```
MONDFE.EXE>D lr0 lr3
lr000 000010d0 0007dfb0 00000000 0007e000 ..............
```

In the example above, the contents of local registers **lr0** through **lr3** are dumped in word format.

DISC — Disconnect from UDI Session

Syntax: DISC

The **DISC** command temporarily disconnects the Debugger Front End (DFE) from the current debug session. The TIP is not destroyed but continues running for later reconnections.

DP — Destroy UDI Process

Syntax: DP

The **DP** command destroys a UDI process previously created using the **CP** command.

EOFF/EON — Turn Off/On Echo Mode

Syntax: EOFF
EON

The **EOFF/EON** commands turn echo mode off/on during the interactive debug session. Echo mode is specified by the **–e** command-line option and a filename. During echo mode, everything displayed on the screen is captured in the file specified.

ESC — Escape to Host Operating System

Syntax: ESC

The **ESC** command temporarily exits **mondfe** to the host operating system. The user can return to the **mondfe** session by using the **EXIT** command at the host operating-system prompt.

Example

```
MONDFE.EXE>esc
Microsoft(R) MS-DOS(R)  Version 3.30
            (C)Copyright Microsoft Corp 1981–1987


C:\>exit
MONDFE.EXE>
```

In the example above, the **ESC** command temporarily exits the program to MS-DOS, which then returns the **C:\>** prompt. Typing **exit** at that prompt takes us back to the debug session, where the **MONDFE.EXE>** prompt returns.

EX — Exit UDI Session

Syntax: EX

The **EX** command causes **mondfe** to exit from a debug session when that session is complete. **mondfe** looks for another debug session in progress and connects to that session. If no more debug sessions are in progress, this command causes **mondfe** to quit (i.e., it has the same effect as the **Quit** command).

F — Fill Memory/Registers

Syntax:

F [W | H | B | F | D |S] *starting_resource ending_resource fill_data*

where:

W — Specifies *fill_data* format is a 32-bit integer (default).

H — Specifies *fill_data* format is a 16-bit integer.

B — Specifies *fill_data* format is an 8-bit integer.

F — Specifies *fill_data* format is a floating-point number.

D — Specifies *fill_data* format is a double-precision floating-point number.

S — Specifies *fill_data* is a string of characters.

starting_resource
Specifies the starting location of the fill. This location can be either a memory address specified in *29K_MEM_ADDR* format, or a register name specified in the *29K_REG_NAME* format. The *29K_MEM_ADDR* and the *29K_REG_NAME* formats are described on pages 2–4 and 2–5. The default memory space is data memory.

ending_resource
Specifies the ending location of the fill. This location can be either a memory address specified in *29K_MEM_ADDR* format, or a register name specified in the *29K_REG_NAME* format. The *29K_MEM_ADDR* and the *29K_REG_NAME* formats are described on pages 2–4 and 2–5.

fill_data — Specifies the data that is to be replicated into the range specified by the *starting_resource* and *ending_resource* parameters. The *fill_data* must be a valid data consistent with the fill format specified. It is interpreted as a hexadecimal number, except when the **S** format is specified.

The **FILL** (**F**) command can be used to fill 29K Family target memory or registers. The format of the *fill_data* is specified by the suffix character of the **FILL (F)** command.

The *fill_data* parameter is interpreted as a hexadecimal value and must conform to the size of the data elements specified by the optional suffix (data format letter) following the **F** command, except in the case of **FS,** when *fill_data* is interpreted as a character string. If the data format-letter is missing, a default size of 32 bits (word) is assumed. All the format specifiers, except the **S** format, can be used to fill either memory or register.

The format of the *fill_data* specified must be consistent with and representable in the format specified. For example, a floating-point data must be supplied for an **FF** command, whereas an integer that can be represented in 32-bits must be supplied when **FW** is used. Thus, "abcdef" would be an invalid *fill_data* for an **FB** command. When no suffix is given, that is, when **F** is used alone, the *fill_data* is assumed to be representable as a 32-bit integer. When the resource type is register, the format specified should be either **W**, **F**, or **D**, because the registers are accessed as 32-bit quantities.

Fill Formats

In the fill formats shown below (word, half-word, byte, floating-point, and double-precision), formats interpret the fill data as hexadecimal values. The string format takes a string of characters as fill data. The **DISPLAY** (**D**) command following each fill command in the examples is shown only to illustrate the fill command results.

Word (Default)

```
MONDFE.EXE>f 80005000 80005010 12345678
MONDFE.EXE>d 80005000 80005010

80005000  12345678 12345678 12345678 12345678 _4Vx. 4Vx. 4Vx. 4Vx.
80005010  12345678                            _4Vx.
```

Half-Word

```
MONDFE.EXE>fh 80005000 80005010 abcd
MONDFE.EXE>d 80005000 80005010

80005000  abcdabcd abcdabcd abcdabcd abcdabcd  ........
80005010  abcd5678                             ..Vx
```

Byte

```
MONDFE.EXE>fb 80005000 80005010 a0
MONDFE.EXE>db 80005000 80005010

80005000   a0 a0 a0 a0 a0 a0 a0 a0 a0 a0 a0 a0 a0 a0 a0     a0.....
....
80005010   a0                                                .
```

Single-Precision Floating-Point

```
MONDFE.EXE>ff 80005000 80005010 1.2
MONDFE.EXE>df 80005000 80005010

80005000  +1.200000e+000 +1.200000e+000 +1.200000e+000 +1.200000e+000
80005010  +1.200000e+000
```

Double-Precision Floating-Point

```
MONDFE.EXE>fd 80005000 80005010 2.3
MONDFE.EXE>dd 80005000 80005010

80005000  +2.300000000000000e+000 +2.300000000000000e+000
80005010  +2.300000000000000e+000
```

String of Characters

```
MONDFE.EXE>fs 80005000 80005010 minimon
MONDFE.EXE>d 80005000 80005010

80005000  6d696e69 6d6f6e6d 696e696d 6f6e6d69 minimonminimonmi
80005010  6e696d6f                            nimo
```

Example

```
MONDFE.EXE>F 103CI 104FI 0
```

In the example above, the **FILL** command is instructed to fill instruction RAM locations 0x103c through 0x104f with zeros. Instruction RAM is indicated in the supplied addresses because the modifier suffix **I** has been appended to the addresses. If the **FILL** command was to be directed to instruction ROM, an **R** would be appended to each of the above addresses (for example, 103CR). Data RAM addresses are entered without a suffix or with an **m** suffix.

Example

```
MONDFE.EXE>FB 14000 140FF 20
```

In the example above, data memory locations 0x14000 through 0x140FF are filled with byte-size values. The value **20** is the hexadecimal code for an ASCII space character; therefore, 256 byte locations, beginning at address 0x14000, are filled with spaces.

G — Go

Syntax: G

The **GO** (**G**) command starts or resumes execution of a previously downloaded program in target memory. (Refer to the **YANK** (**Y**) command description for downloading a program.) The execution always starts or resumes at the current location contained in the 29K Family Program Counter registers (**PC0, PC1**), unless they are changed explicitly by the user before issuing this command.

The first **G** command after downloading a program (using the **YANK** (**Y**) command) causes the execution of the downloaded program.

To restart a program without downloading it again, issue the **INIT** command first, and then issue a **GO** command. The **INIT** command initializes the process to the beginning (entry point).

When using **montip**, use the **–R** option of **montip** in the UDI configuration file to run programs in real/physical mode. To run programs in protected mode (one-to-one mapping of physical addresses to virtual addresses), use the **–P** option of **montip**.

To run the application program in protected mode, the application program must be linked in alignment with the page size used by the target kernel. For example, use the **ALIGN=8192** linker directive to align the sections on an 8K-page boundary.

Example

```
MONDFE.EXE> G
```

This example starts execution from the current location given by the processor's PC registers.

When **mondfe** is used with **montip**, application programs loaded into memory using the **YANK (Y)** command are executed in the mode according to the **–R** or **–P** option given to **montip** by default when the **G** command is issued.

H — Help

Syntax: H [*command*]
? [*command*]

where:

command Is the first letter corresponding to a **mondfe** command or a **mondfe** command name.

The **HELP** (**H** or **?**) command requests on-line help for any of the **mondfe** commands, or displays a brief description of each command's syntax and usage. On-line help is available for all **mondfe** commands. Simply typing **H** or **?** without the optional *command* parameter causes a help screen containing a list of **mondfe** commands to be displayed.

