Microprocessor Debug Interface (MDI) Specification

Document Number: MD00412
Revision 02.41
September 22, 2009

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

Template: nB1.03, Built with tags: 2B

Table of Contents

Chapter 1 Overview 5
1.1 Abstract 5
1.2 MDI Organization 5

Chapter 2 Terms 7

Chapter 3 Principles of Operation 9
3.1 Multi-thread Debugging 9
3.2 Multi-processor Debugging 10
3.2.1 Multi-processor Teams 11
3.2.2 Disabled Multi-processor Devices 14

Chapter 4 MDI Environment Command Set 15
4.1 Version: Obtain the supported MDI versions for this MDILib implementation 15
4.2 Connect: Establish a connnection to the MDILib 15
4.3 Disconnect: Disconnect from the MDILib 17

Chapter 5 Target Group Command Set 19
5.1 Target Group Query: Retrieves the names of the defined target groups. 19
5.2 Target Group Open: Opens a target group 20
5.3 Target Group Close: Close a previously opened target group 21
5.4 Target Group Execute: Place in execution mode the appropriate devices in the target group 21
5.5 Target Group Stop: Stop execution for all appropriate devices in the target group 22

Chapter 6 Device Command Set 23
6.1 Session Control 23
6.1.1 Device Query: Retrieves information about the devices 23
6.1.2 Open: Opens a device. 24
6.1.3 Close: Closes a device. 25
6.1.4 Process Events: Callback function to process periodic events 26
6.1.5 Synchronize State: Callback function to synchronize device state changes 26
6.2 Resource Addresses 27
6.3 Resource Access 27
6.3.1 Read: Reads a contiguous range of data from the specified resource on the device. 28
6.3.2 Write: Writes a contiguous range of data to the specified resource on the device. 29
6.3.3 Read List: Read a set of values 30
6.3.4 Write List 31
6.3.5 Move: Move data from one resource to another on the device 31
6.3.6 Fill: Fill the specified resource on the device with a pattern. 32
6.3.7 Find: Find a pattern in a resource 33
6.3.8 Query Cache: Retrieve cache attributes 34
6.3.9 Get Cache Details: Get Information about the Specified Cache 35
6.3.10 Cache Flush: Write back and/or invalidate the cache 35
6.3.11 Cache Operation: Do Specified Operation on Specified Cache 36
6.3.12 Cache Sync: Synchronize the caches 37
6.4 Run Control 37
6.4.1 Execute: Place the device into its RUNNING state 38
6.4.2 Step: Single steps the device 38
6.4.3 Stop: Stop execution of the device 39
6.4.4 Abort: Terminate the current MDI function 40
6.4.5 Reset: Performs a target reset operation 40
6.4.6 State: Returns the current device execution status. 41

6.5 Breakpoints 43
6.5.1 Set Full Breakpoint 45
6.5.2 Set Software Breakpoint 46
6.5.3 Clear Breakpoint 47
6.5.4 Enable Breakpoint 47
6.5.5 Disable Breakpoint 48
6.5.6 Query Breakpoints 48
6.5.7 Hardware Breakpoint Query: Retrieve a list of supported hardware breakpoint types 49

Chapter 7 MDILib and Target I/O Command Set 53
7.1 Execute Command: Do the command specified 53
7.2 Display Output: Display the MDILib supplied text to the user 53
7.3 Get Input 54
7.4 Evaluate Expression 55
7.5 Lookup Resource 56

Chapter 8 Trace Command Set 59
8.1 Enable Tracing 59
8.2 Disable Tracing 60
8.3 Clear Trace Data 60
8.4 Query Trace Status 61
8.5 Query Trace Data 61
8.6 Read Trace Data 62
8.7 Read PDtrace Data 64
8.8 Get PDtrace Mode 65
8.9 Set PDtrace Mode 67
8.10 Get TCB Trigger Information 67
8.11 Set TCB Trigger Information 68

Chapter 9 Multi-Threaded and Multi-Processor Command Set 71
9.1 Multi-Thread Control 71
9.1.1 Set Thread Context: Sets the current MDI thread context ID 71
9.1.2 Get Thread Context: Returns the current MDI thread context ID 72
9.1.3 Thread Context Query: Retrieves a list of active TCs 72
9.2 Set Run Mode: Specify behavior when returning to the RUNNING state 73
9.3 Multi-processor Team Control 74
9.3.1 Create Team: Create a new multi-processor debugging team 75
9.3.2 Teams Query: Retrieves a list of active teams 75
9.3.3 Clear Team: Removes all members from a multi-processor team 75
9.3.4 Destroy Team: Destroys a multi-processor team 76
9.3.5 Attach Team Member: Add a new member to a team 76
9.3.6 Detach Team Member: Remove a single member from a team 77
9.3.7 Team Member Query: Retrieves a list of team members 78
9.3.8 Team Execute: Place all team members into RUNNING state 79

Chapter 10 Complex Break and Trigger and StopWatch Timer Command Set 81
10.1 Set a Priming Conditon for the Specified Complex Breakpoint 81
10.2 Get the Priming Conditon for the Specified Complex Breakpoint 81
10.3 Query Complex Breakpoint and StopWatch Configuration Options 82
10.4 Get the Current Value of the StopWatch Timer 83
10.5 Clear the Value of the StopWatch Timer 83
10.6 Set the Mode of the StopWatch Timer 83
10.7 Get the Mode of the StopWatch Timer 84

Appendix A MDI.h Header File 85

Appendix B Example Code to Setup an MDILib Connection 97

Appendix C An MDI Addendum for MIPS32® and MIPS64® Architectures 105
C.1 Abstract 105
C.2 MIPS MDIDDataT Fields 105
C.3 MIPS Exception Codes 105
C.4 MIPS16e Instructions 105
C.5 MIPS Resources 105
C.6 MIPS-Specific Breakpoint Implementation 109
C.6.1 MDISetBP() and MDISetSWBp() Function Calls 109
C.6.2 Implementation of MDISetSWBp() 109
C.7 MIPS Specific Header File 109

Appendix D MDI_PDtrace.h Header File 113

Appendix E mdi_tcb.h Header File 117

Appendix F Revision History 121

Chapter 1

Overview

1.1 Abstract

The main goal of Microprocessor Debug Interface (MDI) is to define a set of data structures and functions that abstract hardware (or hardware simulators) for debugging purposes. Having a standard "meta" interface allows development tools (debuggers, debug kernels, ICEs, JTAG probes etc.) from different vendors to inter-operate. A secondary goal of the MDI specification is to define a multi-target environment in which multiple hardware abstracts may coexist

1.2 MDI Organization

MDI is divided into 5 command sets. The first set is the MDI environment. These commands establish the initial connection, maintain version control, handle configuration, and support debugger event processing and multiple debugger synchronization. The second command set is the target group commands. A target group is made up of one or more target devices. The target group command set contains commands to query/open/close individual target groups as well as special multi-target commands that control the individual devices as a group. The third command set is the individual target device commands. This set of commands provide the fundamental functions and resources that are needed to debug individual target devices. The fourth command set is the debugger callbacks, functions provided by the debugger. This command set supports MDILib command processing and provides various character I/O services to both the MDI interface and the target application. The fifth command set is the trace data commands. This command set provides a simple interface to the tracing capabilities provided by many target devices. Another command set deals with calls needed to support multi-threading and multi-core or multi-processor targets.

A complete MDI specification consists of two parts: the architecture independent MDI specification (this document), plus an addendum that provides the necessary details for a specific target architecture. This document includes the addendum needed by the MIPS32® and MIPS64® architectures in the Appendix.

Chapter 2

Terms

The following terms are used throughout this document:

- MDI - This specification, plus the appropriate device specific addendum.
- MDILib - An implementation of the MDI specification providing an interface to one or more devices.
- Debugger - An MDI compliant application that uses one or more MDILibs to access and control one or more devices. Typically, this is a source- or assembly-level debugger, but it could be anything.
- Thread Context (TC) - The hardware state necessary to support a single thread of execution within a multi-threaded CPU device, such as defined by the MIPS MT ASE. This includes a set of general purpose registers, multiplier registers, a program counter (PC) and some privileged state.
- VPE - A virtual processing element (VPE) is an instantiation of the full CPU privileged state on a multi-threaded CPU, sufficient to run an independent per-processor OS image - it can be thought of as a virtual CPU. Each VPE must have at least one TC attached to it in order to execute instructions and be debuggable, but it may contain more than one TC when running an explicitly multi-threaded OS or application. A conventional single-threaded CPU could be considered as implementing a single VPE containing a single TC.
- Multi-processor - A collection of processing elements within a single target system. This may be a set of single-threaded CPUs within a multi-core design, a number of VPEs within a multi-threaded CPU core, or a combination of the two.
- Device - A specific processing element that can be accessed and controlled via MDI. Typically, this is a target board containing a single CPU or DSP, or a simulator. In a multi-processor system, each processing element (CPU or VPE) would be a separate device. The actual mechanism by which an MDILib accesses and controls a device is not addressed by MDI, it is a private implementation detail of the MDILib.
- Target Group - A group of target devices that are capable of being operated on as a group, where the grouping is statically defined by the MDILib.
- Team - A dynamic grouping of devices which stop and start normal execution simultaneously. This allows several debuggers, each debugging a separate but loosely cooperating operating system or program on different devices to safely view and manipulate shared resources (e.g. memory or device state), without any interference from the other *team members*. Alternatively it allows a single debugger to control multiple devices executing a single symmettric multi-processing (SMP) operating system image.

Chapter 3

Principles of Operation

An MDILib is implemented as a dynamically linked library in the Microsoft Win32 environment (mdi.dll) and a shared library in the UNIX environment (mdi.so). In many cases, the caller of an interface function passes a pointer to caller-allocated memory. In all such cases, the caller is required to maintain the validity of the pointer only until the called function has returned.

MDI is designed to allow the MDILib to run synchronously with the debugger. The debugger passes a thread of control to the MDILib by making a call to an MDI function. The MDILib may then use the thread to do maintenance before the requested function is complete. If the processing time for maintenance and the requested function are longer than 100 milliseconds, the MDILib will loan the thread back to the debugger by calling the debugger's MDICBPeriodic routine. At this point the debugger may cancel the current MDI command, update user interfaces or do other debugger maintenance. The debugger then returns the thread to the MDILib by exiting the MDICBPeriodic routine. The thread is then returned to the debugger upon completion or abortion of the original MDI function. The debugger must assume that the MDILib always uses the debugger's thread to execute. It is therefore imperative that the debugger call MDIRunState frequently whenever the device is running, so that the MDILib can be responsive to device events. It is also possible, though less common, that the MDILib may want to be able to process certain device events even when the device is not running. It is therefore recommended that the debugger also call MDIRunState frequently at all times.

