MDI for MAJIC

User's Guide

Embedded Performance, Inc.

P/N 0380-0238-10 Rev 02

March, 2003

EPI has made every attempt to ensure that the information in this document is accurate and complete. However, EPI assumes no responsibility for any errors, omissions, or for any consequences resulting from the use of the information included herein or the equipment it accompanies. EPI reserves the right to make changes in its products and specifications at any time without notice.

Any software described in this document is furnished under a license or non-disclosure agreement. It is against the law to copy this software on magnetic tape, disk, or other medium for any purpose other than the licensee's personal use.

Embedded Performance, Incorporated
606 Valley Way
Milpitas, California 95035
USA
Voice: (408) 957-0350
FAX: (408) 957-0307
email: sales@epitools.com
support@epitools.com
WWW: http://www.epitools.com

Acknowledgments:

Table of Contents

1 About this Manual

This is the user manual for the Embedded Performance Meta Debug Interface (MDI) library for EPI's MAJIC and MAJIC^{PLUS} debug probes. It is aimed at end-users who intend to use the MAJIC with any debugger that supports the MDI specification.

Note: Except where explicitly stated to the contrary, the term MAJIC refers to all models in the MAJIC series. MAJIC^{MX} and MAJIC^{PLUS} refer only to those specific models.

Developers who wish to implement an MDI-compliant debugger and/or an MDI library should refer to the MDI specification document.

Contents of this Manual

About this Manual	Contents of the manual and notational conventions
Introduction	Overview of the MDI
Installation	Installation and configuration information
Interoperability	Detailed information about the MDI for MAJIC implementation

Notational Conventions

The following conventions are used in the syntax descriptions of this manual.

Bold face	Bold identifies characters that must be entered exactly as shown.
Italic	Indicates a general category of input described in detail in the command operand's section.
[ ]	Square brackets enclose an optional operand or group of operands. The brackets are not entered in the command.
{ }	Curly braces are used for grouping purposes. These are not entered in the command. They either enclose a list of alternatives, one of which must be chosen, or they enclose a group of operands that are to be taken together in the context of a list of alternatives or a subsequent repetition.
...	Ellipsis (three dots in succession) indicate a preceding operand, or group of operands if enclosed by [ ] or { }, may optionally be repeated one or more times.
|	A vertical bar indicates an operand, or group of operands if enclosed by [ ] or { }, on either side of the bar may be entered, but not both.

2 Introduction

This chapter provides a brief overview of the Meta Debug Interface API and the MDI library provided by EPI.

What is MDI?

The Meta Debug Interface (MDI) is an Application Programming Interface (API) that defines a standard set of data structures and functions that abstract hardware for debugging purposes. Having a standard "meta" interface allows debuggers and “debug agent” tools (ROM resident debug monitors, ICEs, JTAG probes etc.) from different vendors to work together.

The initial MDI specification was jointly developed by EPI and LSI Logic Corporation. EPI makes the MDI specification freely available, and welcomes its adoption by any interested vendor. While the first implementations targeted MIPS processors, the specification is architecture-neutral, so it can easily be adapted to other architectures.

The MDI API is implemented as a shared library (Windows DLL file or Unix `.so` file), called an MDILib. The MDILib implements the standard MDI API on top of the vendor-specific mechanism for communicating with the actual debug agent which controls the target system. As such, the MDILib is normally provided by the vendor of the debug agent tool that provides target access and control services (typically an ICE, debug monitor, or simulator).

Similarly, the debugger typically supports MDI via a layer of code that implements the debugger’s proprietary API or communications protocol by making calls on the functions exported by the MDILib. This “translation layer” may be included within the debugger executable or it may be in a separate module.

MDI Versions

Over time, EPI may publish updated versions of the MDI specification document. If there is any functional change to the API, the MDI version will be updated. The MDI API includes a mechanism for the debugger and library to negotiate the actual MDI API version that will be used, and it is intended that a debugger or library should be able to support older versions of the API as well as the version that was current when it was implemented.

Any debugger or library claiming MDI compliance should list the range of MDI API versions with which it is compatible. If the supported version range of the debugger and library overlap at all, they should work correctly together. As of this writing, the only released version of the MDI API is 1.0, and the EPI MDI library supports this version. Please check the file **`./mdi/readme.txt`** for the latest information on MDI versions supported.

Note: The specification document itself also has a version number (currently 1.0), but it is the API version that matters when checking for interoperability.