If the optional *command* parameter is given, it must be the first letter of one of the **mondfe** commands or a **mondfe** command name. If no command exists for that letter, the help screen is displayed.

Example

```
MONDFE.EXE>H
Use 'h <letter>' for individual command help.
```

```
 a - ASSEMBLE INSTRUCTION               | b,b050,bc - SET/CLEAR/DISPLAY BREAKPOINT
 c - PRINT CONFIGURATION                | caps - DFE AND TIP CAPABILITY
 cp - CREATE UDI PROCESS                | con - CONNECT TO A UDI DEBUG SESSION
 ch0 - 29K TERMINAL CONTROL             | d,dw,dh,db,df,dd - DUMP MEMORY/REGISTERS
 dp - DESTROY UDI PROCESS               | disc - TEMPORARILY DISOCNNECT UDI SESSION
 ex - EXIT UDI PROCESS                  | esc - ESCAPE TO HOST OPERATING SYSTEM
 eon - TURN ON ECHO MODE                | eoff - TURN OFF ECHO MODE
 g - START/RESUME EXECUTION             | f,fw,fh,ff,fd,fs - FILL MEMORY/REGISTERS
 h - HELP COMMAND                       | init - INITIALIZE CURRENT UDI PROCESS
 ix,il - DISPLAY AM2903X CACHE          | k - KILL RUNNING PROGRAM ON 29K TARGET
 logon - TURN ON LOG MODE               | logoff - TURN OFF LOG MODE
 l - LIST/DISASSEMBLE MEMORY            | m - MOVE DATA TO MEMORY/REGISTERS
 pid - SET UDI PROCESS ID               | q - QUIT MONDFE
 qon - TURN QUIET MODE ON               | qoff - TURN QUIET MODE OFF
 sid - SET UDI SESSION ID               | r - RESET (SOFTWARE RESET) 29K TARGET
 t - TRACE/SINGLE STEP EXECUTION        | s,sw,sh,sb,sf,sd - SET MEMORY/REGISTERS
 ver - MONTIP VERSION COMMAND           | tip - MONTIP TRANSPARENT MODE COMMAND
 y - YANK/DOWNLOAD COFF FILE            | xp - DISPLAY PROTECTED MODE COMMAND
 ze - ECHO FILE FOR ECHO MODE           | zc - EXECUTE COMMANDS FROM COMMAND FILE
 zl - USE LOG FILE FOR LOG MODE         | | - COMMAND CHARACTER (IN COMMAND FILE)
```

```
MONDFE.EXE>
```

Example

```
MONDFE.EXE>h f
```

The above command causes **mondfe** to output a screen containing information on the fill command, as shown below:

```
F[W|H|B|F|D] <start address>, <end address>, <value>

Fill memory or register contents.

FW or F-fill as 32-bit integers| FF – fill as floating point value.
FH  – fill as 16-bit integers  | FD – fill as double precision
FB   – fill as 8-bit integers  | floating point value.
FS   – fill with the string/pattern given.

Register names:

GR01, GR64–GR127         SR00–SR14, SR128–SR135
LR00–LR127               TR00–TR127

Memory addresses:

<hex>m – data memory <hex>i – instruction memory
<hex>r – rom memory  <hex>u – unspecified (no addr check)
MONDFE.EXE>
```

IL — Disassemble Am2903x Processor Cache

Syntax: IL

The **IL** command disassembles the contents of the Am2903x processor cache registers.

Example

```
MONDFE.EXE>il

Cache line 0x1, set 105.
IATAG  V  P US
00004  0  1  1

00004010    00004003    .word    0x00004003
00004014    ce000b60    mtsr     pc1,gr96
00004018    1600607d    load     0,0x0,gr96,gr125
0000401c    157d7d04    add      gr125,gr125,0x4

Cache line 0x10001, set 105.

IATAG  V  P US
ce000  0  0  0

00000a10    00001000    .word    0x00001000
00000a14    fbfbfff7    .word    0xfbfbfff7
00000a18    dfbff7b7    multmu   lr63,lr119,lr55
00000a1c    feefdf7f    .word    0xfeefdf7f
```

INIT — Initialize Downloaded Program

Syntax: INIT

The **INIT** command requests the underlying TIP to initialize the downloaded program. This command restarts a program without downloading it again. An **INIT** command, followed by a **GO** command, restarts the downloaded application program.

IX — Display Am2903x Processor Cache

Syntax: IX

The **IX** command displays the contents of the Am2903x processor cache registers.

Example

```
MONDFE.EXE>ix

Cache line 0x1, set 105.

IATAG  V  P US
00004  0  1  1

00004010    00004003    .word    0x00004003
00004014    ce000b60    mtsr     pc1,gr96
00004018    1600607d    load     0,0x0,gr96,gr125
0000401c    157d7d04    add      gr125,gr125,0x4

Cache line 0x10001, set 105.

IATAG  V  P US
ce000  0  0  0

00000a10    00001000    .word    0x00001000
00000a14    fbfbfff7    .word    0xfbfbfff7
00000a18    dfbff7b7    multmu   lr63,lr119,lr55
00000a1c    feefdf7f    .word    0xfeefdf7f
```

K — Kill Program Execution

Syntax: K

The **KILL** (**K**) command halts (kills) program execution. The **KILL** command causes **mondfe** to call the **UDIStop()** procedure. This procedure is transmitted to the underlying TIP, which processes the request. Program execution can be resumed by issuing a **GO** (**G**) command.

L — List (Disassemble) From Memory

Syntax: L [*29K_MEM_ADDR*$_i$ [*29K_MEM_ADDR*$_f$]]

where:

29K_MEM_ADDR$_i$

Specifies the initial address: the hexadecimal value (and the associated memory space suffix of the memory address in instruction ROM or RAM) at which to begin disassembling instructions. If no suffix is given, instruction RAM memory is assumed for disassembly. The format for *29K_MEM_ADDR* is described on page 2–4.

29K_MEM_ADDR$_f$

Specifies the final address: the hexadecimal value (and the associated memory space suffix of the memory address in instruction ROM or RAM) of the last instruction to be disassembled. If no suffix is given, the memory space of the starting location is assumed. This parameter is optional. The format for *29K_MEM_ADDR* is described on page 2–4.

The **LIST** (**L**) command lists (disassembles) portions of a program stored in the instruction memory of the target. Instruction opcodes are listed using their assembly language mnemonics, and operands are listed in a format appropriate to their individual types.

Instructions are disassembled beginning at the *29K_MEM_ADDR*$_i$ and ending with the *29K_MEM_ADDR*$_f$, inclusive when specified. If no *29K_MEM_ADDR*$_i$ is given, the *29K_MEM_ADDR*$_f$ also must not be specified.

Example

```
MONDFE.EXE>L 10000I 10014I
00010000 00000000 .word    0x00000000
00010004 25010118 sub      gr1.gr1,0x18
00010008 5e40017e asgeu    64,gr1,gr126
0001000c 15810118 add      lr1,gr1,0x18
00010010 036162e8 const    gr98,0x61e8
00010014 02006201 consth   gr98,0x1
```

The example above illustrates the disassembly of five instruction words, beginning at location 0x10000I, and ending at word location 0x10014I (20 bytes). Each line displays the location of the instruction, its contents in hexadecimal, and its mnemonic translation.

LOGOFF/LOGON — Turn Off/On Log Mode

Syntax: LOGOFF
LOGON

The **LOGOFF/LOGON** commands turn off/on the log mode from the **mondfe** command prompt. When log mode is on, every command entered by the user is logged into the log file specified either at invocation or by using the **ZL** command. When log mode is off, commands are not logged.

M — Move Registers/Memory

Syntax: M *source_start source_end destination_start*

where:

source_start Specifies the memory or register address from which the first data item is to be moved. A memory address must be specified using the *29K_MEM_ADDR* format explained on page 2–4. A register name must be specified using the *29K_REG_NAME* format explained on page 2–5. The default memory space is data memory.

source_end Specifies the memory or register address from which the last data item is to be moved. A memory address must be specified using the *29K_MEM_ADDR* format explained on page 2–4. A register name must be specified using the *29K_REG_NAME* format explained on page 2–5.

destination_start Specifies the memory or register address to which the first data item is to be moved. A memory address must be specified using the *29K_MEM_ADDR* format explained on page 2–4. A register name must be specified using the *29K_REG_NAME* format explained on page 2–5.