Though the actual implementation of a particular MDILib or debugger may be multi-threaded, it is not desirable to burden *all* MDILib implementations with a requirement to be re-entrant; therefore the communications path between debugger and MDILib is defined to be single threaded (synchronous), that is, for a given debugger process the same thread must make all MDILib calls.

The simplest development environment would be a single debugger using a single MDILib to control a single device. In this case, the debugger can be implicitly linked to the standard MDILib library file (`mdi.dll` or `mdi.so`); however, MDI envisions that a complete development environment may include multiple devices, multiple debuggers, and multiple MDILibs, potentially all from different vendors. In this case, each MDILib will necessarily have a unique file name, and the debuggers must provide a way for the actual MDILib file name to be configured, and use explicit linking to load the file and get pointers to its MDI functions at run time. To allow operability in this more complex environment, debugger vendors are strongly encouraged to use explicit linking even if they do not support multi-device debugging.

Note that the MDI specification allows the debugger to call any MDI service function at any time, including while the target program is running. MDILibs are encouraged to support as many services as possible during execution, but not all target environments will be able to support all MDI services while the target device is executing, so the MDILib may return MDIErrTargetRunning in response to most MDI calls if the service can not be performed because of target execution. The debugger vendor should also be aware that MDILibs that do support debugger operations during execution may do so by temporarily interrupting execution to perform the service.

To ease development of debuggers and MDILibs, MDI includes C language header files defining the interface (`mdi.h)` MDILibs must #define MDI_LIB before including `mdi.h` in their source files. Also provided for the Microsoft Win32 environment is `mdi.def`, a linker input file used when building an MDILib DLL, and `mdiload.c`, a C language file providing function MDIInit, which loads the MDILib DLL. The debugger must call MDIInit before using any of the MDI functions. All MDI functions are built using the __stdcall calling convention for the Win32 environment.

3.1 Multi-thread Debugging

Within each processing element of a multi-threaded CPU there may be more than one TC or Thread Context: that is a set of general purpose registers and program counter capable of executing an instruction stream, or *thread*. The CPU can execute instructions from all runnable TCs "simultaneously", or at least apparently so, by interleaving instructions from the TCs through its pipeline at high speed. However TCs are subsidiary to the processing element, and when any TC enters debug mode (e.g. completes a single-step or hits a breakpoint), then all of the other TCs contained within that processing element will be suspended. So TCs are not exposed as a first class Device to which you can connect an MDI debugger - the connection is instead made to the processing element, and additional MDI functions described in Chapter 9, “Multi-Threaded and Multi-Processor Command Set,” on page 71 allow a debugger to determine the list of active TCs, access their registers, and specify their behavior (e.g. remain suspended, single step, or run freely) upon leaving debug mode.

Note that an MDILib controlling a hardware probe or CPU simulator is not expected to be able to debug software threads in a complex operating system where there are more software threads than TCs. In such operating systems the software thread state is being context switched by the OS between hardware TCs and memory-based thread data structures. A hardware debugger does not typically have the OS-specific knowledge that would allow it to interpret the memory-based thread state. So while an MDI debugger can be used to debug the low-level TC management within such an operating system, debugging "application" software threads will typically require the use of an OS-provided "thread aware" remote debug protocol - possibly tunneled through MDI via shared memory - or enhancements to the debugger to make it OS aware by traversing and manipulating the OS’s thread data structures in memory using MDIRead() and MDIWrite(). Both of these techniques are outside the scope of this document.

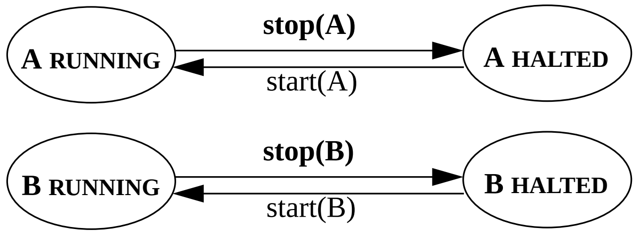
3.2 Multi-processor Debugging

A multi-processor target contains multiple devices, either virtual (VPEs) on a multi-threaded CPU core, true multi-core CPUs, or some combination of the two - i.e. multiple CPU cores, one or more of which may contain multiple VPEs. In all cases the MDILib is required to allow multiple parallel connections to this collection of devices from one or more debuggers simultaneously. In other words one super-debugger may open multiple MDI connections to several devices at once, or there may be several "legacy" single-processor debuggers running in parallel, each connecting to a single, separate device.The MDILib shall provide a unique Target Group/Device name and ID for each device - virtual or physical. As a convenience for single-processor debuggers, an MDILib may coordinate some or all of the devices internally to provide the illusion of a single device with multiple TCs, this is recommended but is not required.

MDI requires that each device appears to the debugger or debuggers to be capable of operating independently of the other devices, i.e. as if they were truly independent CPU cores, even if they are in fact VPEs within the same CPU core. All devices must be capable of being simultaneously in RUNNING state, or HALTED in debug mode and servicing MDI i/o requests, or any permutation thereof. If the hardware implementation does not allow this directly (e.g. if debug mode suspends other VPEs on a multi-threaded CPU), then an MDILib must simulate the required behavior by suspending the device which originally entered debug mode, and then returning from debug mode so that the other devices can run target code or themselves enter debug mode to service MDI calls from other debuggers.

Requiring VPE devices to operate as if they were truly independent cores is important, since it allows parallel debugging of non-cooperating or loosely cooperating separate program images using "legacy" debuggers, most of which can work with only one program image at a time. It also permits the debugging of one VPE while the other VPEs continue to run normally. For example you might be using an MDILib and hardware probe to debug a low-level DSP or data plane task which is running on one VPE, while running a control plane application on a multi-tasking OS on a second VPE, or debugging it using the OS’s standard application debugger. The multi-tasking OS must continue to run uninterrupted even while the signal processing device is halted by the MDI debugger. Figure 3-1 below illustrates the states associated with this form of debugging, where two devices operate completely independently of each other.

Figure 3-1 State Transitions for Independent Multi-Processor Debugging



Actions
stop = breakpoint/single-step exception, or debugger calls MDIStop()
start = debugger calls MDIExecute() or MDIStep()

States
HALTED = debug mode, MDIRunState() returns appropriate "non-running" status
RUNNING = normal execution mode

3.2.1 Multi-processor Teams

When the software running on devices within a multi-processor is more tightly coupled: sharing data structures in memory, or even running a full-blown SMP operating system with a single shared instruction and shared data (SISD) image, then it is useful to be able to dynamically join the devices together into a debugging *team*, such that when any one of them enters debug mode, the others simultaneously stop running. This presents a stable shared memory image to the debugger(s), and permits inspection of the state of all the cooperating processors at the "same" moment in time: in a multi-core system "same" may mean within a few cycles, to allow time for debug interrupts to propagate from one core to another; in a multi-VPE processor all team members within the same CPU should stop running literally simultaneously. A team may include devices which have not been opened by a debugger. A team is also persistent, in that it survives MDI library disconnects, until the last disconnect from the MDILib (or MDILibs) which manage the team.

There are several ways in which an MDI *team* might be used in practice - two examples being as follows:

- When debugging a different program image on each device (e.g. a control program on one, and a real-time DSP or data plane task on the other), then several "legacy" single-processor debuggers may be used in parallel, each debugging a single program image on a single device. But if the user needs to debug low-level hardware interactions between the programs/processors, then it will be helpful if both devices can be forced to stop running simultaneously, whenever one or the other reaches a breakpoint or is forcibly stopped.
- When debugging an high-level SMP (SISD) operating system, a multi-processor aware debugger will open several MDI connections, one for each device, and then join them together into a team so that they stop and start execution simultaneously, simulating a single CPU with multiple thread contexts. The debugger might present each single-threaded device to the user as if it were one thread context within a single virtual CPU, or if any devices contain multiple TCs, then as a unified set of TCs. The debugger will iterate over the open MDI devices to set global breakpoints, execution mode, and so on.

3.2.1.1 Legacy Team Debugging

A conventional single-processor "legacy" debugger will not take kindly to its debuggee spontaneously resuming execution when the debugger thinks it is halted. The MDI *team* concept therefore virtualizes the device's HALTED state, so that each debugger believes that it is totally in control of its device, even though in reality it may be stopping and starting outside of the debugger's control.

Figure 3-2 below illustrates how two devices A and B should behave when they are each controlled by a separate "legacy" debugger, but affiliated within a team. The crucial concept is the FROZEN state which is internal to the MDILib, and not reported to the debugger. The FROZEN state may be implemented by freezing or disabling a device's pipeline whenever a team member enters debug mode, or by linking one element's debug mode output to another's debug interrupt input, but hiding this debug interrupt from its debugger.

The MDIRunState() function shall return MDIStatusRunning whenever a device is not in the HALTED state. It indicates only that the processing element is *capable* of running without further intervention from the debugger. A device may be in the FROZEN state, or multi-threading may have been temporarily disabled by another VPE, or all of its TCs may be idle, or blocked waiting for some hardware event to occur - but in all cases MDIStatusRunning is returned.

When device A stops running target code and enters debug mode because it hits a breakpoint or its debugger calls MDIStop(), then device B is stopped automatically and held in the internal FROZEN state. As far as debugger-B is concerned its processor is still reported to be in the RUNNING state. If device A is instructed to resume execution (i.e. debugger-A calls MDIExecute or MDIStep), then device B is automatically restarted, again without notifying its debugger. Only if device A stops, and then device B is explicitly stopped.(i.e. debugger-B calls MDIStop()), are both reported as HALTED. From that state resuming execution of one or the other leaves its opposite number in the HALTED state, until its debugger tells it to resume execution too.