MIPS Debug Interface

MIPS Technologies Inc. (MTI) has used version 1.0 of the Meta Debug Interface specification document as the basis for an API called the “MIPS Debug Interface”. To the best of our knowledge, MTI made no changes (other than the name) in the initial version of their specification, so as of this writing debuggers implementing at least the initial release of the “MIPS Debug Interface” should also work with libraries supporting version 1.0 of the Meta Debug Interface API, and vice-versa. However, future versions of the MTI specification may include extensions that make it incompatible with the Meta Debug Interface.

3 Installation

This chapter describes how to install and configure the Meta Debug Interface API library provided by EPI.

Getting Started

1. Read the MAJIC User's Manual, and follow the steps described in Chapters 2 and 3 to install and configure the MAJIC for your target, and confirm proper operation. If you are using a Unix host computer, it is recommended to install the Windows software at least temporarily in order to utilize the included setup wizard to assist with initial setup.

2. Consult your debugger manual for details on starting up the debugger and connecting to an MDI library. In particular, it is important to learn whether the debugger requires the MDI library file to be located in a particular directory, and whether you can configure the actual file name it will use when loading the library.

3. Re-run the setup wizard, this time selecting either "MDI compliant debugger" or "GDB" in the "Choose your debugger" dialog. If your Windows debugger supports it, you should accept the default location for the configuration files (the **`./bin`** installation directory). If you are using a Unix debugger, you should specify an empty directory since you will be copying the generated files to your Unix host. Otherwise, specify the directory required by your debugger.

4. For Unix hosts, copy the configuration files generated by the setup wizard (**`epimdi.cfg`** and **`startice.cmd`**) to the **`./bin`** EPI installation directory. The pathname of this directory will need to be added to your **`PATH`** environment variable. If your debugger supports specifying the pathname of the **`mdi.so`** shared library file, configure it to load **`./bin/mdi.so`**. Otherwise, either copy **`mdi.so`** to the directory where your debugger expects to find it or use the appropriate Unix utility (**`ldconfig`** on Linux) to add it to the default shared library search list.

5. For Linux hosts, make sure you have the necessary system libraries installed on your system. **`mdi.so`** requires **`glibc`** version 2.1 or later,

which should be no problem unless your Linux distribution is very old (e.g. Red Hat 4.x or earlier. It also requires **`libtermcap.so`**, which is an optional package (e.g. **`libtermcap`** in RedHat or **`termcap`** in SuSE) that may not have been installed on your system when it was set up.

Files

The following files relating to the MDI API are provided in the directory where your EPI software is installed:

`./mdi/readme.txt`
An ASCII file containing last minute updates to the MDI documentation.

`./mdi/mdi.dll` (Windows) or **`./mdi/mdi.so`** (Solaris/Linux)
The library to be loaded by your debugger. A copy of this file is also installed in the **`./bin`** directory. This file must be located where your debugger can find it. Usually this means the directory where the debugger program is located, or any directory on your **`PATH`**. Alternatively, your debugger may allow you to specify the full path name of the MDI library file, in which case it can be run from the EPI installation directory (recommended). See your debugger's documentation for more details.

`./mdi/epimdi.cfg`
A sample MDI configuration file. This file must be customized for your development environment, and it must be located where it can be found by **`mdi.dll`** (see section "The MDI configuration file" on page 7). You can use the MAJIC Setup Wizard to create the customized file or you can edit it manually with any ASCII text editor.

`./mdi/mdispec.pdf`
The MDI Specification, which describes the API in great detail. It is provided for informational purposes, users do not need to read this document to use an MDI compliant debugger and library.

`./mdi/mdi.h`, **`./mdi/mdimips.h`**, **`./mdi/mdiload.c`**
C source code, headers for the MDI API and sample debugger code to load an MDI library. These files are of interest only to those actually implementing an MDI library or an MDI compliant debugger.

`./manuals/mdiuser.pdf`
The MAJIC MDI User's Guide (this document). If you chose not to include the manuals during installation, then this file can still be accessed directly from the **`./manuals`** directory on the installation CD.

`./samples/majic/*/startice.cmd`
Sample startup command files to configure the MAJIC for some common reference platforms. A startup command file must be customized for your environment (see section "The startup command file" on page 11). These files are not specific to the MDI library. They are needed regardless of whether you are using an EPI debugger or a third-party debugger.