The **MOVE** (**M**) command moves data between registers or memory locations. The *source_start*, *source_end*, and *destination_start* parameters are all required. The source and destination parameters can be either register names or memory addresses, or a combination of both. However, the *source_start* and the *source_end* parameters cannot be a mixture of register names and memory addresses.

If the source parameters describe a series of memory locations that do not encompass a whole number of words, only the inclusive range of bytes is moved to the destination address. If a single register or memory location is to be moved, *source_start* and *source_end* must be identical.

Example

```
MONDFE.EXE> M 10000,1000f,20000
```

This example illustrates the use of the **M** command to move a block of memory data from one location to another. In this case, 16 bytes (four instructions) are moved from the data memory locations 0x10000–0x1000f to locations 0x20000–0x2000f.

PID — Set UDI Process ID

Syntax: PID *pid_number*

where:

pid_number Specifies the UDI process ID number.

The **PID** command sets the current UDI process to the *pid_number* specified. A *pid_number* of –1 represents the target system and can be used to access physical addresses and to reset the target. (The **CP** command creates a process, the **DP** command destroys a process, and the **INIT** command initializes a process.)

Q — Quit

Syntax: Q

The **QUIT** (**Q**) command terminates the debug session and exits the MiniMON29K Debugger Front End (**mondfe**). Any open log file or command files are closed before returning control to the host operating system.

If more than one debug session is in progress, with one or more TIPs, the **QUIT (Q)** command terminates all debug sessions. The associated TIP(s) become inactive.

Use the **EXIT (EX)** command to terminate only the current debug session.

Use the **DISCONNECT (DISC)** session command to temporarily exit from one debug session to another. At least one other debug session must be in progress when using this command.

QOFF/QON — Turn Off/On Quiet Mode

Syntax: QOFF
QON

The **QOFF/QON** command turns off/on quiet mode of **mondfe**. The **–q** command line option invokes **mondfe** in quiet mode. In quiet mode, debug messages are suppressed. These messages can be turned on anytime during the debug session using the **QON** command and turned off using the **QOFF** command.

R — Reset Target Processor

Syntax: R

The **RESET** (**R**) command performs a software reset of the target, resulting in the starting up of the target.

The **R** command requires a responsive target. When the target is hung or has crashed, a hardware reset (power cycle) may be required.

S — Set Memory/Registers

Syntax: S [W | H | B | F | D] *target_resource set_data*

where:

SW or S	Specifies *set_data* format is a 32-bit integer (default).
SH	Specifies *set_data* format is a 16-bit integer.
SB	Specifies *set_data* format is an 8-bit integer.
SF	Specifies *set_data* format is a floating-point number.
SD	Specifies *set_data* format is a double-precision floating-point number.
target_resource	Specifies a memory location or a register whose content is to be set to the *set_data* value given. A memory location or a register must be specified using the *29K_MEM_ADDR* format or *29K_REG_NAME* format described on pages 2–4 and 2–5. The default memory space is data memory.
set_data	Specifies the value to be written into the memory location or register specified by the first parameter. This value is interpreted as a hexadecimal value and according to the format suffix specified, if any. Its size must be consistent with the unit size of that format.

W, **H**, **B**, **F**, and **D** suffixes to the **SET (S)** command specify the format of *set_data*. When no suffix is given, **W** is assumed.

The **SET** (**S**) command sets a memory location or a register to a particular value specified by *set_data*. The format of the *set_data* is specified by the suffix character of the **S** command.

All the format specifiers above can be used to set either a memory location or a register. The *set_data* specified is interpreted as a hexadecimal value. The format of *set_data* must be consistent with and representable in the format specified. For example, a floating-point data must be supplied for an **SF** command, whereas an integer that can be represented in 32 bits must be supplied when **SW** is used. Thus, “abcdef” would be an invalid *set_data* for an **SB** command. When no suffix is given, that is, when **S** is used alone, *set_data* is assumed to be representable as a 32-bit integer.

The format specified should be either **W**, **F**, or **D**, when the resource type is register, because registers are accessed as 32-bit quantities.

Example

```
MONDFE.EXE>S 12000 03FC
```

In the example above, data memory locations 0x12000 through 0x12003 are set to the hexadecimal word value 0x000003FC. Because a modifier character was not given, 4 bytes are stored into the destination location.

Example

```
MONDFE.EXE>SB 12002 20
```

The example above sets the byte at data memory location 0x12002 to 0x20, which is an ASCII space character. The contents of the remainder of the data memory word remain unchanged by this substitution.

Example

```
MONDFE.EXE>SD GR96 3.1415926535
```

In the example above, global registers **gr96** and **gr97** are set to a double-precision floating-point value. When **SD** is used, an 8-byte value (double) is always stored. This *stored value* will occupy two consecutive memory or register locations.

Example

```
MONDFE.EXE>S 10020I 0300A165
```

This example illustrates using the **SET** (**S**) command to patch an instruction memory location to contain a different instruction. However, the **ASSEMBLE** (**A**) command is preferable for this purpose. See the description of the **ASSEMBLE** (**A**) command for more details.

SID — Set UDI Session ID

Syntax: SID *sid_number*

where:

sid_number Specifies the UDI session ID.

The **SID** command sets the UDI session ID to *sid_number.* This command can be used to set the current debug session when multiple debug sessions are occurring.

T — Trace (Single/Multiple Step) Execution

Syntax: T [*count*]

where:

count Specifies the number of instructions to trace. The *count* value is specified in decimal notation. The default is 1.

The **TRACE** (**T**) command traces or steps through the execution of a program. Either single or multiple instructions can be executed by using the optional *count* parameter.

If no optional *count* parameter is given, one instruction is executed from the current location.

After the completion of the **TRACE** (**T**) command, the instruction at the stopped location is disassembled and displayed on the screen.

Example

```
MONDFE.EXE>T 20
00010058 15846000 add      lr4,gr96,0x0
```

In the example above, **mondfe** is asked to trace 20 instructions. Prior to entry of the trace command, the next instruction to be executed was located at address 0x10008I. After tracing 20 instructions, **mondfe** prints out the next instruction to be executed, which is located at address 0x10058I.

Example

```
MONDFE.EXE>T
0001005c 01ff82ff constn   lr2,0xffff
```

In this example, **mondfe** is asked to trace a single instruction, beginning with the instruction at the memory location displayed by the previous example. Only one instruction is traced because there is no *count* parameter. After executing the instruction at address 0x10058, the next instruction, located at address 0x1005C, is displayed.

TIP — MONTIP Transparent Mode

Syntax: TIP *command*

where:

command Is a valid **montip** command.

The **TIP** command sends the *command* string to **montip** for execution. The command uses UDI Transparent Mode to pass the command string. The following **TIP** commands are supported:

- **tip lpt=0** requests that **montip** stop using the parallel port for communicating with 29K Family microcontroller targets.
- **tip lpt=1** requests that **montip** resume using the parallel port for communicating with 29K Family microcontroller targets.

The **TIP** command can be used before issuing a **YANK** (**Y**) command to download a program (COFF) file using the PC parallel port. The parallel port download capability is applicable only for an MS-DOS system. The parallel port to be used *must* be specified as a **montip** command-line option in the UDI configuration file using the **–par** command-line option of **montip**. (The UDI configuration file is **udiconfs.txt** on MS-DOS systems, and **udi_soc** on UNIX systems.)

Because the parallel port communication is unidirectional only, the serial communications port, **com1** or **com2**, also must be specified on the **montip** command line in the UDI configuration file.

The **TIP** command is valid only with the MiniMON29K **montip**.

VER — Display DFE, TIP, and Target Version Numbers

Syntax: VER

```
MONDFE.EXE>ver

                    MiniMON29K R2.0
     Copyright 1992 Advanced Micro Devices, Inc.

         Host code:
                   Version  2.3-11
                   Date:    4-Mar-92

          Tip code:
                   Version  2.3-11
                   Date:    4-Mar-92

          Target code:
                   Debug core version:            1.3
                   Configuration version:         0.5
                   Message system version:        1.0
                   Communication driver version:  0.0
                   OS system version:             0.4

Maximum message buffer size on target:   0x800
Maximum number of breakpoints on target:    30
MONDFE.EXE>
```

The **VERSION** (**VER**) command obtains the version numbers of the individual components of the MiniMON29K product, **mondfe**, **montip**, and the target system. It also prints the maximum message buffer size on the target.