This behavior extends in the obvious way when more than two devices are present in the system. For example, assume that there are three devices A, B, and C. Assume a state where all devices are currently running. The list below is an example of the state of the three devices and different events that cause those state transitions:

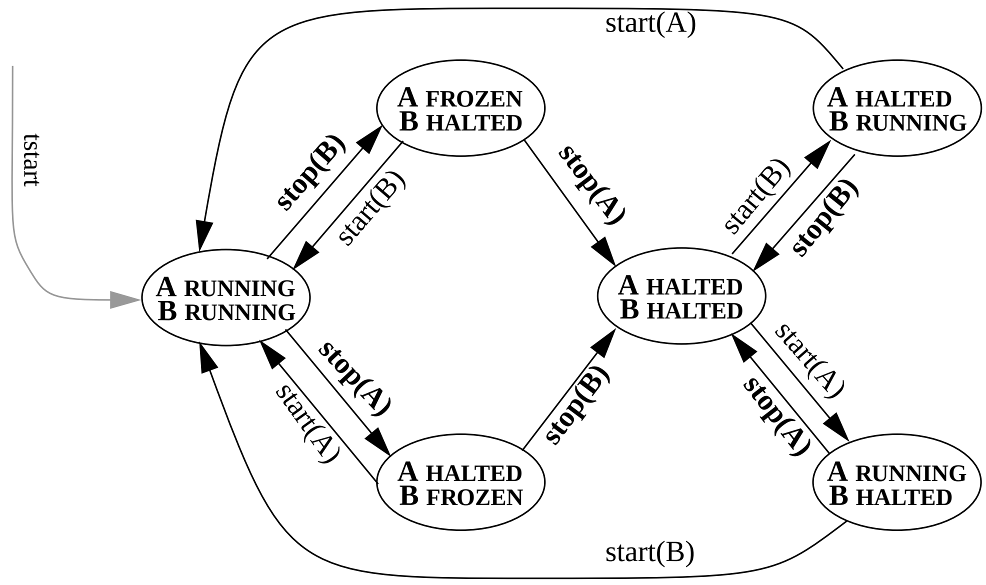
1. State: A-RUNNING, B-RUNNING, C-RUNNING
2. Event: Stop B
3. State: A-FROZEN, B-HALTED, C-FROZEN (A and C are frozen by halting B)
4. Event: Stop A
5. State: A-HALTED, B-HALTED, C-FROZEN (A is halted, B stays halted, and C stays frozen)
6. Event: Start B
7. State: A-HALTED, B-FROZEN, C-FROZEN (B is started, but since A is halted, B is now frozen, and C stays frozen)
8. Event: Start A
9. State: A-RUNNING, B-RUNNING, C-RUNNING

From the above example, it is clear that all devices in a team have to be started (or unhalted) for execution to resume. Otherwise, all devices are kept in the HALTED or FROZEN state as determined by whether or not MDIStop() has been called by their respective debuggers.

While in the FROZEN state a device must continue to respond to all MDI calls that it would have done while in the RUNNING state, i.e. at least MDIRunState() and MDIStop(). Similarly when a device is in the HALTED state, then it must be capable of responding to all normal MDI calls, irrespective of the state of the other devices.

For multi-processor aware debuggers, the MDITeamExecute() call will force all team members to be placed simultaneously (or as simultaneously as possible) into the RUNNING state, irrespective of their previous states.

Figure 3-2 State Transitions for Legacy Team Debugging



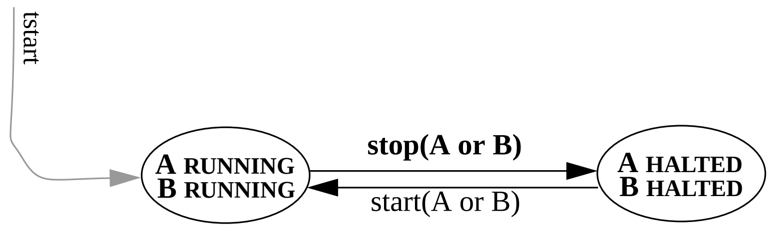
Actions
stop = breakpoint/single-step exception, or debugger calls MDIStop()
start = debugger calls MDIExecute() or MDIStep()
tstart = debugger calls MDITeamExecute()

States
FROZEN = processor/VPE freeze or hidden debug mode, MDIRunState() returns MDIStatusRunning
HALTED = debug mode, MDIRunState() returns appropriate "non-running" status
RUNNING = normal execution mode

3.2.1.2 MP-Aware Team Debugging

A multi-processor-aware debugger may set the *MDICBSync()* callback linkage when connecting to an MDILib, to indicate that it is willing to handle dynamic changes in a device's state caused by another debugger. See Section 6.1.5, "Synchronize State: Callback function to synchronize device state changes" on page 26. If the *MDICBSync()* callback is not null then the FROZEN state is no longer required, and starting or stopping any team member simply starts or stops the other team members immediately. If a team member is changed from the HALTED state to the RUNNING state by the action of another debugger, then its controlling debugger is notified by a call to its *MDICBSync()* function with a *SyncType* argument of MDISyncState.

Figure 3-3 State Transitions for MP-Aware Team Debuggingc



Actions
stop = breakpoint/single-step exception, or debugger calls MDIStop()
start = debugger calls MDIExecute() or MDIStep()
tstart = debugger calls MDITeamExecute()

States
HALTED = debug mode, MDIRunState() returns appropriate "non-running" status
RUNNING = normal execution mode

3.2.2 Disabled Multi-processor Devices

A disabled device is one which is incapable of executing instructions, even in debug mode. For a single-threaded CPU this may mean that it is powered down, or its clocks are switched off. In a multi-threaded CPU it may be a VPE that has no TCs bound to it.

Connecting to a disabled device requires special handling. The MDITGQuery() and MDIDquery() calls must list the device even when it is disabled. The calls to MDITGOpen() and MDIOpen() must also succeed. But after that the only MDI functions which are required to have any useful effect on the device are:

- *MDIStop()*: Raises a debug interrupt request to the device, so that as soon as it is enabled (presumably by another device), it will immediately enter debug mode and the HALTED state, before executing any normal instructions.
- *MDIRunState():* If the device is disabled after the wait time expires, then returns MDIStatusDisabled. The debugger can either display an error and terminate the connection, or continue to poll MDIRunState interruptibly until the device is enabled. A debugger detecting that its device has switched from returning MDIStatusRunning to MDIStatusDisabled will most likely report that the target program has been terminated, and may disconnect from the device.
- *MDIAttachTM():* It is permitted to attach a disabled device to a team even if it hasn't been opened, but it will be in a *pending* state (i.e. pending RUNNING, pending FROZEN or pending HALTED), shadowing the state diagrams shown above, but with MDIRunState() still returning MDIStatusDisabled. If and when it is enabled then it shall immediately switch to the equivalent real state and return the appropriate status from MDIRunState(). If it should later be disabled again, then it will return to the appropriate pending shadow state, returning MDIStatusDisabled.
- *MDIDetachTM():* It is permitted to detach a disabled device to a team, which will remove it from any pending HALTED or FROZEN state and return it immediately to the free running state if and when it is later enabled.
- *MDIReset()*: Be beware that this resets the whole CPU, not just the VPE to which you are connected.

All other MDI functions which target the device may return MDIErrDisabled.

Note that VPEs which are temporarily prevented from issuing instructions by another VPE, but still have at least one TC bound to them, are not reported as disabled, but remain in the RUNNING or HALTED states.

Chapter 4

MDI Environment Command Set

4.1 Version: Obtain the supported MDI versions for this MDILib implementation

```
MDIInt32
MDIVersion (MDIVersionRangeT *versions)
```

Returns:

MDISuccess	No Error, requested data has been returned.
MDIErrParam	Invalid parameter.

Structures:

```
typedef struct MDIVersionRange_struct {
      MDIVersionT   oldest;
      MDIVersionT   newest;
} MDIVersionRangeT;
```

Description:

For the given MDILib implementation, this call retrieves the range of supported MDI specification versions. *versions* is a pointer to a structure where the oldest and newest version numbers supported by this MDILib implementation are returned. All versions between oldest and newest must also be supported. The 32 bit version number is divided into a 16 bit Major field (Bits 31:16) and a 16 bit Minor field (Bits 15:0). The current release of this specification is version 0x000200D. For implementations that only support only one revision of the specification, oldest == newest.

The macro MDICurrentRevision (defined in the mdi.h file) always shows the latest (or current) revision number of this specification.

4.2 Connect: Establish a connnection to the MDILib

```
MDIInt32
MDIConnect ( MDIVersionT MDIVersion,
             MDIHandleT * MDIHandle,
             MDIConfigT * Config)
```

Returns:

MDISuccess	No Error, handle and configuration have been returned.
MDIErrFailure	An unspecified error occurred, connection was not successful.
MDIErrParam	Invalid parameter
MDIErrVersion	Version is not supported.
MDIErrNoResource	Maximum connections has been reached.
MDIErrAlreadyConnected	MDI Connection has already been made for this thread.
MDIErrConfig	Required debugger callback functions are not present in Config structure.
MDIErrInvalidFunction	A callback function pointer is invalid.

Structures:

```
typedef MDIUint32 MDIVersionT;

typedef MDIUint32 MDIHandleT;

typedef struct MDIConfig_struct {
   /* Provided By */
   /*        Other Comments */
   char   User[80]; /* Host:  ID of caller of MDI */
   char   Implementer[80]/* MDI: ID of MDI implementer */
   MDIUint32 MDICapability; /* MDI:   Flags for optional capabilities */

   MDIInt32  (__stdcall *MDICBOutput)/* Host:  CB fn for MDI output */
         (MDIHandleT Device, MDIInt32 Type,
         char *Buffer, MDIInt32 Count );

   MDIInt32 (__stdcall *MDICBInput)/* Host:  CB fn for MDI input */
         (MDIHandleT Device, MDIInt32 Type,
         MDIInt32 Mode, char **Buffer,
         MDIInt32 *Count);

   MDIInt32 (__stdcall *MDICBEvaluate)/* Host:  CB fn for expression eval */
         (MDIHandleT Device, char *Buffer,
         MDIInt32 *ResultType, MDIResourceT *Resource,
         MDIOffsetT *Offset, MDIInt32 *Size, void **Value);

   MDIInt32   (__stdcall *MDICBLookup)/* Host:  CB fn for sym/src lookup */
         (MDIHandleT Device, MDIInt32 Type,
         MDIResourceT Resource, MDIOffsetT Offset,
         char **Buffer );

   MDIInt32   (__stdcall *MDICBPeriodic)/* Host: CB fn for Event processing */
       (MDIHandleT Device);

   MDIInt32   (__stdcall *MDICBSync)/* Host:  CB fn for Synchronizing */
       (MDIHandleT Device, MDIInt32 Type,
         MDIResourceT Resource);

} MDIConfigT;

/* MDIConfigT.MDICapability flag values, can be OR'ed together */

#define MDICAP_NoParser       0x00000001  /* No command parser */
#define MDICAP_NoDebugOutput  0x00000002  /* No Target I/O */
#define MDICAP_TraceOutput    0x00000004  /* Supports Trace Output */
#define MDICAP_TraceCtrl      0x00000008  /* Supports Trace Control */
#define MDICAP_TargetGroups   0x00000010  /* Supports Target Groups */
#define MDICAP_PDtrace        0x00000020  /* Supports PDtrace functions */
#define MDICAP_TraceFetchI    0x00000040  /* Supports Instr Fetch during Trace */
#define MDICAP_TC             0x00000080  /* Supports Thread Contexts */
#define MDICAP_Teams          0x00000100  /* Supports Teams */
```

Description:

This opens the requested connection and is also used to configure and retrieve information about supported MDI features.

The *MDIVersion* input parameter is the version of the MDI specification to which this connection will adhere. It will typically be the highest version number within the version range returned by the MDIVersion() call, that is supported by the debugger. If *MDIVersion* is not within the version range returned by MDIVersion(), MDIConnect() will return MDIErrVersion and the connection will not be made. Note that the behavior of some MDI functions may change dependent on the value of MDIVersion.