The MDI configuration file

The MDI specification requires the debugger to query the MDI library for a list of "devices" that are available, and to give the user a way to select which device to connect to. In the case of the MDI for MAJIC, there is also the possibility that there is more than one physical MAJIC probe available to connect to. The MDI configuration file (**`epimdi.cfg`**) provides the necessary device and probe identification information to the MDILib, so that it can respond to the debugger's query request.

When the debugger first connects to the MDI library, the library will try to open the **`epimdi.cfg`** file by looking first in the current working directory, then in the directory containing the library (Windows version only), then in the directories specified in your **`PATH`** environment variable. It is usually most convenient to have a single **`epimdi.cfg`** file located in the same directory as the library.

For most users, who have only a single type of target system and a single MAJIC probe, the setup wizard will generate a correct **`epimdi.cfg`** file automatically. The rest of this section is of interest to users who have multiple targets and/or multiple MAJIC probes, and therefore need to edit the **`epimdi.cfg`** file to include information about all of them.

Contents

The MDI configuration file is an ASCII file containing comments (C++ style) and keyword-value pairs. Keywords are not case-sensitive. There are two types of values, strings and numbers. Strings must be enclosed in double quotes (**`"`**), unless they are valid identifiers (start with alphabetic or underscore, and contain only alphanumerics and underscore). Numbers and quoted strings conform to C/C++ language syntax.

The sample configuration file uses line breaks and indentation to improve readability, but these are not required. As long as there is at least one whitespace character (space, tab, or line break) between all keywords and values the file will be processed correctly.

Organization

The configuration file is organized into sections, where each section defines **`Global`** settings, a **`Device`**, a **`Controller`**, or the **`MDIDeviceList`**. Conventionally, the **`Global`** section is first, followed by all the **`Device`** sections, followed by **`Controller`** sections and finally the **`MDIDeviceList`** section. But the only requirement is that the **`MDIDeviceList`** section follow all the **`Device`** and **`Controller`** sections.

Global

The **Global** section defines values for configuration settings that are not associated with a particular target device or connection. This section is optional, but it is strongly recommended to include it and specify the EDTPath value. The syntax is:

```
Define Global
        EDTPath         EDTPathString
        LogFile         LogPathString
        CommandFile     CmdPathString
```

EDTPathString is the full path name of the directory into which your EDT package was installed. It is used to locate various files needed by the MDILib without having to add EPI specific directories to the PATH environment variable.

LogPathString is the full or relative path name of a file which will have detailed MDI session log information and console output written to it. This entry should only be used when requested by EPI Technical Support.

CmdPathString is the full or relative path name of a default startup command file that will be processed automatically when a device is opened which does not specify a more specific **CommandFile** value in its **MDIDeviceList**, **Device**, or **Controller** definitions.

The **CommandFile** entry is optional in every definition section, but at least one applicable value is required for every *DevNameString* defined in the **MDIDeviceList** section. If multiple values are available, the order of precedence is the value specified in the *DevNameString* definition itself, followed by the value specified in the referenced **Device** definition, followed by the referenced **Controller** definition, followed by the **Global** definition.

CommandFile is normally specified in the **Device** definition, since there are commonly different configuration commands needed for different devices. Specifying a default in the **Global** definition is useful if you have several devices sharing a common startup command file.

Device

Each **Device** section defines a target device and includes the CPU type, memory organization, and the various identification strings required by the MDI specification. If you have more than one target type with different CPU types or memory organizations, you must provide multiple **Device** sections to define them. The syntax is:

```
Define Device DevIDString
        Family          FamilyString
        Class           ClassString
        ISA             ISAString
        Part            PartString
        Vendor          VendorString
        VendorFamily    VendorFamilyString
        VendorPart      VendorPartString
        VendorPartRev   VendorPartRevisionString
        VendorPartData  VendorPartDataString
        Endian          { Big | Little }
        EPIcpuid        CPUString
        CommandFile     CmdPathString
```

DevIDString is an arbitrary string value used to identify the **Device** section when it is referenced from the **MDIDeviceList** section. *CPUString* is a string value giving the EPI-specific CPU type identifier. This is the same value specified by the **-v** command line option when running an EPI debugger (see the MAJIC User's Manual for more information). The **Endian** value specifies the target system's memory organization (big-endian or little-endian).