This **mondfe** command can be executed only against the MiniMON29K target interface process, **montip**. Executing this command on any other TIP may cause unpredictable results.

XP — Display Special Registers

Syntax: XP

The **XP** command displays the contents of many of the special registers of the target processor, including the following:

- Old and current processor status (**OPS** and **CPS**)
- Vector area base (**VAB**)
- Configuration register (**CFG**)
- Channel registers (**CHA**, **CHD**, and **CHC**)
- Register bank protect (**RBP**)
- Timer counter reload (**TCR**)
- Timer counter value (**TCV**)
- Program counters (**PC0** through **PC2**)
- Memory management unit register (**MMU**)

The **XP** command displays the appropriate values for the type of Am29*xxx* processor installed.

Example

```
MONDFE.EXE>xp
       TD MM CA IP TE TP TU FZ LK RE WM PD PI SM IM DI DA
CPS:    0  0  0  0  0  0  0  0  0  0  0  1  1  1  0  0  0
OPS:    0  0  0  0  0  0  0  1  0  0  0  1  1  1  0  1  1

  VAB       CFG: PRL DW VF RV BO CP CD
80000000          03  1  1  0  0  0  0

  CHA       CHD     CHC: CE CNTL CR LS ML ST LA TF TR NN CV
80001d14  00000000        0    0  0  1  0  0  0  0 f0  0  0

RBP: BF BE BD BC BB BA B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
      0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1

 TCV TR: OV IN IE   TRV     PC0      PC1      PC2
fff3ef    0  0  1  ffffff 00001a54 00001a50 00002590

MMU: PS PID LRU
     03  01  10

MONDFE.EXE>
```

The example output from the **XP** command is shown above. For special registers that have multiple fields (e.g., **CPS, OPS,** and **CFG**), the value of each field is shown.

Y — Yank (Download) a COFF File

Syntax: Y [–noi|–i] [–ms *mem_stack_size_in_hex*]
[–rs *reg_stack_size_in_hex*] [–*sections*]
[*Coff_Filename* [*program_args*]]

where:

–noi|–i Specifies initialization of a process for the downloaded program (COFF file). When the **–noi** option is specified, no process is initialized for the downloaded program. When the **–i** option is specified, a process is initialized for the downloaded program and is ready for execution (using the **G** command). The **–i** option is the default.

–ms *mem_stack_size_in_hex*
Specifies the memory stack size to use at the time of process initialization (warm start) for the downloaded program. The value specified is interpreted as a hexadecimal value, which can override the value specified on the command line of **mondfe** at the time of its invocation.

–rs *reg_stack_size_in_hex*
Specifies the register stack size to use at the time of process initialization (warm start) for the downloaded program. The value specified is interpreted as a hexadecimal value, which can override the value specified on the command line of **mondfe** at the time of its invocation.

–*sections* Downloads specific COFF sections of the COFF file specified. The *sections* option can be composed of one or more of the following letters:

b	Indicates BSS sections
d	Indicates DATA sections
l	Indicates LIT sections
t	Indicates TEXT sections

Only the sections specified using the above letters are downloaded. The default is to download all sections. For example, **–td** downloads only the TEXT and the DATA sections of the COFF file.

Coff_Filename [*program_args*]

Specifies the relative pathname of the COFF file to be downloaded. Only files in COFF format can be downloaded using **mondfe**. *program_args* is the optional list of command-line arguments for the downloaded program.

The *Coff_Filename* option is always the first element of this list. When no *Coff_Filename* is specified, the filename specified in the previous **YANK** (**Y**) command is used. However, a *Coff_Filename* option must precede the optional *program_args* parameter.

The **–i** and **–noi** options can be used to download COFF files either for execution or as an easy way to assemble a set of commands into memory. For example, the following code downloads the TEXT sections of the COFF file **assemble.out** into memory:

```
y –noi –t assemble.out
```

The **–noi** option does not cause a process to be initialized for execution. Therefore, any previously stopped program can be resumed by issuing the **GO** (**G**) command.

The **mondfe** command **INIT** initializes a process for a downloaded COFF file, if none was initialized at the time of downloading.

The **–ms** and **–rs** options to the **YANK** (**Y**) command specify the memory stack and register stack requirements for the program being downloaded. The underlying operating system uses these values during warm start to set up the stack environment for the program.

The *–sections* option selectively loads specific COFF sections onto the target. This downloads only the DATA and BSS sections in cases when the TEXT and LIT sections from the previous download are known to be undisturbed. The *–sections* option must be comprised of one or more of the letters **t**, **d**, **l**, and **b**, as explained previously.

The execution mode (real or protected) of the program is controlled by the TIP. In the case of **montip**, the TIP options **–R** and **–P** select real or protected mode, respectively, for the program.

Example

```
MONDFE.EXE>Y test.out
Loading TEXT section at 0x10000 (17872 bytes) ...
Loading LIT section at 0x16000 (292 bytes) ...
Loading DATA section at 0x18000 (1504 bytes) ...
Loading DATA section at 0x1a000 (188 bytes) ...
Loading BSS section at 0x1c000 (924 bytes) ...
```

In the example above, **mondfe** loads a file named **test.out**, which is taken from the current directory. The output produced by loading that file is shown. The TEXT section is the program code. The LIT section contains read-only data. The DATA sections contain initialized constant and variable data. The BSS section contains space for uninitialized variables. The default values for the options not specified are used.

ZC — Execute Commands From Command File

Syntax: ZC *cmdfile_name*

where:

cmdfile_name Is the filename of a **mondfe** command file.

The **ZC** command executes a series of **mondfe** commands from the *cmdfile_name* command file. The *cmdfile_name* is the name of the file containing the command input. The **ZC** command can be executed at the **mondfe** command prompt. When all commands from the file are executed, the **mondfe>** prompt appears again.

Nesting of command files is not allowed.

ZE — Specify Echo File for Echo Mode

Syntax: ZE *echofile_name*

where:

echofile_name Is the name of the echo file.

The **ZE** command turns on echo mode and specifies the echo file. When echo mode is on, everything that is displayed on the screen is written into the echo file. The *echofile_name* string specifies the filename of the echo file.

ZL — Specify Log File for Log Mode

Syntax: ZL *logfile_name*

where:

logfile_name Is the filename of the log file.

The **ZL** command turns on log mode and specifies the log file to use. When log mode is on, every **mondfe** command entered by the user is logged in the log file. This log file can be used directly as an input command file for subsequent debug sessions to repeat the same sequence of commands. Log mode can be turned on or off by using the **logon** or **logoff** command.

Chapter 3

MONDFE Tutorial

This chapter provides a step-by-step approach for debugging an example program that contains intentional bugs. Although the example program is relatively simple, the methodology presented in the tutorial is applicable to many of the debugging situations that a user may encounter.

This tutorial program (**buggy.c**) is run on an Am29000 microprocessor, compiled using AMD's High C 29K compiler, and linked with the High C 29K linker. In the tutorial, compiling and linking are accomplished with a single command that compiles the source file, links it with the necessary library routines, and produces a standard AMD COFF file that can be executed on the AMD EB29K™ board. The tutorial example uses a stand-alone board as the target. This is used instead of a PC plug-in card because it can be reproduced from either MS-DOS or UNIX hosts. Note that the MiniMON29K software already is ported and running on a number of hardware platforms supported by AMD.

Advance preparation for a debugging session is an important first step. In this tutorial, instructions are provided for preparation of batch and shell script files, and a linker command file, and instructions for entering commands that illustrate the debugging features of the MiniMON29K debugger core.

NOTE: In the following tutorial, the **C:>** and the **MONDFE.EXE>** indicate prompts by the computer, and should not be typed in.