On input, Config->User contains a null-terminated ASCII character string identifying the debugger to the MDILib. The Implementor string is returned by the MDILib. The User and Implementer strings are arbitrary, but it is recommended that the strings include the name of the vendor of the debugger and MDILib. They are intended to allow the debugger and MDILib to determine if the other is a known implementation, perhaps to enable vendor-specific extensions. (No feature extensions may use public names beginning with the characters “MDI” or “Mdi”. These are reserved for the MDI specification.)

The two values, Config->MDICBOutput and Config->MDICBInput are set to the addresses of the call-back functions that the debugger must provide for I/O. If these are NULL, then the MDILib returns the MDIErrConfig error condition. The other four callback functions (Config->MDICBEvaluate, Config->MDICBLookup, Config->MDICBPeriodic, and Config->MDICBSync) are optional. If these are not implemented, the debugger must initialize these values to NULL.

On output, the MDILib returns an unique handle, *MDIHandle* for the connection. This must be used in all future interactions of this debugger to the MDILib. Since multiple debuggers are allowed to simulateneously talk to the MDILib, this allows the MDILib to know which debugger is making any specific request.

Zero or more of the following flag values specifying MDILib capabilities are OR'ed together into Config->MDICapability. The intent is to allow a GUI debugger to disable user interface elements not supported by the MDILib connection.

MDICAP_NoParser	MDILib has no command parser (see MDIDoCommand())
MDICAP_NoDebugOutput	MDILib will not call MDICBOutput()
MDICAP_TraceOutput	Capable of producing Trace Output
MDICAP_TraceCtrl	Capable of controlling Trace
MDICAP_TargetGroups	Capable of executing Target Group commands
MDICAP_PDtrace	Capable of supporting PDtrace
MDICAP_TraceFetchI	Capable of supporting Instruction Fetch during trace
MDICAP_TC	Capable of supporting thread contexts
MDICAP_Teams	Capable of supporting teams

4.3 Disconnect: Disconnect from the MDILib

```
MDIInt32
MDIDisconnect(MDIHandleT MDIHandle,
              MDIUint32 Flags)
```

Returns:

MDISuccess	No Error
MDIErrMDIHandle	Invalid MDI Handle
MDIErrParam	Invalid flags value

MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Structures:

Flags:

MDICurrentState	Close all open target groups and target devices
MDIResetState	Place all open target devices in reset, then close all open target groups and target devices

Description:

Disconnect from the MDILib after first closing any open Target Groups and Devices associated with this connection. It must be possible to disconnect even when some or all of the Devices on a multi-processor core are disabled. All team data associated with this MDILib should be retained until the final debugger disconnects from the library.

Chapter 5

Target Group Command Set

A collection of devices can form a target group. For example, all the processors in a multiprocessor implementation might be a target group, while each individual processor would be one device in this target group. Or, in another implementation the target group could comprise of the main processor core and the DSP.

The MDILib may optionally support the ability to perform certain operations on a target group. If so, it will set the MDICAP_TargetGroups flag in Config->MDICapability. If this flag is set, then it implies not only that the MDILib supports target group calls, but that there is at least one target group present. Hence, if this flag is set, the debugger must use the function calls in this group to get a list of target groups and open the required group before it can query and open a specific device within that group.

If the Config->MDICapability flag is not set, the debugger is required to bypass all the function calls in this command set and proceed directly to the device query call, MDIDQuery(). For MDILib implementations that do not support group operations, all Target Group functions will return MDIErrUnsupported.

5.1 Target Group Query: Retrieves the names of the defined target groups.

```
MDIInt32
MDITGQuery (  MDIHandleT MDIHandle,
              MDIInt32 *HowMany,
              MDITGDataT *TGData)
```

Returns:

MDISuccess	No Error, requested data has been returned
MDIErrMDIHandle	Invalid MDI Handle
MDIErrParam	Invalid parameter
MDIErrMore	More target groups defined than requested
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Structures:

```
typedef struct MDITGData_struct {
    MDITGIdT      TGId;
    char          TGName[81];
} MDITGDataT;
```

Description:

MDIHandle must be the value returned by a previous MDIConnect() call.

If the requested number of target groups (**HowMany*) is 0, the function returns no error (MDISuccess) and **HowMany* is set to the number of available target groups. If **HowMany* is non-zero on entry, it specifies the number of elements in the *TGData* array being passed in. The function fills in the *TGData* array with information for up to **HowMany* target groups and sets **HowMany* to the number filled in. If there is not enough room in the *TGData* array to hold all the available target groups, MDIErrMore is returned. If the debugger then calls MDITGQuery() again before any other MDI functions are called, information is returned for the next **HowMany* target groups.

Target groups are identified by a null terminated ASCII string (*TGData->TGName*) and a unique target group ID (*TGData->TGId*). The strings are intended to be descriptive, but they are MDILib implementation-specific and the debugger should not interpret this or rely on this for any implementation-specific information. It is simply displayable text that names the target group. It is intended that the debugger should show these target group names to the user for selection of the target group to be opened. The string name may not be more than 80 characters excluding the null terminator.

Information about groups within a multi-processor shall be returned even when a group is disabled and awaiting initialization by another device.

The target group ID (*TGData->TGId*) is used in the MDITGOpen() function to select the specific target group.

5.2 Target Group Open: Opens a target group

```
MDIInt32
MDITGOpen (MDIHandleT MDIHandle,
           MDITGIdT TGId,
           MDIUint32 Flags,
           MDIHandleT *TGHandle)
```

Returns:

MDISuccess	No Error, *TGHandle has been set to the target group handle
MDIErrFailure	An unspecified error occurred, open was not successful
MDIErrParam	Invalid parameter
MDIErrTGId	Invalid TGId
MDIErrNoResource	The TG has already been opened by another debugger either on an exclusive basis, or the TG does not support shared access
MDIErrWrongThread	Call was not made by the connected thread
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Structures:

Flags:

MDISharedAccess	Shared Access
MDIExclusiveAccess	Exclusive Access

Description:

MDIHandle must be the value returned by the previous MDIConnect call. MDILib implementations are not required to support shared access to a Target Group.

Flags is set to MDIExclusiveAccess if the debugger wants exclusive control over any open devices in this target group; otherwise *Flags* is set to MDISharedAccess to allow other debuggers to open devices in this target group. If shared access is not supported by the target group, an attempt to open a Target Group already opened by another debugger will return MDIErrNoResource even if both the open calls requested shared access.

The handle returned in **TGHandle* is used to reference this target group. If the debugger does not support group execute operation (MDITGExecute()) and is connected to an MDILib that does, then the debugger should open the selected target group with exclusive access to avoid the possibility that devices opened by the current debugger could be affected by group execute commands issued by another debugger.

It must be possible to open a target group within a multi-processor when that group is disabled and awaiting initialization by another device.

5.3 Target Group Close: Close a previously opened target group

```
MDIInt32
MDITGClose (  MDIHandleT TGHandle,
              MDIUint32 Flags);
```

Returns:

MDISuccess	No Error
MDIErrTGHandle	Invalid Target Group handle
MDIErrParam	Invalid Flags parameter
MDIErrWrongThread	Call was not made by the connected thread
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Structures:

Flags:

MDICurrentState	Leave in current state
MDIResetState	Reset all target devices

Description:

Any open devices in the group will be first closed automatically before the target group is closed.

It must be possible to close a target group within a multi-processor even when that group is disabled and awaiting initialization by another device.

Beware that using MDIResetState will reset all VPEs within a multi-threaded CPU, not just the connected VPE.

5.4 Target Group Execute: Place in execution mode the appropriate devices in the target group

```
MDIInt32
MDITGExecute (MDIHandleT TGHandle);
```

Returns:

MDISuccess	No Error
MDIErrFailure	Unable to perform group execute
MDIErrTGHandle	Invalid target group handle

MDIErrWrongThread	Call was not made by the connected thread
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback
MDIErrDisabled	Service cannot be performed because the target group is disabled

Description:

Place all the devices in the specified target group that have been configured for target group control in a run state and run them. There is no need to call this function if there is only one device in a target group, it suffices to call the device run command (Section 6.4.1, "Execute: Place the device into its RUNNING state" on page 38).

5.5 Target Group Stop: Stop execution for all appropriate devices in the target group

```
MDIInt32
MDITGStop (MDIHandleT TGHandle)
```

Returns:

MDISuccess	No Error
MDIErrFailure	Unable to perform group stop
MDIErrTGHandle	Invalid target group handle
MDIErrWrongThread	Call was not made by the connected thread
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

Stop the execution of all those devices in the target group that have been configured for target group control.

Issuing a stop request to a target group within a multi-processor is permitted even if that group is disabled and awaiting initialization by another device. The stop request should be serviced as soon as the group is enabled.

Chapter 6

Device Command Set

The device command set is subdivided into the following sections:

- Section 6.1, "Session Control" on page 23 has commands used to identify and select the necessary device to open, control, and support debugger event processing and multiple debugger synchronization.
- Section 6.2, "Resource Addresses" on page 27 defines device resources and how they can be accessed.
- Section 6.3, "Resource Access" on page 27 has commands that access device resources.
- Section 6.4, "Run Control" on page 37 has commands that control a device.
- Section 6.5, "Breakpoints" on page 43 has commands that establish and maintain breakpoints within a device.

6.1 Session Control

6.1.1 Device Query: Retrieves information about the devices

```
MDIInt32
MDIDQuery (   MDIHandleT Handle,
              MDIInt32 *HowMany,
              MDIDDataT *DData)
```

Structures:

```
typedef struct MDIDData_Struct {
   MDIDeviceIdT Id;
   char         DName[81];
   char         Family[15];
   char         FClass[15];
   char         FPart[15];
   char         FISA[15];
   char         Vendor[15];
   char         VFamily[15];
   char         VPart[15];
   char         VPartRev[15];
   char         VPartData[15];
   char         Endian;
} MDIDDataT;
```

Returns:

MDISuccess	No Error
MDIErrTGHandle	Invalid target group handle
MDIErrParam	Invalid parameter
MDIErrMore	More devices defined than requested
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

If the requested number of devices (*HowMany) is 0, the function returns no error (MDISuccess) and *HowMany is set to the number of devices in the target group. If *HowMany is non-zero on entry, it specifies the number of elements in the DData array being passed in. The function fills in the DData array with information for up to *HowMany devices and sets *HowMany to the number filled in. If there is not enough room in the DData array to hold all the available devices, MDIErrMore is returned. If the debugger then calls MDIDQuery again before any other MDI functions are called, information is returned for the *next* *HowMany devices.

Retrieves the general configuration information about the devices in the target group, or all devices if the MDILib does not support Target Groups.

If the MDILib implementation did not set the MDICAP_TargetGroups capability, *Handle* must be the MDIHandle returned by the previous MDIConnect() call. Otherwise *Handle* must be the TGHandle returned by a previous MDITGOpen call.