CmdPathString is the full or relative path name of a startup command file that will be processed automatically when a device is opened which does not specify a more specific **CommandFile** value in its **MDIDeviceList** definition. If **CommandFile** is specified here, the value overrides values specified in the **Global** and **Controller** definitions.

All of the other values are string values that are passed to the debugger to identify the CPU type. The intent of the MDI specification is that each chip vendor will document standard values for these strings for their parts. That may or may not happen, but EPI's MDI library makes no use of these values. You can set these values to whatever strings your debugger is expecting, if any. Otherwise the values are arbitrary.

Note: Only the **Endian** and **EPIcpuid** values are required to be present in a **Device** section. The **Family** and **Class** values are set to the MDI specified strings by default, so they can always be omitted. The rest are set to the string **"NotSet"** by default, and can be omitted unless your debugger expects them to be set to a particular value (which it must document). For example, the **mdi-server** GDB interface program needs the **ISA** value to be set correctly for MIPS targets.

Controller

Each **Controller** section defines a particular MAJIC to connect to and includes the communication port (serial port or Ethernet host name), baud rate, and the startup command file to load for initialization. If you have more than one MAJIC probe, or more than one startup initialization file, you must provide multiple **Controller** sections to define them. The syntax is:

```
Define Controller ControllerIDString
        Port          PortString
        Speed         SpeedNumber
        CommandFile   CmdPathString
```

ControllerIDString is an arbitrary string value used to identify the **Controller** section when it is referenced from the **MDIDeviceList** section. *PortString* is a string value giving the serial port or Ethernet hostname to use to connect to the MAJIC probe. This is the same value specified by the **-d** command line option when running an EPI debugger (see the MAJIC User's Manual for more information). *SpeedNumber* is the EPI-specific baud rate value to use when *PortString* specifies a serial interface. Valid values are **0** (1200 baud) through **7** (115,200 baud).

CmdPathString is the full or relative path name of a startup command file that will be processed automatically when a device is opened which does not specify a more specific **CommandFile** value in its **MDIDeviceList** or **Device** definition. If **CommandFile** is specified here, the value overrides a value specified in the **Global** definition.

MDIDeviceList

The last section in the configuration file is the **MDIDeviceList** section. It defines the MDI device name strings that the debugger will display to the user, and associates each name with a specific **Device** and **Controller** section. The syntax is:

```
Define MDIDeviceList
        {  DevNameString
                Device      DevIDString
                Controller  ControllerIDString
                CommandFile CmdPathString
        }....
```

If you have more than one **Device** or **Controller** section, the **MDIDeviceList** section will contain multiple sets of *DevNameString* + **Device** + **Controller** entries to list and name all the valid combinations of **Device** and **Controller**.

DevNameString is a string value to be passed to the debugger to identify a particular "MDI Device" that the debugger can connect to. If there is only one *DevNameString* entry in the **MDIDeviceList** section, the debugger may just automatically open the device. Otherwise, the debugger should present you with a list of the *DevNameString* values and let you select which one to connect to.

CmdPathString is the full or relative path name of a startup command file that will be processed automatically when the device is opened. If **CommandFile** is specified here, the value overrides values specified in the **Global**, **Controller**, and **Device** definitions.

The startup command file

The MAJIC has a number of configuration options that must be set correctly for the target system. These options are set by debugger commands, and EPI debuggers automatically look for a command file named **startice.cmd** at start up. See the MAJIC User's Manual for information on this topic.

When using a third-party debugger with the EPI MDI library, the name of the startup command file is specified in the MDI configuration file. A full path name can be provided, or a relative path name or just the file name. If the full path is not specified, the MDI library will try to open the file by looking first relative to the current working directory, then the directory containing the library (Windows version only), then the `bin` subdirectory of the EDT installation directory if **EDTPath** is specified, then the directories specified in your **PATH** environment variable. The directory containing the startup command file will also added to the list of directories to search for other files.

Troubleshooting

There are two troubleshooting techniques you can use if you are having problems using an MDI-compliant third party debugger with the MAJIC.

The first is to try to reproduce the problem using the MONICE debugger provided by EPI. This eliminates the third party debugger and its possibly non-compliant use of the MDI API from the equation. Make sure that MONICE is loading the same **`startice.cmd`** file as the MDILib so the initial configuration will be the same. If you have similar symptoms when using MONICE, then it must be a MAJIC issue (most likely a configuration problem) rather than an MDI issue. Refer to the MAJIC User's Manual for help.