Demo Directory for MS-DOS Hosts

The **demo** directory under the **29k\tutorial\minimon** directory contains batch files and 29K Family executables that can be run as is on an EB29K board or on an Am29000 processor-based stand-alone board with a MiniMON29K target in PROMs. Note that the **PATH** and **UDICONF** environment variables must be set to locate the executables, the MiniMON29K target to download, and the UDI configuration file, **udiconfs.txt**. Refer to the UDI man pages for a detailed description of the search rules.

ebdemo.bat

This batch file runs the demo on an EB29K target. The **ebdemo.bat** file uses the **ebdemo.cmd** file as the input command file, and loads and executes **buggy.eb** and **buggyfix.eb** files, illustrating the bug in the **buggy.eb** program. The **ebdemo.out** file contains the complete session for reference. The files **buggy.map** and **buggyfix.map** are the map files for **buggy.eb** and **buggyfix.eb,** respectively. The C source files are in the **src** directory.

run384.bat

This batch file runs the demo on an Am29000 processor-based stand-alone board connected over a serial communications link. The assumed baud rate is 38400 bps. The **run384.bat** file uses the **run384.cmd** file as the input command file, and loads and executes **buggy.stb** and **buggyfix.stb** files, illustrating the bug in the **buggy.stb** program. The file **run384.ou**t contains the complete session for reference. The files **buggystb.map** and **bfixstb.map** are the map files corresponding to **buggy.stb** and **buggyfix.stb**, respectively. The C source files are in the **src** directory.

Preparing Batch Files for MS-DOS Hosts

The **compile.bat**, **runeb29k.bat**, and **runsteb.bat** batch files are located in the **29k\tutorial\minimon\src** directory of the product installation tree. These files can be modified to suit the user's work environment.

Two linker command files used in the tutorial also are placed in this directory: **load.lnk** and **ebload.lnk**.

compile.bat

```
hc29 –v –cmdebload.lnk –o %1.out –m %1.c > %1.map
```

This batch file compiles a C program using the High C 29K compiler, **hc29**. It invokes the compiler in verbose mode (**–v**) and specifies the linker command file to use (e.g., **ebload.lnk**) by using the **–cmd** option of **hc29**. A map file also is generated and redirected to a file with the **.map** extension and the same base name as that of the C source file. The output is placed in a file with the **.out** extension and the same base name as that of the C source file. This batch file can be used by specifying the base name of the C source file (e.g., **buggy** for **buggy.c**) as the first argument.

runeb29k.bat

```
mondfe –D –TIP eb29k
```

This batch file expects the **PATH** and **UDICONF** variables to be set appropriately. The **UDICONF** variable should be set to use the **udiconfs.txt** file for an MS-DOS host. The **PATH** variable must be set to invoke the **mondfe.exe** and **montip.exe** executables. The **PATH** variable must be set to use the target object **eb29k.os** file to download. This batch file invokes **mondfe** and **montip** for executing programs interactively on an EB29K PC plug-in card target.

runsteb.bat

```
mondfe –D –TIP serial384
```

This batch file expects the **PATH** and **UDICONF** environment variable to be set appropriately. The **UDICONF** should be set to use the correct **udiconfs.txt** file for an MS-DOS host. The **PATH** variable must be set to invoke the **mondfe.exe** and **montip.exe** executables. This batch file can be used to invoke **mondfe** and **montip** to execute programs interactively on a stand-alone target board connected over a serial communications link. The assumed baud rate is 38400 bps.

Linker Command Files for the MONDFE Tutorial

ebload.lnk

```
ALIGN=8192
RESNUM  0x0, 0x10000
ORDER   .text=0x10000, !text
ORDER  .data=0x80004000, .data1, !data, .lit, !lit, .bss,!bss
```

This linker command file produces a COFF file for an EB29K target. It orders the TEXT section in the Instruction memory, reserving the first 0x10000 bytes for the MiniMON29K debugger core and the optional kernel. The DATA sections are ordered starting at 0x80004000, reserving the first 0x4000 bytes for use by the MiniMON29K debugger core and the AMD example kernel, **osboot**. The **ALIGN** directive forces the alignment of sections on an 8K-page boundary. This alignment must match the Page Size bits of the MMU register to successfully run the user's program in protected mode using **montip**. The default for the Page Size bits of the MMU register is 8K-page.

load.lnk

```
ALIGN=8192
RESNUM  0x0, 0x10000
```

This linker command file produces a COFF file for a target that has a joint I and D memory space. The first 0x10000 bytes are reserved for the MiniMON29K debugger core and the AMD example kernel, **osboot**.

Demo Directory for UNIX Hosts

The **demo** directory under the **29k/tutorial/minimon** directory for UNIX hosts consists of a C shell script that runs the demo programs on an Am29000 processor-based stand-alone board connected over a serial communications link. The assumed baud rate is 38400 bps. Note that the **PATH** and **UDICONF** environment variables must be set to locate the executables and the UDI configuration file, **udi_soc**. Refer to the UDI man pages for a detailed description of the search rules.

run384 C Shell Script

This batch file runs the demo on an Am29000 processor-based stand-alone board connected over a serial communications link. The assumed baud rate is 38400 bps. It uses **run384.cmd** as the input command file, and loads and executes **buggy.stb** and **buggyfix.stb**, illustrating the bug in the **buggy.stb** program. The file **run384.out** contains the complete session for reference. The files **buggystb.map** and **bfixstb.map** are the map files corresponding to **buggy.stb** and **buggyfix.stb,** respectively. The C source files are in the **src** directory.

Preparing Shell Script Files for UNIX Hosts

Two C shell scripts are placed under the **29k/tutorial/minimon/src** directory: **compile** and **runsteb**. These C shell scripts can be modified to suit the user's work environment.

Two linker command files used in the tutorial also are placed in this directory: **load.lnk** and **ebload.lnk**.

compile

```
hc29 –v –cmdload.lnk –o $1.out –m $1.c > $1.map
```

This C shell script compiles a C source program for an Am29000 processor-based stand-alone board. It invokes the compiler in verbose (**–v**) mode and specifies **load.lnk** as the linker command file to use. The output is placed in the file with the **.out** extension and the base name as that of the input C source file. A map file is generated and stored in a file with the **.map** extension and the same base as that of the input C source file. The base name of the C source program is specified as the first argument to the shell script.

runsteb

```
mondfe –D –TIP serial384
```

This C shell script expects the **PATH** and **UDICONF** environment variable to be set appropriately. The **UDICONF** should be set to use the correct **udi_soc** file on the UNIX host. The **PATH** variable must be set to use **mondfe** and **montip**. This shell script invokes **mondfe** and **montip** to execute programs interactively on a stand-alone target board connected over a serial communications link. The assumed baud rate is 38400 bps.

Compiling the Tutorial Example

The **src** directory under the **29k\tutorial\minimon** directory contains the tutorial program, which is written in C language and can be compiled with the High C 29K compiler. The program is included with the product, so that the user does not have to create it. The program name is **buggy.c**, indicating it contains errors, but none that will be discovered by the compiler. The **buggyfix.c** source file fixes the bugs that were purposely introduced into the **buggy.c** program. The extensions of **.map** and **.out** represent the linker map and absolute COFF files associated with their respective source files.

Note that the examples throughout the remainder of this tutorial are shown in MS-DOS; however, equivalent UNIX filenames are listed in parentheses where appropriate.

A listing of **buggy.c** is shown below:

```
/*
** Program listing for buggy.c, a demonstration program
** for use with the mondfe tutorial.
*/
#include <stdio.h>
double Fahrenheit, Celsius;

main()
{
   for(; ;)
   {
      printf(”Enter a temperature (in Fahrenheit): ”);
      if(scanf(”%lf”, Fahrenheit) < 1) exit(0);
      Celsius = Fahrenheit – 32 * 0.555555556;
      printf(”A temperature of %6.2lf (Fahrenheit) = %6.2lf
                  (Celsius)\n”,
      Fahrenheit, Celsius);
   }
}
```

The bugs in this program may be apparent, but follow along with the tutorial to learn how to use the commands in the MiniMON29K software to discover these problems. The methodology for discovering and fixing these bugs can be applied to debugging more complex programs.

The first step is to compile the program. The **compile.bat** file (**compile** for UNIX versions) provided can be used to produce an absolute COFF file for an EB29K target. For this example, the **compile.bat** file was edited to use **load.lnk** instead of **ebload.lnk** as the linker command file. You may have to edit this file to suit your target system. To compile a C program, enter the following command at your host computer's operating-system prompt (assuming that you are using the batch files discussed above).

Before entering the command, you must be in the directory containing the **buggy.c** source file and the batch files.

```
C:> compile buggy
```

The **C:>** shown above is the MS-DOS prompt for a command to be entered. Type only the two words **compile buggy** followed by a carriage return. This compiles and links the **buggy.c** program, producing a **buggy.map** map file and a **buggy.out** absolute COFF file.