DData->DName is an 80 character plus null terminated ASCII string that describes and identifies a device available for connection. Its value is determined by the MDILib and debuggers should not attempt to interpret the data. When more than one device is available, it is intended that the debugger will display the *DName* strings to allow the user to select the desired device. *DData->Id* is a unique device ID assigned by the MDILib, and used by the debugger to specify the desired device to MDIOpen().

Information about devices within a multi-processor shall be returned even when a device is disabled and awaiting initialization by another device. In such cases, the data returned by MDIQuery may not contain fully accurate information about the device. The FPart, FISA, and Endian fields may return a value of "Unknown", or a cached value from the last successful connection. A debugger may continue to open the device and then poll MDIRunState() until the device returns a status other than MDIStatusDisable, at which point the debugger may call MDIQuery again to retrieve valid information about the device.

Devices are also identified by family, class, generic part, vendor, vendor family, vendor part, vendor part revision and vendor part specific fields. All of these fields are ASCII strings with a maximum length of 15 characters including null termination. Any excess bytes in the field beyond the null termination will be set to zero to facilitate using a memory compare function to determine if the device is supported by the debugger.

DData->Family is the type of device. Valid values for *DData->Family* are part of the generic MDI specification. The only values currently specified are MDIFamilyCPU ("CPU") and MDIFamilyDSP ("DSP"). *DData->FClass* further isolates the device type (E.g., MIPS, PPC, X86, etc.). *DData->FPar*t is the industry common name for the processor. (LR4102, NEC5440, 80486). *DData->FISA* is the "Instruction Set Architecture" supported by the device (MIPS I, MIPS IV). Valid values for *DData->FClass* and *DData->FISA* are architecture-specific and are listed in the corresponding Appendix. *DData->Vendor* identifies the device manufacturer or IP vendor. *DData->VFamily*, *DData->VPart*, *DData->VPartRev*, and *DData->VPartData* are vendor specific values intended to refine the generic part. It is intended that device vendors will publish a list of standard values for these fields for each of their devices.

Debugger and MDILib implementations may have their own mechanism for configuring the device type and are not required to make any use of the architecture- and vendor-specific values; however, if they do make any use of these fields, they are required to document which fields are inspected and what values they look for.

6.1.2 Open: Opens a device.

```
MDIInt32
MDIOpen ( MDIHandleT Handle,
          MDIDeviceIdT DeviceID,
          MDIUint32 Flags,
          MDIHandleT * DeviceHandle)
```

Structures:

Flags:

MDISharedAccess	Shared Access
MDIExclusiveAccess	Exclusive Access

Returns:

MDISuccess	No Error. Device handle is returned in DeviceHandle
MDIErrFailure	An unspecified error occurred, open was not successful
MDIErrDeviceId	Invalid Device ID
MDIErrParam	Invalid parameter
MDIErrHandle	Invalid target group or connection handle specified
MDIErrNoResource	Device already opened, either exclusively or shared access is not supported
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

If the MDILib implementation did not set the MDICAP_TargetGroups capability, *Handle* must be the MDIHandle returned by the previous MDIConnect call; otherwise *Handle* must be the TGHandle returned by a previous MDITGOpen() call.

The returned handle is used to reference this device in all other target device commands. Devices that are opened for shared access may be opened by another debugger. Debuggers may be kept in sync via the call back function MDICBSync. MDILib implementations are not required to support shared access to a Device. If shared access is not supported, an attempt to open a Device already opened by another debugger will return MDIErrNoResource even if both opens specified shared access.

It must be possible to open a device within a multi-processor when that device is disabled and awaiting initialization by another device. A debugger must then poll MDIRunState() until the device returns a status other than MDIStatusDisabled, at which point the debugger may call MDIQuery again to retrieve valid device information.

6.1.3 Close: Closes a device.

```
MDIInt32
MDIClose (   MDIHandleT DeviceHandle,
             MDIUint32 Flags)
```

Structures:

Flags:

MDICurrentState	Leave in current state
MDIResetState	Reset target device

Returns:

MDISuccess	No Error
MDIErrFailure	Unable to close for an unspecified reason

MDIErrParam	Invalid flags parameter
MDIErrDevice	Invalid device handle specified
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

It must be possible to close a target group within a multi-processor even when that group is disabled and awaiting initialization by another device.

It must be possible to close a device within a multi-processor even when that device is disabled and awaiting initialization by another device.

Beware that using MDIResetState will reset all VPEs within a multi-threaded CPU, not just the connected VPE.

6.1.4 Process Events: Callback function to process periodic events

```
MDIInt32
MDICBPeriodic (MDIHandleT DeviceHandle)
```

Returns:

MDISuccess	No Error
MDIErrDevice	Invalid device handle

Description:

This call-back function is optionally implemented by the debugger. Its address, or NULL if it is not implemented, is passed to the MDILib in *Config->MDICBPeriodic* when MDIConnect is called. The purpose of this call-back is to give the debugger a chance to process user events during a long-running MDI service call. If the debugger implements this function, the MDILib is required to call it at least every 100 milliseconds. At this point the debugger may cancel the current MDI command by calling MDIAbort, update user interfaces or do other debugger maintenance. It may not call any MDI functions other than MDIAbort.

6.1.5 Synchronize State: Callback function to synchronize device state changes

```
MDIInt32
MDICBSync (   MDIHandleT Device,
              MDIInt32 SyncType,
              MDIResourceT SyncResource)
```

Structures:

SyncType:

MDISyncBP

MDISyncState

MDISyncWrite

```
typedef MDIUint32   MDIResourceT;
```

Returns:

MDISuccess	No Error
MDIErrDevice	Invalid device handle

Description:

This call-back function is optionally implemented by debuggers. Its address, or NULL if it is not implemented, is passed to the MDILib in *Config->MDICBSync* when MDIConnect() is called. The purpose of this callback is to inform an MDI application of device state changes caused by MDI functions performed by others when a device has been opened in MDISharedAccess mode by multiple MDI applications, or is a member of a multi-processing team. The reported device state changes keep the application informed of resource, break point and run state changes that have occurred in the target.

When the MDILib receives a command that modifies the current breakpoint settings, all sharing MDI applications with MDICBSync() call-back functions will receive an MDICBSync() call with *SyncType* set to MDISyncBP. The SyncResource parameter will be set to 0.

When the MDILib receives a command that modifies the current run state of the device, all sharing MDI applications with MDICBSync call-back functions will receive an MDICBSync() call with SyncType set to MDISyncState. The SyncResource parameter will be set to 0.

When the MDILib receives a command that modifies a resource (MDIWrite, MDIWriteList, MDIFill, MDIMove), all sharing MDI applications with MDICBSync() call-back functions will receive an MDICBSync() call with *SyncType* set to MDISyncWrite, and *SyncResource* set to the resource that has been modified.

Actions to be taken by the MDI Application are application dependent, but could include querying the MDILib for current run state, BP list, or resource values.

6.2 Resource Addresses

Device resources (e.g. memory and registers) are identified by their *address*. An *address* consists of an *offset* and a *space* (resource number). The *space* is a 32-bit unsigned integer specifying the type of resource (address “space”), and the *offset* is a 64-bit unsigned integer specifying the location of a specific storage unit within that space. The interpretation of the *offset* is determined by the *space*. The list of specific resource numbers, and the corresponding interpretation of the offset and meaning of the address, is architecture dependent; however, the MDI specification assumes that the offset for "memory like" resources will be a byte offset while the offset for "register like" resources will be a "register number". This distinction is important for alignment considerations.

6.3 Resource Access

The functions in this section allow target resources (memory and registers) to be inspected, set, and manipulated. The following parameter descriptions apply to all of these functions:

MDIHandleT	Device	Device handle.
MDIResourceT	SrcResource	Source resource address space for data provided from the device.
MDIOffsetT	SrcOffset	Source resource address offset for data provided from the device.
MDIResourceT	DstResource	Destination resource address space for data provided from the device.
MDIOffsetT	DstOffset	Destination resource address offset for data provided from the device.

MDIUint32	ObjectSize	Size of each object being referenced by the Src and/or Dst address. Where applicable and possible, the device should perform the actual read or write accesses with bus cycles having the specified size. For memory mapped resources, the offset is required to be aligned appropriately for the object size. Valid values for ObjectSize are: 0 Valid only for memory mapped resources. Object size is 1. The device data can be read or written in the most efficient manner 1 Byte (8-bit) 2 Half-word (16-bit) 4 Word (32-bit) 8 Double word (64-bit)
MDIUint32	Count	The number of objects to be accessed. For memory-mapped resources, if ObjectSize is 0 then Count is to be interpreted as a byte count.
void *	Buffer	The address of a host data buffer supplying or receiving the device data. The buffer must be large enough to hold all the data. The buffer pointer must remain valid until the MDILib function to which it is passed has returned.

Device data is always passed as a packed array of Count elements, with each element in device byte order (endian). The size of each element is given by ObjectSize. For register type resources where ObjectSize is less than the actual size of the registers being addressed, the low order ObjectSize bytes of each register is returned by read operations and each value is either sign-extended or zero-extended to the register size by write operations; this is architecture-specific. For register type resources where ObjectSize is greater than the actual size of the registers being addressed, each register value is either sign-extended or zero-extended to ObjectSize bytes by read operations and the high order bytes of each value are ignored by write operations.

For resources which are duplicated by each thread context, such as the general purpose registers, the current MDI TC ID is used to select which TC's registers to access. See Section 9.1.1, "Set Thread Context: Sets the current MDI thread context ID" on page 71.

6.3.1 Read: Reads a contiguous range of data from the specified resource on the device.

```
MDIInt32
MDIRead ( MDIHandleT Device,
          MDIResourceT SrcResource,
          MDIOffsetT SrcOffset,
          void * Buffer,
          MDIUint32 ObjectSize,
          MDIUint32 Count)
```

Structures:

```
typedef MDIUint64   MDIOffsetT;
```

Returns:

MDISuccess	No Error, requested data has been returned or resource address validated
MDIErrFailure	Unable to perform read operation. This implies a probe hardware failure or some such fatal reason.
MDIErrDevice	Invalid device handle

MDIErrSrcResource	SrcResource is an invalid or unsupported resource type, for example, the device in question might not have a secondary or tertiary cache.
MDIErrInvalidSrcOffset	SrcOffset is invalid for the specified resource, that is out of range.
MDIErrSrcOffsetAlignment	SrcOffset is not correctly aligned for the specified ObjectSize.
MDIErrSrcObjectSize	Invalid ObjectSize for the specified SrcResource.
MDIErrSrcCount	Specified Count and SrcOffset reference space that is outside the scope for the given resource. No objects were returned.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Note that it is valid, and useful, to call MDIRead() with *Count* set to 0. In this case, no data is transferred and the return value can be checked to determine whether the address is valid and access to the resource is supported. The MDILib is required to validate the address and return MDIErrSrcResource, MDIErrInvalidSrcOffset, or MDIErrSrcOffsetAlignment as appropriate, even when *Count* is 0. When there are no errors, then MDISuccess is returned even if no data is returned.