If the MAJIC appears to be operating properly when using MONICE, the next question is whether the culprit is the MDILib provided by EPI or the debugger itself. This may not be easy to determine from the visible symptoms, so the MDILib includes a logging facility that can record all of the MDI operations requested by the debugger and their results. The log file will provide the detailed information that EPI customer support will need to see exactly what is going wrong with the debug session.

To enable MDI session logging, define an environment variable named **`MAJICLOG`** whose value is the pathname of the file which will contain the log output. It is also possible to set **`MAJICLOG`** to "**`console`**", in which case the log output will be sent to the debugger via its MDICBOutput() callback function, but generally logging to a file is preferable.

4 Interoperability

The MDI specification includes both required and optional services. Any debugger or library implementation claiming MDI compliance must support all required services and document the optional services it supports. This chapter provides this information and other details about EPI's MDILib implementation for MAJIC that may affect interoperability with MDI-compliant debuggers.

Required Services

EPI's MDILib implements all required MDILib services. However, there are a few cases where the MDI specification leaves aspects of the implementation of a required service up to the MDILib or where EPI's implementation does not exactly match the specification. These cases are detailed below.

Most services defined by the MDI specification are provided by functions in the MDILib that are called by the debugger. There are also a few services that the debugger is required to provide for use by the MDILib, via "call-back" functions. We list the ones that EPI's MDILib actually uses.

MDICacheFlush

This function allows the debugger to request that the processor's Instruction and/or Data caches be flushed and/or invalidated. EPI's MDILib supports this service by passing the request on to the MAJIC. But what the MAJIC does with it can vary depending on the capabilities of the actual CPU. For example, it may not be possible to flush the cache without also invalidating it.

MDIRunState

This function returns the current status of the target system (running, hit breakpoint, etc.) to the debugger. The MDI specification requires the debugger to call MDIRunState() frequently when the target is running. It also *recommends* that the debugger call MDIRunState() frequently even when the target is not running.

The MAJIC may occasionally send status or event notification messages when the processor is not running, such as when target power is turned on

or off. So EPI's MDILib does depend on the debugger calling MDIRunState() when the target is not running, to process these events in a timely manner. If the debugger does not make these calls, notifications may be delayed until the next MDI service is requested or can even be lost entirely.

MDIRead

This function is called by the debugger to read the contents of memory and registers. The debugger passes an address, an object size, and a count of the number of objects to read. An address consists of an offset and a "resource". Resources are values that identify specific memory and register address types, and are architecture-specific. MDILib support for some resources is optional. To allow the debugger to determine whether a particular resource is supported, the MDI specification assigns special meaning to MDIRead() calls with a count of zero. When the requested count is zero, the MDILib is required to just check the address and return a success or error status, based on whether the address is valid and supported.

Currently, the MAJIC does not provide a way to directly query whether an address is valid for the current target. Instead, it generates an error message when a transfer to an invalid or unsupported address is actually attempted. So EPI's MDILib uses built-in knowledge of the CPU type to determine whether to return a success status for MDIRead() calls with a count of zero. This works well enough for typical usage, like whether or not there are floating point registers or cache tag registers. But there are cases, like cores that allow optional coprocessors to be added, where the MDILib can not know whether the target has the corresponding registers or not. In such cases, it will return success to the zero count query, but actual reads and writes may fail after displaying an error message.

MDIDoCommand

This function is called by the debugger to cause the MDILib to execute an ASCII command using its internal command interpreter. Recognizing that no debugger API can possibly abstract all possible features of all possible debug tools, the MDI specification allows the MDILib to provide additional functionality via a command interpreter. MDIDoCommand() is unusual in that it is an *optional* service, but if it is provided by the MDILib then the debugger is *required* to use it. That is, the debugger must provide a way for users to enter arbitrary command lines that the debugger will pass to MDIDoCommand().

EPI's MDILib does implement an extensive command language, and therefore relies on the debugger to support MDIDoCommand(). Nearly all the MONICE commands described in the MAJIC User's Manual are available in the MAJIC MDILib as well, with the notable exceptions being breakpoint and execution commands.

MDICBInput, MDICBOutput

These are required "call-back" functions provided by the debugger. MDICBOutput() allows the MDILib to pass strings that the debugger must display to the user, while MDICBInput() allows the MDILib to get keyboard input from the user.