The **buggy.map** file contains information that will be used in subsequent debugging steps. In particular, the following lines (extracted from the **buggy.map** file) are pertinent to the remaining tutorial steps:

```
_main       text  00010120
_Fahrenheit bss   0001C0B8
_Celsius    bss   0001C0A4
```

Note that these addresses may vary, depending on the compiler and library versions you use.

These three locations were taken from the global symbol table listing in the **buggy.map** file. It is a common practice for C compilers to prefix symbols with an underscore character. The **main** function is located in the TEXT section of the COFF file and is found at memory location 0x10120 in the loaded program. The global variables, **Fahrenheit** and **Celsius**, are located in the BSS section, at addresses 0x1C0B8 and 0x1C0A4, respectively. The TEXT section identifies locations in instruction memory, and DATA and BSS sections identify locations in data memory.

Loading and Running the Program

After the compilation of **buggy.c** is complete, the next step in the debugging process is to invoke the MiniMON29K Debugger Front End **mondfe**. (The target is a stand-alone execution board based on an Am29000 microprocessor, connected over a serial link transmitting at 9600 bps.) To invoke **mondfe**, enter the following command:

```
C:> mondfe –D –TIP serial96
```

The batch file **runsteb.bat** (**runsteb** for UNIX versions) invokes **mondfe** for this tutorial. As indicated in the compilation step, the **C:>** characters are the operating system prompt. Make sure that your **PATH** environment variable is set up to invoke the **mondfe.exe** and **montip.exe** executables. Also, ensure that the **UDICONF** environment variable is set to the **udiconfs.txt** file for MS-DOS hosts or the **udi_soc** file for UNIX versions.

After **mondfe** establishes communications with the target, it outputs a prompt, soliciting a command. The prompt is shown below:

```
MONDFE.EXE>
```

When this prompt is shown in the following narrative, it is not to be typed. It is included only to define the context in which the command is being entered. To load the absolute COFF file produced by the previous compilation step, enter the following command:

```
MONDFE.EXE> Y buggy.out
```

This command causes **mondfe** to read the COFF file (**buggy.out**), load its various sections into the memory locations specified during the linking process, and prompt for the next command.

After the program has been loaded, **mondfe** reports the locations of the COFF file sections. In the case of **buggy.out**, these locations are shown below:

```
Loaded TEXT section at 0x10000 (22412 bytes)
Loaded LIT section at 0x16000 (356 bytes)
Loaded DATA section at 0x18000 (1632 bytes)
Loaded DATA section at 0x1a000 (52 bytes)
Cleared BSS section at 0x1c000 (620 bytes)
```

In the above output, the TEXT section is the program's code. The LIT section contains read-only data. The two DATA sections contain initialized constants and variables, and the BSS section contains uninitialized variables.

Referring back to the information from the **buggy.map** file, you see that the **main** function of the program is located within the TEXT section, and the two variables of interest (**Fahrenheit** and **Celsius**) are located in the BSS section. The next step is to run the program. Enter the following command at the **mondfe** prompt:

```
MONDFE.EXE> G
```

The **G** command starts execution at the point where execution was previously halted.

When this command is entered, the following output is displayed on the screen:

```
Enter a temperature (in Fahrenheit): _
```

The underscore character at the end represents the cursor position on the terminal display. Respond by typing a numeric temperature value, such as 212.0, followed by a carriage return. The following output appears on the display:

```
User-mode data TLB miss (trap 9).
```

This trap indicates that an attempt was made to access an illegal data location while executing the program.

Debugging the Program

At this point, having discovered an error, it is time to enter the debugging phase of our program development. Looking at the illegal access trap that just occurred, attempt to isolate the location being accessed by displaying the contents of the program counters at the time the exception occurred. To display the contents of the program counters, enter the following command:

```
MONDFE.EXE> D PC0 PC2

sr008                       00012448 00012444
sr012    00012440
```

The output, shown below the command, indicates that the processor was executing the instruction located at address 0x12440 when the exception occurred. To list this instruction and the two that follow, enter the following command:

```
MONDFE.EXE> L 12440I 12448I

00012440 1e006062 store    0,0x0,gr96,gr98
0000c444 a0ff00c8 jmp      0xc364
0000c448 1e00617a store    0,0x0,gr97,gr122
```

This output indicates that the program is attempting to store the values contained in global registers **gr96** and **gr97** into the addresses contained in global registers **gr98** and **gr122**. To display the contents of these registers, enter the following:

```
MONDFE.EXE> D GR96 GR122

gr96     406a0000 00000000 00000000 00000000
gr100    000124e8 00000002 00000000 00000000
. . .
gr120    deff5eee ffffffff 00000004
```

In the output listed above, some lines were eliminated for the sake of brevity. This output indicates that the addresses at which the value in **gr96** (0x406a0000) and the value in **gr97** (0x00000000) are being stored are 0x00000000 and 00000004, respectively. Both of these are illegal data addresses. Recall that the legal DATA sections indicated by the COFF file began at 0x18000 (1632 bytes) and 0x1a000 (52 bytes).

Further inspection of the contents of global registers **gr96** and **gr97** reveals that they may contain a double-precision floating-point value. To display their contents in that form, enter the following:

```
MONDFE.EXE> DD GR96 GR97

gr96      +2.120000000000000e+002
```

This output shows that the value entered (212.0) is the one being stored in an illegal address.

Looking at the **buggy.map** file, you can identify the function that was executing at the time the exception occurred. The first part of the map file lists the section summary. Remembering that the program counter held the value 0x00012440, look for a **.text** section that spans that area of instruction memory. The following entry is found:

```
.text     TEXT
                    00011A0C 00012453 ... (_doscan.o)
```

This indicates that the bug appears in the library module **_doscan.o**, which is probably associated with the **scanf** function. At this point, it is reasonable to assume that something is wrong with the call to **scanf** in the program.

Because the call to **scanf** appears in the first part of the program just after the prompt is displayed, list the program code, beginning at the start of the **main** routine. Referring to the map again, you can see that **_main** is located beginning at address 0x00010120. Enter the following command :

```
MONDFE.EXE> L 10120I 1015CI

00010120 25010120 sub      gr1,gr1,0x20
00010124 5e40017e asgeu    0x40,gr1,gr126
00010128 15810128 add      lr1,gr1,0x28
0001012c 03608200 const    lr2,0x6000
00010130 0303606c const    gr96,0x36c
00010134 02006001 consth   gr96,0x1
00010138 c8008060 calli    lr0,gr96
0001013c 02008201 consth   lr2,0x1
00010140 03608228 const    lr2,0x6028
00010144 02008201 consth   lr2,0x1
00010148 03c062b8 const    gr98,0xc0b8
0001014c 02006201 consth   gr98,0x1
00010150 16006462 load     0,0x0,gr100,gr98
00010154 157a6204 add      gr122,gr98,0x4
00010158 1600657a load     0,0x0,gr101,gr122
0001015c 15836400 add      lr3,gr100,0x0
```

You can see from looking at the **buggy.c** source program that **printf** is the first routine called. Its location is shown in the map file (**_printf**) as 0001036C. Noting in the above listing that this value is being loaded into global register **gr96** and that the **CALLI** instruction calls the routine pointed to by the contents of **gr96**, assume that this is the call to the **printf** routine.

List another set of instructions before the next call by entering the following command:

```
MONDFE.EXE>L
```

```
00010160    0304607c    const    gr96,0x47c
00010164    02006001    consth   gr96,0x1
00010168    c8008060    calli    lr0,gr96
0001016c    15846500    add      lr4,gr101,0x0
00010170    4d606001    cpge     gr96,gr96,0x1
00010174    ac006005    jmpt     gr96,0x10188
00010178    032b60a8    const    gr96,0x2ba8
0001017c    02006001    consth   gr96,0x1
00010180    c8008060    calli    lr0,gr96
00010184    03008200    const    lr2,0x0
00010188    03c7601c    const    gr96,0xc71c
0001018c    02406031    consth   gr96,0x4031
00010190    033161f1    const    gr97,0x31f1
00010194    02726104    consth   gr97,0x7204
00010198    036062b8    const    gr98,0x60b8
0001019c    02006201    consth   gr98,0x1
```

The first two instructions in the above code load the address 0x1047C into global register **gr96**, and the third instruction calls the routine. You have located the context of the call, but have gone too far into the program. Back up to the previous list to see the code that immediately precedes this call. It is shown below, for reference:

```
00010140    03608228    const    lr2,0x6028
00010144    02008201    consth   lr2,0x1
00010148    03c062b8    const    gr98,0xc0b8
0001014c    02006201    consth   gr98,0x1
00010150    16006462    load     0,0x0,gr100,gr98
00010154    157a6204    add      gr122,gr98,0x4
00010158    1600657a    load     0,0x0,gr101,gr122
0001015c    15836400    add      lr3,gr100,0x0
00010160    0304607c    const    gr96,0x47c
00010164    02006001    consth   gr96,0x1
00010168    c8008060    calli    lr0,gr96
0001016c    15846500    add      lr4,gr101,0x0
```

Note that local register **lr2** is being loaded with the constant 0x16028, which is the address of a value in the LIT section of the program. This is the address of the format string, which is the first parameter to the **scanf** function. The second parameter should be the address of the Fahrenheit variable, and this should be placed into local register **lr3** prior to making the call.