Note that it is the responsibility of the debugger to have allocated *Buffer* of the appropriate size before calling MDIRead().

6.3.2 Write: Writes a contiguous range of data to the specified resource on the device.

```
MDIInt32
MDIWrite (   MDIHandleT Device,
             MDIResourceT DstResource,
             MDIOffsetT DstOffset,
             void *Buffer,
             MDIUint32 ObjectSize,
             MDIUint32 Count)
```

Returns:

MDISuccess	No Error, requested data has been written.
MDIErrFailure	Unable to perform write operation. This implies a probe hardware failure or some such fatal reason.
MDIErrDevice	Invalid device handle.
MDIErrDstResource	DstResource is an invalid or unsupported resource type, for example, the specified device might not have floating-point registers, if there is no floating point unit.
MDIErrInvalidDstOffset	DstOffset is invalid for the specified resource.
MDIErrDstOffsetAlignment	DstOffset is not correctly aligned for the specified ObjectSize.
MDIErrDstObjectSize	Invalid ObjectSize for the specified DstResource.
MDIErrDstCount	Specified Count and DstOffset reference space that is outside the scope for the given resource. No objects were written.
MDIErrWrongThread	Call was not made by the connected thread.

MDIErrTargetRunning	Service cannot be performed at this time because the target program is running
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

6.3.3 Read List: Read a set of values

```
MDIInt32
MDIReadList (MDIHandleT Device,
             MDIUint32 ObjectSize,
             MDICRangeT *SrcList,
             MDIUint32 ListCount,
             void *Buffer)
```

Structures:

```
typedef struct MDICRange_struct {
    MDIOffsetT      Offset;
    MDIResourceT    Resource;
    MDIInt32        Count;
} MDICRangeT;
```

Returns:

MDISuccess	No Error, requested data has been returned.
MDIErrFailure	Unable to perform read operation. This implies a probe hardware failure or some such fatal reason.
MDIErrDevice	Invalid device handle.
MDIErrSrcResource	Invalid or unsupported resource type in SrcList.
MDIErrInvalidSrcOffset	Offset is invalid for the specified resource, i.e., it is out of range.
MDIErrSrcOffsetAlignment	Offset is not correctly aligned for the specified ObjectSize.
MDIErrSrcObjectSize	Invalid ObjectSize for the specified Resource.
MDIErrSrcCount	Specified Count and SrcOffset reference space that is outside the scope for the given resource.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Read a set of values from a list of address ranges on the device. The list may contain different resource types, but a single *ObjectSize* must apply to all objects in the list.

SrcList is an array of object descriptors, each of which includes an address (Resource and Offset) and the number of objects to read. *ListCount* is the number of entries in the *SrcList* array.

6.3.4 Write List

```
MDIInt32
MDIWriteList (   MDIHandleT Device,
                 MDIUint32 ObjectSize,
                 MDICRangeT * DstList,
                 MDIUint32 ListCount,
                 void *Buffer)
```

Returns:

MDISuccess	No Error, requested data has been written.
MDIErrFailure	Unable to perform write operation. This implies a probe hardware failure or some such fatal reason.
MDIErrDevice	Invalid device handle.
MDIErrDstResource	DstResource is an invalid or unsupported resource type, for example, the device might not have floating-point registers.
MDIErrInvalidDstOffset	DstOffset is invalid for the specified resource.
MDIErrDstOffsetAlignment	DstOffset is not correctly aligned for the specified ObjectSize.
MDIErrDstObjectSize	Invalid ObjectSize for the specified Resource.
MDIErrDstCount	Specified Count and DstOffset reference space that is outside the scope for the given resource. No objects were written.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Write a set of values to a list of address ranges on the device. The list may contain different resource types, but a single *ObjectSize* must apply to all objects in the list.

DstList is an array of object descriptors, each of which includes an address (Resource and Offset) and the number of objects to write. *ListCount* is the number of entries in the *DstList* array.

6.3.5 Move: Move data from one resource to another on the device

```
MDIInt32
MDIMove ( MDIHandleT Device,
          MDIResourceT SrcResource,
          MDIOffsetT SrcOffset,
          MDIResourceT DstResource,
          MDIOffsetT DstOffset,
          MDIUint32 ObjectSize,
          MDIUint32 Count,
          MDIUint32 Direction);
```

Structures:

Direction

MDIMoveForward	Start to End
MDIMoveBackward	End to Start

Returns:

MDISuccess	No Error, requested data has been moved.
MDIErrFailure	Unable to perform read operation.
MDIErrDevice	Invalid device handle.
MDIErrSrcResource	SrcResource is an invalid or unsupported resource type.
MDIErrInvalidSrcOffset	SrcOffset is invalid for the specified SrcResource.
MDIErrSrcOffsetAlignment	SrcOffset is not correctly aligned for the specified ObjectSize.
MDIErrSrcObjectSize	Invalid ObjectSize for the specified SrcResource.
MDIErrSrcCount	Specified Count and SrcOffset reference space that is outside the scope for the given SrcResource.
MDIErrDstResource	DstResource is an invalid or unsupported resource type.
MDIErrInvalidDstOffset	DstOffset is invalid for the specified DstResource.
MDIErrDstOffsetAlignment	DstOffset is not correctly aligned for the specified ObjectSize.
MDIErrDstCount	Specified Count and DstOffset reference space that is outside the scope for the given resource.
MDIErrDstObjectSize	Invalid ObjectSize for the specified DstResource.
MDIErrAbort	Command was aborted in response to an MDIAbort call.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Moves data from one resource to another resource on the device. If *Direction* is set to MDIMoveForward the move will be done starting from the beginning of the range until the end is reached. If *Direction* is set to MDIMoveBackward, the move will be done backwards starting from the end of the range to the beginning.

6.3.6 Fill: Fill the specified resource on the device with a pattern.

```
MDIInt32
MDIFill ( MDIHandleT Device,
          MDIResourceT DstResource,
          MDIRangeT DstRange,
          void *Buffer,
          MDIUint32 ObjectSize,
          MDIUint32 Count);
```

Structures:

```
typedef struct MDIRange_struct {
```

```
    MDIOffsetT      Start;
    MDIOffsetT      End;
} MDIRangeT;
```

Returns:

MDISuccess	No Error, requested data has been written.
MDIErrFailure	Unable to perform fill operation.
MDIErrDevice	Invalid device handle.
MDIErrDstResource	DstResource is an invalid or unsupported resource type.
MDIErrInvalidDstOffset	DstRange is invalid for the specified resource.
MDIErrDstOffsetAlignment	DstOffset is not correctly aligned for the specified ObjectSize.
MDIErrDstObjectSize	Invalid ObjectSize for the specified DstResource.
MDIErrDstCount	Specified Count and DstOffset reference space that is outside the scope for the given resource. No objects were written.
MDIErrAbort	Command was aborted in response to an MDIAbort call.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

The pattern is an array of *Count* objects of size *ObjectSize*. It is not required that the destination range be an exact multiple of the pattern size. *ObjectSize* must be non-zero. The MDILib is required to support this function only for memory-mapped resources, and up to a maximum *Count* of 256.

6.3.7 Find: Find a pattern in a resource

```
MDIInt32
MDIFind ( MDIHandleT Device,
          MDIResourceT SrcResource,
          MDIRangeT SrcRange,
          void *Buffer,
          void *MaskBuffer,
          MDIUint32 ObjectSize,
          MDIUint32 Count,
          MDIOffsetT *FoundOffset,
          MDIUint32 Mode)
```

Structures:

Search *mode*:

MDIMatchForward	Match specified Pattern, searching forward from the start address.
MDIMismatchForward	Matches anything that is not the specified Pattern, searching forward from the start address.
MDIMatchBackward	Matches specified Pattern, searching backward from the end address.

MDIMismatchBackward	Matches anything that is not the specified Pattern, searching backward from the end address.

Returns:

MDISuccess	No Error, requested pattern match has been found at the address returned in FoundOffset.
MDINotFound	No Error, entire range was searched without finding a pattern match.
MDIErrFailure	Unable to perform find operation.
MDIErrDevice	Invalid device handle.
MDIErrSrcResource	Invalid Resource type.
MDIErrInvalidSrcOffset	SrcRange is invalid for the specified SrcResource.
MDIErrSrcOffsetAlignment	SrcOffset is not correctly aligned for the specified ObjectSize.
MDIErrSrcObjectSize	Invalid ObjectSize for the specified SrcResource.
MDIErrAbort	Command was aborted in response to an MDIAbort call.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Finds an optionally masked pattern in a resource. The resource address range is searched for a match or mismatch with a pattern consisting of *Count* values of size *ObjectSize*, possibly masked. *ObjectSize* must be non-zero. *Buffer* is an array of *Count* values to compare. *MaskBuffer* is the array of *Count* mask values to apply before comparing, or NULL if no masking is desired. The search can be forwards or backwards through the specified range. If a match is found, the starting offset of the match is returned in **FoundOffset*.

The MDILib is required to support this function only for memory-mapped resources, and up to a maximum Count of 256.

6.3.8 Query Cache: Retrieve cache attributes

```
MDIInt32
MDICacheQuery (  MDIHandleT Device,
                 MDICacheInfoT CacheInfo[2]);
```

Structures:

```
typedef struct MDICacheInfo_struct {
   MDIInt32   Type;
   MDIUint32  LineSize;     // Bytes of data in a cache line
   MDIUint32  LinesPerSet;  // Number of lines in a set
   MDIUint32  Sets;         // Number of sets
   MDIUint32  TagSize;      // Number of bytes in a cache tag entry
} MDICacheInfoT;
```

Returns:

MDISuccess	No Error, cache information has been returned.
MDIErrFailure	Unable to perform the query operation.

MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Retrieve the attributes of the caches, if present, on the target device. MDILibs are encouraged, but not required to return useful information.

Information is returned in the CacheInfo array for up to two caches. If it exists and the information is available, the first element will contain information about the primary unified or instruction cache and CacheInfo[0]. Type will be set to MDICacheTypeUnified or MDICacheTypeInstruction. If it exists and the information is available, the second element will describe a separate data cache and CacheInfo[1].Type will be set to MDICacheTypeData. If there is no such cache, or no information is available, the CacheInfo.Type member will be set to MDICacheTypeNone.

6.3.9 Get Cache Details: Get Information about the Specified Cache

```
MDIInt32
MDICacheInfo (   MDIHandleT Device,
                 MDIResourceT Resource,
                 MDICacheInfoT *CacheInfo)
```

Returns:

MDISuccess	No Error, cache information has been returned.
MDIErrFailure	Unable to perform the query operation.
MDIErrDevice	Invalid device handle.
MDIErrNoResource	The named resource is not a cache, or the named cache does not exist.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Retrieve the attributes of the cache specified by *Resource*, if present, on the target device. MDILibs are encouraged, but not required to return useful information. *CacheInfo* points to a single MDICacheInfoT structure. This function is a specialized version of the MDICacheQuery() function described above.