EPI's MDILib relies on MDICBOutput() to display informational and warning messages, target program output generated via the EPI-OS or

Semi-hosting features supported by MIPS and ARM MAJIC, and output generated by commands passed to MDIDoCommand(). It also relies on MDICBInput() to get input for the target program, interactive commands passed to MDIDoCommand(), and sometimes responses to error or warning message prompts. If the debugger does not provide these required services, it is not MDI-compliant and it will not be able to connect to EPI's MDILib successfully.

Optional Services

The MDI specification defines a number of optional services that an MDILib may support, and some required services have optional aspects. MDILibs are required to document what they actually implement for all optional behavior. These cases are detailed below for EPI's MDI for MAJIC.

MDIOpen

This function is called by the debugger to establish a connection to a particular target device. It must be called before any debug services (read, write, execute, etc.) can be performed. The debugger passes a parameter indicating whether it wants exclusive or shared access to the device. To enable various types of multi-processor debugging, the MDI specification permits one or more debuggers to open multiple devices at the same time, and even to open the same device multiple times. However, since the capabilities of debug tools varies widely, an MDILib is not required to support multiple simultaneous connections.

EPI's MDILib does not support multiple simultaneous connections, so once the debugger calls MDIOpen() to connect to a device, any further calls will return an error until MDIClose() is called to close the original connection. However, this does not mean that there is no way to do multiprocessor debugging with the MDILib and MAJIC. The MAJIC itself supports debugging multiple processors on the same JTAG chain, by allowing the host to open a separate communications channel for each device. If your MDI-compliant debugger does not directly support multiprocessor debugging, this is exactly what happens when you run multiple copies of the debugger, each of which loads a copy of the MDILib.

If your debugger directly supports multiprocessor debugging, you should be able to configure it to load a different MDILib for each processor. This is required because you may be using different probes, and therefore different MDILib implementations, for each processor. To support multiprocessor debugging with a single debugger and one or more MAJICs, you need to make a copy of the MDILib library with a different name for each processor. Then configure your debugger to load a different copy of the MDILib for each connection.

Target Groups

The MDI specification includes an abstraction for the concept of having multiple target devices treated as a group. If connections are made to multiple target devices in the same group, execution can be started and stopped on all of them with a single service request. Support for the Target Group services is optional.

Since EPI's MDILib does not support connecting to more than one target device at a time, it also does not support Target Group services.

Target access while running

The MDI specification allows the debugger to make calls on MDI service functions even while the target is executing code. But for most services, it also allows the MDILib to return an error status if it does not support the particular service while the target is running.

EPI's MDILib supports these concurrent operations only if the MAJIC supports them, which depends on the MAJIC firmware version installed. If the MAJIC supports concurrent operations, it will temporarily halt the processor long enough to perform the service if necessary, but will avoid doing so whenever possible. For example, some processors support a method of memory access that does not require halting the processor.

MDIFind

This function is called by the debugger to search for a value or pattern. The MDI specification requires that the MDILib support this service for patterns up to 256 objects long, but only for memory (as opposed to registers). Since an object can be up to eight bytes in size, a pattern can be as long as 2048 bytes with an optional mask of equal size.

EPI's MDILib supports MDIFind() only for memory address ranges, not registers. Also, MAJIC limits the total length of the search pattern to about 1400 bytes (700 bytes when masked). If this limit is exceeded MDIFind() returns a "not found" status. For object sizes of four and eight bytes, this is a technical violation of the MDI specification since we support less than the required pattern length of 256 objects. But this is *very* unlikely to be a problem in practice. Most debuggers only support searching for a single value, so our worst-case search pattern limit of 88 masked doublewords should be more than adequate.

MDICacheQuery

This function may be called by the debugger to retrieve the attributes of the caches present on the target device, if any. In the MDI specification, support for this function is optional.

EPI's MDILib uses the MAJIC cache configuration options **`trgt_cache_type`**, **`trgt_icache_*`**, and **`trgt_dcache_*`** to provide the requested information. Whether these options are available depends on the specific processor type being used. If they are not available, MDICacheQuery() returns no information. See the MAJIC User Manual for more details.

MDIReset

This function is called by the debugger to reset the target device. The MDI specification defines four types of reset operation that can be requested: MDIFullReset (reset entire target system, processor and board), MDIDeviceReset (reset processor, including peripherals in SoC devices), MDICPUReset (reset processor only), and MDIPeripheralReset (reset peripherals only in SoC devices). Support for multiple types of reset is optional for both the debugger and the MDILib.