The address of the Fahrenheit variable is 0x160B8, so you should see it referenced in the code above. What you see, however, is the following sequence of instructions (with explanatory comments added):

```
00010148  const    gr98,0xc0b8      ; put 0x1C0B8 into gr98
0001014c  consth   gr98,0x1         ; put 0x1C0B8 into gr98
00010150  load     0,0x0,gr100,gr98 ; put 0x1C0B8 into gr100
00010154  add      gr122,gr98,0x4
00010158  load     0,0x0,gr101,gr122
0001015c  add      lr3,gr100,0x0    ; put it into lr3
```

It seems clear at this point that the contents, rather than the address, of the **Fahrenheit** value are being loaded into **lr3**. Going back to the source code, you find that in the call to **scanf**, an ampersand character (&) was not placed in front of **Fahrenheit**. The correct source code line should appear as shown below:

```
if(scanf(”%lf”, &Fahrenheit) < 1) exit(0);
```

Exit **mondfe** by entering the following command:

```
MONDFE.EXE> Q
```

Make this correction to the source code in **buggy.c** and recompile, using the same procedure indicated in the “Compiling the Tutorial Example” section on page 3–7.. Next, invoke **mondfe**, reload the COFF file (**buggy.out**) using the **Y** command, and start program execution by entering the **G** command. This sequence of operating system and **mondfe** commands is listed below:

```
C:> compile buggy
C:> mondfe –D –TIP serial384

MONDFE.EXE> Y buggy.out
MONDFE.EXE> G
```

When you are asked to enter a temperature, again enter 212. The output should appear as shown below:

```
Enter a temperature (in Fahrenheit): 212.0
A Temperature of 212.00 (Fahrenheit) = 194.22 (Celsius)
```

Well, at least this time there is output. Unfortunately, a temperature of 212.0 degrees Fahrenheit should be 100.0 degrees Celsius. Something else is wrong. It must be in the computation of the Celsius value.

At this point terminate the program by entering any non-numeric value, such as the following, and observe the output:

```
Enter a temperature (in Fahrenheit): end

Program exited (exit code 0)
```

The output indicates that the program has exited, because you indicated in the **if** statement that if less than one numeric value is entered in response to the **scanf** call, the program should call the **exit** routine. The display reflects that this has occurred. Because of the recompilation, the object code has changed, so it is uncertain where the computation begins. In addition, the contents of the **buggy.map** file has changed. The relevant entries in the new map file are shown below:

```
_main             text              00010120
_scanf            text              00010468
_Fahrenheit       bss               0001c0B8
_Celsius          bss               0001c0A4
```

To locate the point at which the computation begins, execute the following commands:

```
MONDFE.EXE> Y buggy.out
MONDFE.EXE> B 10468I
MONDFE.EXE> G
breakpoint hit at 00010468
00010468 25010118 sub      gr1,gr1,0x18
MONDFE.EXE> D lr0 lr0

lr000             0001015c

MONDFE.EXE>
```

To summarize:

1. The first command reloads the **buggy.out** file.
2. The second command sets a breakpoint at the entry point of the **scanf** routine.
3. The third command tells **mondfe** to begin execution.

Immediately, **mondfe** reports that the breakpoint you set has been reached. You can see from examining the code that the **CALLI** instruction that invokes the **scanf** routine stores the return address in local register **lr0** when the call is made.

4. The fourth command then displays the register contents of **lr0**. By displaying the contents of local register **lr0** immediately after the breakpoint has been hit, you find the return address in the main program. In this case, it is 0x1015C.

The next step is to set a breakpoint at the return location, so that the program can run until the next breakpoint is reached. Enter the following commands, and examine the output:

```
MONDFE.EXE> B 1015CI
MONDFE.EXE> G
Enter a temperature (in Fahrenheit): 212.0
breakpoint hit at 0001015c
0001015c 4d606001 cpge     gr96,gr96,0x1
MONDFE.EXE>
```

After the new breakpoint is installed and the program continues execution (when the **G** command has been entered), it immediately prompts for a temperature value. Entering 212.0 again causes **scanf** to store the value in the **Fahrenheit** variable and return to the main program. At this point, the newly installed breakpoint has been hit. The **CPGE** instruction displayed at the breakpoint location is the comparison associated with the **if** statement in the source program.

Because you do not know how many instructions are associated with the computation of the Celsius temperature, the best approach at this point is to trace the program execution, one instruction at a time. Enter the following command:

```
MONDFE.EXE> T

00010160 ac006005 jmpt     gr96,0x10174
```

Entering the **T** command with no options or parameters traces a single instruction (in this case the **CPGE** instruction that was previously displayed), and displays the next instruction to be executed.

The **JMPT** instruction should jump, because a valid numeric value was entered at the prompt for a temperature value (**scanf** should return a value of 1, indicating that one item has been successfully scanned); however, the instruction immediately after a jump instruction is *always* executed. Therefore, you should expect (in the following procedure) to see that the instruction at address 0x10164 is being traced. At that point, the program should trace the instruction located at the jump target location.

Continue entering the **T** command until the following set of instructions has been executed and the **DSUB** instruction is displayed as the next one to be executed. (Most of the **T** commands have been omitted for the sake of brevity.)

```
MONDFE.EXE> T
00010164 032b6094 const    gr96,0x2b94
00010170 032b6094 const    gr96,0x2b94
00010174 03c7601c const    gr96,0xc71c
00010178 02406031 consth   gr96,0x4031
0001017c 033161f1 const    gr97,0x31f1
00010180 02726104 consth   gr97,0x7204
00010184 03c083b8 const    lr3,0xc0b8
00010188 02008301 consth   lr3,0x1
0001018c 16006483 load     0,0x0,gr100,lr3
00010190 157a8304 add      gr122,lr3,0x4
00010194 1600657a load     0,0x0,gr101,gr122
MONDFE.EXE> T
00010198 f3626460 dsub     gr98,gr100,gr96
```

Ignore the first instruction (at location 0x10164, which immediately follows the **JMPT** instruction). The first two **CONST**/**CONSTH** instruction pairs load a double-precision floating-point value into global registers **gr96** and **gr97**. You can display this value by entering the following command:

```
MONDFE.EXE> DD GR96 GR97

gr096             +1.777777779200000e+001
```

At this point, it is instructive to repeat the source program line that computes the Celsius temperature, as shown below:

```
Celsius = Fahrenheit – 32 * 0.55555556;
```

It is clear that a value of 17.777777792 does not appear anywhere in our equation, so the equation must have been interpreted incorrectly. In fact, this is exactly the case. If you multiply 32 by 0.55555556, the result is 17.77777792. Clearly, the compiler is giving precedence to the multiplication, before the subtraction is executed. To correct the program, put parenthesis around the subtraction of 32.0 from the Fahrenheit variable, as shown below in the corrected statement:

```
Celsius = (Fahrenheit – 32) * 0.55555556;
```

It is clear that a mistake has been found, so terminate **mondfe** by entering the following command:

```
MONDFE.EXE> Q
```

Edit the **buggy.c** program to correct this error, and recompile the program as previously described.