Note that unified caches should return information for instruction cache only, with *CacheInfoT->Type* set to MDICacheTypeUnified. In this case, MDIErrNoResource is returned for a data cache (since the unified cache resources share the same resource number as the instruction cache).

6.3.10 Cache Flush: Write back and/or invalidate the cache

```
MDIInt32
MDICacheFlush (  MDIHandleT Device,
                 MDIUint32 Type,
                 MDIUint32 Flags);
```

Structures:

MDICacheWriteBack	Write Back All Dirty Cache Lines if set.
MDICacheInvalidate	Invalidate All Cache Lines if set.

Returns:

MDISuccess	No Error, cache operation is complete.
MDIErrFailure	Requested cache operation cannot be performed.
MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Type is set to MDICacheTypeUnified, MDICacheTypeInstruction, or MDICacheTypeData to specify which cache to operate on. Flags indicate the operations to perform with values potentially OR'ed together, If Flags specifies both a write back and invalidate, the write back will happen before the invalidate.

6.3.11 Cache Operation: Do Specified Operation on Specified Cache

```
MDIInt32
MDICacheOp ( MDIHandleT Device,
             MDIResourceT Resource,
             MDIInt32 Type,
             MDIResourceT AddrResource,
             MDIOffsetT Offset,
             MDIUint32 Size);
```

Structures:

op:

MDICacheWriteBack	Write back dirty cache lines specified.	0x01
MDICacheInvalidate	Invalidate cache lines specified.	0x02
MDICacheWBInval	Write back and invalidate dirty cache lines specified.	0x03
MDICacheLock	Lock the lines of cache specified.	0x05
MDICacheHit	Do one of the above 4 operations for the virtual address range specified by offset and size.	0x00
MDICacheIndex	Do one of the above 4 operations for the cache line index specified by offset and set number specified by size.	0x80

Returns:

MDISuccess	No Error, cache operation is complete.
MDIErrFailure	Requested cache operation cannot be performed.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	The specified *op* flag is not supported.

MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Resource specifies which cache to operate on. *Type* indicates the operation to perform with one of MDICacheHit or MDICacheIndex OR'ed in. *AddrResource* specifies the address resource and the *Offset* depends on the type of operation, i.e. a virtual address for hit-type operations, or a cache line index for index-type operations. *Size* specifies the size of the address region in bytes, starting at the specified offset to which to apply the hit-type operation. For index-type operations, *Size* specifies the cache set (way) number. An invalid *Size* value is reported as MDIErrFailure.

6.3.12 Cache Sync: Synchronize the caches

```
MDIInt32
MDICacheSync (   MDIHandleT Device
                 MDIResourceT AddrResource,
                 MDIOffsetT Offset,
                 MDIUint32 Size);
```

Returns:

MDISuccess	No Error, cache operation is complete.
MDIErrFailure	Requested cache operation cannot be performed.
MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

This is a comprehensive synchronize call that for the entire cache hierarchy writes back dirty data cache lines and invalidates instruction caches. This routine allows the software to be worry-free with respect to the details of the cache hierarchy and the method by which the hierarchy can be brought to a known state. Some architectures provide a convenenient instruction or some other hardware mechanism by which to achieve this synchronization. This call is meant to invoke that architecture-specific mechanism. For example, in the MIPS32 Release 2 architecture, this routine would invoke the "SYNCI" instruction over the specified global or ASID virtual memory address range.

6.4 Run Control

The debugger requests device execution by calling MDIExecute() or MDIStep(). It must then periodically call MDIRunState() to monitor the status of the target until execution halts. If the CPU has not started running before MDIRunState completes, then it returns MDIStatusNotRunning. In general, the actual target execution will have begun by the time MDIExecute() returns and the requested number of steps will have been executed by the time MDIStep() returns, but this is not required to be the case. For example some types of target systems such as simulators may not behave this way. The actual execution may only take place during the MDIRunState() calls. Also, it is only during MDIRunState() calls that the MDILib is able to service I/O requests and other events that the target debug environment may support. Debuggers that do not support user operations while the target is executing will usually tell MDIRunState() to wait indefinitely; otherwise, the debugger should call MDIRunState() as frequently as possible with a fairly short wait interval.

6.4.1 Execute: Place the device into its RUNNING state

```
MDIInt32
MDIExecute (MDIHandleT Device)
```

Returns:

MDISuccess	No Error, device is in its RUNNING state.
MDIErrFailure	Device cannot be set to its RUNNING state.
MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

When the version of the API specified to MDIConnect() is 0x0002000C or lower then if there is a breakpoint set on the first instruction to be executed, it should not be taken. In other words, at least one instruction should always be executed as a result of an MDIExecute() call. If there are cases where this may not happen, the MDILib implementation must document the circumstances. When the selected API version is 0x0002000D or higher, a breakpoint at the first instruction must be honored, and it is up to the debugger to perform a single-step with breakpoints removed if it wants to step over a breakpoint.

The behavior of the device and its TCs (if any) upon returning to RUNNING state is governed by any previous calls to the MDISetRunMode() function, see Section 9.2, "Set Run Mode: Specify behavior when returning to the RUNNING state" on page 73. Multi-processor aware debuggers may use the MDITeamExecute() function, described in Section 9.3.8, "Team Execute: Place all team members into RUNNING state" on page 79.

6.4.2 Step: Single steps the device

```
MDIInt32
MDIStep ( MDIHandleT Device,
          MDIUint32 Steps,
          MDIUint32 Mode)
```

Structures:

MDIStepInto	Step Into
MDIStepForward	Step Forward
MDIStepOver	Step Over

Returns:

MDISuccess	No Error, stepping is initiated.
MDIErrFailure	Device refuses to single step.

MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Initiates the execution of the specified number of instructions in the specified mode.

In Step Into mode, there is no special handling of procedure calls, interrupts, traps or exceptions. If an interrupt or exception is pending when a step is initiated, and the target system supports stepping through interrupt handlers, the actual instruction stepped may be the first instruction in the handler rather than the instruction at the PC. In environments where interrupts are occurring faster than the time it takes to step through the interrupt handler, it may not be possible to make any progress in the foreground application in Step Into mode.

In Step Forward mode (also known as "step over traps"), the device ensures that each step operation executes an instruction in the foreground application. It may accomplish this by noticing when an interrupt is taken, and using breakpoints and full-speed execution to continue until the instruction at the original PC is executed. As a minimum this may be implemented simply by disabling interrupts while executing the target instructions.

In Step Over mode, the target system steps over procedure calls as well as interrupts and exceptions. If a procedure call instruction is being stepped, the called procedure is executed at full speed until it returns. This counts as one step. Support for Step Over mode is optional, since it is more usually implemented within the invoking debugger.

When the version of the API specified to MDIConnect() is 0x0002000C or lower then in any mode, if a breakpoint is encountered at any point after the first instruction is executed it is honored and execution stops. If there is a breakpoint set on the first instruction to be executed, it should not be taken--if there are cases where this may not happen, the MDILib implementation must document the circumstances. When the selected API version is 0x0002000D or higher, a breakpoint at the first instruction must be honored, and it is up to the debugger to perform a single-step with breakpoints removed if it wants to step over a breakpoint.

The MDIStep() function is now almost redundant, and when *Steps* is equal to 1 is equivalent to the following sequence of calls. See Chapter 9, “Multi-Threaded and Multi-Processor Command Set,” on page 71 for details.

```
MDITCIdT tcid;
if (MDIGetTC (Device, &tcid) != MDISuccess)
    tcid = -1;
MDISetRunMode (Device, tcid, Mode, 0);
MDIExecute (Device);
```

6.4.3 Stop: Stop execution of the device

```
MDIInt32
MDIStop (MDIHandleT Device)
```

Returns:

MDISuccess	No Error, device will attempt to stop.
MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.

MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

If the device is currently running, execution is halted. If the device is running, the debugger should still call MDIRunState to determine that it has successfully halted.

It must be possible to issue a stop to a device when it is disabled. It shall respond to the stop request as soon as it is enabled.

6.4.4 Abort: Terminate the current MDI function

```
MDIInt32
MDIAbort ( MDIHandleT Device)
```

Returns:

MDISuccess	No Error, current MDI Command is aborted.
MDIErrFailure	Not called from within debugger callback routine.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Abort is used from a debugger call back function to terminate the current MDI function. MDI functions that are thus terminated return MDIErrAbort.

6.4.5 Reset: Performs a target reset operation

```
MDIInt32
MDIReset (MDIHandleT Device,
          MDIUint32 Mode,
          MDIUint32 Flags)
```

Structures:

Mode:

MDIFullReset	Full Reset, reset entire target system if possible.
MDIDeviceReset	Device Reset, if device consists of a CPU plus peripherals, reset both if possible.
MDICPUReset	CPU Reset, if device consists of a CPU plus peripherals, reset just the CPU if possible.
MDIPeripheralReset	Peripheral Reset, if device consists of a CPU plus peripherals, reset just the peripherals if possible.

Flags:

MDINonIntrusive	Reset the target system and allow normal execution to continue unaffected by the debugger.

Returns:

MDISuccess	No Error, device has been reset and RunState has changed to RESET.

MDIErrFailure	Device refuses to reset.
MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Depending on the type of target system and the debug tool used to control it, there are several possible types of reset operations. The MDI specification supports the following reset concepts:

- MDIFullReset - A full reset of the entire target system. Normally, this means asserting a physical board-level reset signal that affects all components on the target board. Only hardware debug tools (ICEs) and board-level simulators are likely to support this reset option.
- MDIDeviceReset - A full reset of the target device (CPU/DSP and any associated on-chip peripheral circuitry). For typical single processor devices, including microcontroller and SoC devices, this may mean asserting a physical reset signal that is connected directly to the component rather than the entire board's reset circuit. For multi-processor devices where asserting a physical reset signal would reset all processors, the MDILib should treat MDIDeviceReset as a combination of MDICPUReset plus MDIPeripheralReset. In other words, it should use other means to reset or emulate resetting the specific CPU/DSP and peripheral logic being debugged, and assert the physical reset signal only as part of an MDIFullReset.
- MDICPUReset - Resets just the CPU/DSP being debugged. For microcontroller and SoC devices that support separate resetting of the processor and its associated peripheral logic, the peripheral logic is not reset. A reset issued to a VPE device within a multi-threaded CPU will reset the whole CPU, not just the specified device.
- MDIPeripheralReset - For microcontroller and SoC devices that support separate resetting of the processor and its associated peripheral logic, only the peripheral logic is reset. If there is no peripheral logic, or it can not be reset without also resetting the processor, nothing is done.

MDILibs are not required to implement all four modes as distinct operations. If the debugger requests an unsupported reset mode, the closest supported subset mode is performed instead. The MDILib must clearly document the supported modes and any mapping of unsupported modes.