The MAJIC provides flexible support for reset operations. It supports two types of reset requests, "reset target" and "reset processor", and their behavior can be configured via the **`ice_reset_output`** and **`ice_reset_peripheral`** configuration options. See the MAJIC User's Manual for details. The MDILib maps the MDI reset types as follows: If the debugger requests MDIFullReset, the MDILib performs a target reset. MDIDeviceReset and MDICPUReset both cause the MDILib to perform a processor reset. Since the MAJIC does not provide a mechanism for resetting peripheral logic without also resetting the CPU, EPI's MDILib does nothing if an MDIPeripheralReset is requested.

MDISetBp

This function is called by the debugger to set hardware and software breakpoints and triggers. The MDI specification provides a fairly ambitious abstraction for breakpoints, including permanent and temporary software (standard) breakpoints, and hardware breakpoints on instruction execution, data load and/or store access, or bus transaction. Hardware breakpoints can include an address range, data value, and a mask for the data value. They can also be specified to generate a trigger signal as well as or instead of halting the processor.

Obviously, the actual ability of debug systems to support all these types of breakpoints can vary widely. An MDILib is only required to support the software breakpoint types, all hardware breakpoint support is optional.

All of the breakpoint options are supported by EPI's MDILib in the sense that they are passed on to the MAJIC. The MAJIC, in turn, supports all of the hardware breakpoint and external trigger capabilities of the particular processor. See the MAJIC User's Manual for more details.

Trace services

The MDI specification provides a set of optional services that can be used by the debugger to enable the capturing of execution history trace data (instructions executed and possibly data accesses), and to fetch the captured data for display. When used with a MAJIC^{MX} or MAJIC^{PLUS}, EPI's MDILib supports all MDI Trace services.

MDICBPeriodic

This is an optional "call-back" function that may be provided by the debugger. If provided, the MDI specification requires that the MDILib call this function at least once every 100 milliseconds while processing a long-running MDI function, to give the debugger an opportunity to process user interface events. The specification does not specify a limit on how frequently MDICBPeriodic may be called.

If it is provided, EPI's MDILib actually calls the MDICBPeriodic function much more frequently than the required 100 milliseconds, and will call it at least once even during services that are completed in less than 100 milliseconds.

MDICBLookup, MDICBEvaluate

These are optional "call-back" functions that may be provided by the debugger. MDICBLookup() allows the MDILib to ask the debugger what symbol or source line corresponds to an address, if any. MDICBEvalute() allows the MDILib to ask the debugger to evaluate an expression using the debugger's rules, and return a value or address. These callbacks are intended to be useful to MDILibs that implement a command language, since they may not have access to symbolic information for the program being debugged.

Currently, EPI's MDILib does not call MDICBEvaluate(). It will use MDICBLookup() if it is available.

Command Interpreter

EPI's MDILib for MAJIC implements the MON command language interpreter in order to provide full access to all the capabilities of the MAJIC, not just the subset that is supported by the MDI API and the debugger.

As described in the Required Services section, the debugger is required by the MDI specification to provide a way for you to enter commands and data to, and see the output from, the MDILib's command interpreter. When using GDB and **`mdiserver`**, the GDB **`monitor`** command (which can be abbreviated as **`mon`**) can be used to execute MON commands.

The MDILib command interpreter supports nearly all of the MONICE command language, as documented in the MAJIC User Manual. The exceptions are the commands associated with program loading and execution control (load/verify load, go/step/stop/calls, and breakpoint commands). All data display and enter commands, including trace data display, reset commands, and configuration commands are available. As are the command file and alias commands that allow the command language to be extended.

Note: If the MDILib can not find the **`./bin`** EPI installation directory, then the MON Help command will not work and some register definitions may be missing. Specifying the **`EDTPath`** value in the **`Global`** section of the **`epimdi.cfg`** file is recommended for this reason. For Windows and cygwin environments, the directory will be found if the **`mdi.dll`** or **`mdi.so`** file is being loaded from there. Otherwise, the EPI **`./bin`** directory must be added to your **`PATH`**.

Note: As of version 1.0, the MDI specification does not provide a reasonable way for an MDILib to ask the debugger to return the address associated with a symbol name. So, unfortunately, the only symbol names that can be used in MDILib commands are the built-in register names and any names you have defined with previous Enter Name (**`en`**) commands.