Now that the program has been recompiled, you can re-invoke **mondfe**, load the program and execute it by entering the following series of commands:

```
C:> mondfe –D –TIP eb29k

MONDFE.EXE> Y buggy.out
MONDFE.EXE> G
```

At the first prompt to enter a temperature, again enter the value 212.0. The resulting output is shown below:

```
Enter a temperature (in Fahrenheit): 212.0
A temperature of 212.00 (Fahrenheit) = 100.00 (Celsius)
Enter a temperature (in Fahrenheit): _
```

As you can see, the result is correct and the program has prompted for another temperature value. Try entering 32.0 (the Fahrenheit temperature at which water freezes). The result is 0.00, the corresponding Celsius temperature. You can continue entering different temperatures and verifying the results. To exit the program, enter any non-numeric value at the temperature prompt. **mondfe** will acknowledge that the program has exited, and you can quit **mondfe** by entering the **Q** command at the **MONDFE.EXE>** prompt.

The debugging technique just presented represents a combination of analysis and action. This is common in the debugging process. You have to perform the analysis, but the MiniMON29K debugger core can do a lot of work to help you verify your assumptions. A common methodology for debugging C programs, consists of editing, compiling, and testing; these steps are repeated as many times as necessary to verify that the program is correctly performing its intended task.

mondfe incorporates many features that have not been explored in this tutorial; however, they are just as easy to use. Commands are provided for changing and displaying the contents of registers or memory, setting both sticky and non-sticky breakpoints, assembling and disassembling instructions using standard AMD mnemonics, and more.

In addition to the **mondfe** command features, it is worthwhile to use the log file as a source of commands for subsequent debugging sessions. Invoke **mondfe** using the **–log** *logfile_name* command-line parameter and rename this file immediately after quitting a debugging session. Use the file later as input by including the **–c** *cmd_file* option on the command line when you invoke **mondfe.** (Remember to delete the **Q** command at the end of the file before using the file as input.)

Appendix A

Error Messages

The following pages describe the error messages produced by a command typed at the MONDFE prompt as well as those produced by the UDI layer.

MONDFE Error Messages

Table A–1 describes error messages reported in response to a user command typed at the **MONDFE.EXE>** prompt. Error messages can be prefixed by a warning or fatal error message.

When **mondfe** can recover from the error, it displays a warning message. Warnings are prefixed by "DFEWARNING," and they do not cause **mondfe** to terminate.

When **mondfe** cannot recover from the error, it displays a fatal error message. Fatal error messages are prefixed by "DFEERROR," and they cause **mondfe** to terminate.

Table A–1. MONDFE Error Messages

Error Code	Error Message	
2	EMFAIL:	Unrecoverable error.
3	EMBADADDR:	Illegal address.
4	EMBADREG:	Illegal register.
5	EMSYNTAX:	Illegal command syntax.
6	EMACCESS:	Could not access memory.
7	EMALLOC:	Could not allocate memory.
8	EMTARGET:	Unknown target type.
9	EMHINIT:	Could not initialize host.
10	EMCOMM:	Could not open communication channel.
11	EMBADMSG:	Unknown message type.
12	EMMSG2BIG:	Message too large for buffer.
13	EMRESET:	Could not RESET target.
14	EMCONFIG:	Could not get target CONFIG.
15	EMSTATUS:	Could not get target STATUS.
16	EMREAD:	Could not READ target memory.
17	EMWRITE:	Could not WRITE target memory.

Error Code	Error Message	
18	EMBKPTSET:	Could not set breakpoint.
19	EMBKPTRM:	Could not remove breakpoint.
20	EMBKPTSTAT:	Could not get breakpoint status.
21	EMBKPTNONE:	All breakpoints in use.
22	EMBKPTUSED:	Breakpoint already in use.
23	EMCOPY:	Could not COPY target memory.
24	EMFILL:	Could not FILL target memory.
25	EMINIT:	Could not initialize target memory.
26	EMGO:	Could not start execution.
27	EMSTEP:	Could not single step.
28	EMBREAK:	Could not BREAK execution.
29	EMHIF:	Could not perform HIF service.
30	EMCHANNEL0:	Could not read CHANNEL0.
31	EMCHANNEL1:	Could not write CHANNEL1.
32	EMOPEN:	Could not open COFF file.
33	EMHDR:	Could not read COFF header.
34	EMMAGIC:	Bad COFF file magic number.
35	EMAOUT:	Could not read COFF a.out header.
36	EMSCNHDR:	Could not read COFF section header.
37	EMSCN:	Could not read COFF section.
38	EMCLOSE:	Could not close COFF file.
39	EMLOGOPEN:	Could not open log file.
40	EMLOGREAD:	Could not read log file.
41	EMLOGWRITE:	Could not write log file.
42	EMLOGCLOSE:	Could not close log file.
43	EMCMDOPEN:	Could not open command file.
44	EMCMDREAD:	Could not read command file.
45	EMCMDWRITE:	Could not write command file.
46	EMCMDCLOSE:	Could not close command file.

Error Code	Error Message	
47	EMTIMEOUT:	Host timed out waiting for a message.
48	EMCOMMTYPE:	A **–t** flag must be specified.
49	EMCOMMERR:	Communication error.
50	EMBAUD:	Invalid baud rate specified.
51	EMTIPINIT:	TIP initialization failed. Exiting TIP.
52	EMIOSETF:	Host I/O setup failure.
53	EMIORESETF:	Host I/O reset failure.
54	EMLOADF:	Loading COFF file failure.
55	EMNOFILE:	No program to run.
56	EMECHOPEN:	Could not open echo file.
57	EMCTRLC:	Ctrl-C interrupt. Exiting.
58	EMNOSUCHCMD:	Unrecognized command.
59	EMNOPROCESS:	Failed creating process zero.
60	EMNOTCOMP:	DFE and TIP versions not compatible.
61	EMFATAL:	No session in progress.
62	EMNOINITP:	(**–n**) No process initialized for downloaded program.
63	EMDOSERR:	DOS error. Cannot escape to DOS.
64	EMSYSERR:	System error. Cannot escape to host.
65	EMINCECHOFILE:	Invalid echo file. Cannot enable echo.
66	EMCMDFILENEST:	Nesting of command files not allowed.

UDI Error Messages

Error messages reported by the UDI layer and by the underlying TIP may also be displayed by **mondfe**. UDI error messages are prefixed by "UDIERR" and a number that corresponds to the error numbers defined in the UDI specification. Table A–2 describes UDI error messages returned by **mondfe**.

TIP error messages are prefixed by "TIPERROR." See the error codes listed in the appropriate TIP manual for TIP errors.

Table A–2. UDI Error Messages

Error Code	Error Message	
1	UDIERR:	No Such Configuration in Config File.
2	UDIERR:	Cannot Happen With Current Environment Setup.
3	UDIERR:	Cannot Connect to TIP Specified.
4	UDIERR:	No Such Connection Found.
5	UDIERR:	No Connection Occurred.
6	UDIERR:	Cannot Open UDI Config File.
7	UDIERR:	Cannot Start TIP In Current Environment Setup.
8	UDIERR:	Requested Connection Unavailable.
9	UDIERR:	Try Another TIP For Connection.
10	UDIERR:	TIP Specified in Config File Not An Executable.
11	UDIERR:	Connection Failed Due To Invalid TIP Options in Config File.
12	UDIERR:	Cannot Disconnect TIP.
13	UDIERR:	Unknown Error Number Specified.
14	UDIERR:	TIP Cannot Create a New Process.
15	UDIERR:	No Such Process in the Current TIP.
16	UDIERR:	Unknown Resource Space Encountered By TIP.
17	UDIERR:	Invalid Resource Specified To TIP.
18	UDIERR:	Unsupported Step Type For This TIP Specified.
19	UDIERR:	Could Not Set The Breakpoint.

Error Code	Error Message	
20	UDIERR:	Too Many Breakpoints Already In Use.
21	UDIERR:	Breakpoint Does Not Exist For This BreakId.
22	UDIERR:	No More Breakpoints. BreakId Too High.
23	UDIERR:	TIP Does Not Support The Requested Service.
24	UDIERR:	Error Occurred. Trying Again.
25	UDIERR:	IPC Limitation Exceeded.
26	UDIERR:	Service Incomplete. More Data Available.
27	UDIERR:	Aborted Requested Service.
28	UDIERR:	Transaction Completed.
29	UDIERR:	Cannot Accept.
30	UDIERR:	Transaction Input Needed.
31	UDIERR:	Transaction ModeX.
32	UDIERR:	Invalid Object Size Specified.
33	UDIERR:	Bad Entry In UDI Config File Found.
34	UDIERR:	Internal Error Occurred In IPC Layer.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z