Similarly, debuggers are not required to provide a user interface for all four modes. If the debugger supports only a single type of reset, it is recommended that it map this to the MDIDeviceReset mode.

6.4.6 State: Returns the current device execution status.

```
MDIInt32
MDIRunState (MDIHandleT Device,
             MDIInt32 WaitTime,
             MDIRunStateT *RunState);
```

Structures:

```
typedef struct MDIRunState_struct {
    MDIUint32    Status;
    union u_info
    {
        void        *ptr;
        MDIUint32   value;
    } Info;
} MDIRunStateT;
```

RunState->Status	*RunState->Info*
MDIStatusNotRunning	Not used.
MDIStatusRunning	Not used
MDIStatusHalted	Not used
MDIStatusExited	value = exit code
MDIStatusBPHit	value = BpID
MDIStatusUsrBPHit	Not used
MDIStatusException	value = exception code
MDIStatusStepsDone	Not used
MDIStatusTraceFull	Not used
MDIStatusDisabled	Not used.

Returns:

MDISuccess	No Error, RunState has been loaded with the device's current state.
MDIErrDevice	Invalid device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Returns the current device execution status in the MDIRunStateT structure pointed to by parameter, *RunState*. If the device is currently running, this function will wait a specified amount of time for the status to change. *WaitTime* specifies an approximate maximum amount of time to wait, in milliseconds. If *WaitTime* is 0 or the device is not running, the current status is returned immediately. If it is MDIWaitForever, MDIRunState will wait indefinitely for the device to stop running. Otherwise, *WaitTime* specifies an approximate time to wait before returning the status. If the device status changes before the time period expires, MDIRunState() will return the new status immediately.

If the device status has not changed since the last call to MDIRunState, *RunState->Status* will be set to MDIStatusNotRunning, MDIStatusRunning, or MDIStatusDisabled. It may take some finite time for a device to change its status from MDIStatusNotRunning to MDIStatusRunning, and a debugger must be willing to wait and timeout this transition.

If the target has stopped execution since the last call, *RunState->Status* will be set to one of the other codes to indicate the cause of the halt. MDIStatusExited means that the target program terminated itself by calling exit or a similar system service. MDIStatusBPHit means that a breakpoint set by the debugger was taken. MDIStatusUsrBPHit means that the target was halted by the breakpoint mechanism, but not at a breakpoint set by the debugger. MDIStatusException means that the target program took an unexpected interrupt/trap/exception. Exception codes are architecture specific. MDIStatusStepsDone means that the number of steps requested in the MDIStep() call have been completed. MDIStatusTraceFull means that execution halted due to filling up the trace buffer. MDIStatusHalted is returned for all other halt reasons, including being halted in response to an MDIStop() call.

MDIStatusDisabled means that the device can neither execute target code nor enter the halted state. In this state only MDIReset(), MDIStop() and MDIRunState() may have any useful effect on the device. This status would occur, for example, when connected to a VPE does not yet have any TCs bound to it.

There are also three flag values that can be OR'ed with the MDIStatusRunning, MDIStatusNotRunning, and MDIStatusHalted values in RunState->Status to provide additional information. They are:

MDIStatusReset	currently held reset; this should typically be reported by the debugger
MDIStatusWasReset	reset was asserted & released
MDIStatusDescription	RunState->Info.ptr points to a descriptive string

MDIStatusReset will be combined with MDIStatusRunning if the device may resume execution at any time by the release of Reset MDIStatusReset or MDIStatusWasReset will be combined with MDIStatusNotRunning or MDIStatusHalted if the execution was halted due to a target reset but will not be resumed until the next MDIExecute() call.

6.5 Breakpoints

The following data structure is used to fully describe a breakpoint being set or queried:

```
typedef struct MDIBpData_struct {
    MDIBpIdT      Id;        // Unique ID assigned by MDISetBp()
    MDIBpT        Type;      // Breakpoint type
    MDIUint32     Enabled;   // 0 if currently disabled, else 1
    MDIResourceT  Resource;
    MDIRangeT     Range;     // Range.End may be an end addr or mask
    MDIUint64     Data;      // valid only for data read/write breaks
    MDIUint64     DataMask;  // valid only for data read/write breaks
    MDIUint32     PassCount;// Pass count reloaded when hit
    MDIUint32     PassesToGo;// Passes to go until next hit
} MDIBpDataT;
```

Id is a unique ID assigned by MDISetBp or MDISetSWBp and used to specify a particular breakpoint for the other calls. The reserved value MDIAllBpID (-1) may not be used as a breakpoint ID. *Type* is the breakpoint type. The debugger can specify one of the following breakpoint types to MDISetBp:

Type:

MDIBPT_SWInstruction	Is an instruction execution breakpoint. Execution stops when control reaches the instruction at the address specified. The address is specified by the combination of the *Resource* field and the *Range.Start* field. The *PassCount* value specifies the number of times to pass by the break condition before actually halting. The values that make the most sense for an architecture and MDILib implementation can be found in the architecture addendum as well as in the documentation for the specific MDILib. This breakpoint type is usually implemented by inserting a special instruction in memory--in which case all devices which share memory and execute the instruction will enter debug mode, where it will be reported as MDIStatusUsrBpHit to all except the devices which have requested a software breakpoint at that address.
MDIBPT_SWOneShot	A temporary Instruction execution breakpoint. Like `MDIBPT_SWInstruction`, except that *PassCount* is not applicable and the breakpoint is deleted automatically once execution stops for any reason. This breakpoint type is useful for the common "run to cursor" debugger function.
MDIBPT_HWInstruction	A Hardware Instruction breakpoint. Target devices that provide hardware breakpoint capabilities may allow execution to be halted when an instruction or range of instructions is fetched or executed.

MDIBPT_HWData	A Hardware Data access breakpoint. Target devices that provide hardware breakpoint capabilities may allow execution to be halted when a datum is loaded or stored at a particular address or range of addresses.
MDIBPT_HWBus	A Hardware Data bus breakpoint. Target devices that provide hardware breakpoint capabilities may allow execution to be halted when certain bus transactions are detected.
MDIBPT_HWInstructionPrimed	This and the next 5 breakpoint types belong to the Complex Break and Trigger (CBT) architectural extension to the EJTAG breakpoint types. All these breakpoint types must specify the index of the instruction (inst0..N) or data (data0..N) breakpoint using the MDIBPT_IndexMask value since unlike regular instruction or data breakpoints, the CBT breakpoints have different settings depending on their position. For example, for the primed breakpoint types the priming conditions for inst0, inst1, data0, data1, etc. are different. Similarly, the qualifying condition for each qualified breakpoint is different and implementation-dependent. Since primed, qualified, and tuples are breakpint types, the standard MDI routines for query, enable, disable, and clear apply. This represents the CBT primed instruction breakpoints.
MDIBPT_HWInstructionQualified	This represents a CBT qualified instruction breakpoints.
MDIBPT_HWInstructionTuple	This represents a CBT instruction breakpoint of a tuple-type.
MDIBPT_HWDataPrimed	This is a CBT primed data breakpoint.
MDIBPT_HWDataQualified	This is a CBT qualified data breakpoint.
MDIBPT_HWDataTuple	This is a CBT data breakpoint of a tuple-type.

All three Hardware breakpoint types may have one or more of the following flag bits OR'ed in to specify additional qualifications:

MDIBPT_HWFlg_AddrMask	The break address in *Range.Start* and the actual address are masked by the value in *Range.End* before being compared. Non-zero bits in the mask indicate address and data bits to be masked (excluded) from the comparison. Zero bits indicate that the corresponding address bits are to be compared.
MDIBPT_HWFlg_AddrRange	Any address in the range from *Range.Start* to *Range.End* will trigger the break.
MDIBPT_HWFlg_Trigger	If the target device supports it, matching the break condition should cause a "trigger" signal to be generated. This is intended to be used with probes and emulators that provide an external trigger signal for connection to other devices, such as logic analyzers (or vice-versa).
MDIBPT_HWFlg_TriggerOnly	Like `MDIBPT_HWFlg_Trigger`, except that device execution should not actually stop when the break condition is met. If this flag is set, the MDIBPT_HWFlg_Trigger flag is also implied and its actual value is ignored.
MDIBPT_HWFlg_TCMatch	If the target device is multi-threaded, then by default a Hardware break will occur when accessed by any thread context (TC). But when this flag is set the break will occur only when accessed by the TC which was "current" when MDISetBp() was called. On a multi-VPE processor all hardware breakpoints should be specific to the device which request them, and should not trigger on accesses by other VPE devices.

MDIBPT_HWFlg_InvertMatch	This flag supports the IVM (invert value match) bit, which is valid for data breakpoints and permits match inversion to be notified.

The Data and Bus Hardware breakpoint types may also have one or more of the following flag bits OR'ed in to specify additional qualifications:

MDIBPT_HWFlg_DataValue	The break will occur only if the data value specified in *Data* is read from and/or written to the break address.
MDIBPT_HWFlg_DataMask	The mask value specified in *DataMask* is applied to the data value before comparison. Non-zero bits in the mask indicate corresponding address bits to be excluded from the comparison.
MDIBPT_HWFlg_DataRead	The break will occur on read accesses.
MDIBPT_HWFlg_DataWrite	The break will occur on write accesses.

If neither `MDIBPT_HWFlg_DataRead` nor `MDIBPT_HWFlg_DataWrite` is specified, the effect is the same as if both are specified - the break will occur on any access type, read or write.

`PassCount` specifies the number of times the break condition must be satisfied before device execution is stopped and the halted status reported back to the debugger. For example, a software breakpoint with *PassCount* set to one will be taken every time the breakpoint condition is met, but if it is set to ten, then the break will be taken every tenth time the break condition is met. If *PassCount* is set to zero, then MDISetBp() will assume a pass count value of one.

All MDILib implementations are required to support the two software breakpoint types. Support for the hardware breakpoint types depends on the capabilities of the target device and is therefore optional. If an unsupported type of hardware breakpoint is requested, MDISetBp() will return MDIErrUnsupported.

The maximum number of breakpoints of a particular type that can be set also depends on the underlying capabilities of the target device. With some devices the limit, if any, may not even be known to the MDILib implementation; therefore MDI does not specify a minimum number of breakpoints that MDILib implementations must support. If an attempt to set a breakpoint exceeds a capacity limit, MDISetBp() and MDISetSWBp() will return MDIErrNoResource.

6.5.1 Set Full Breakpoint

```
MDIInt32
MDISetBp (MDIHandleT Device,
          MDIBpDataT *BpData)
```

Structures:

```
typedef struct MDIBpData_struct {
   MDIBpIdT       Id;
   MDIBpT         Type;
   MDIUint32      Enabled;    /* 0 if currently disabled, else 1 */
   MDIResourceT   Resource;
   MDIRangeT      Range;      /* Range.End may be an end addr or mask */
   MDIUint64      Data;       /* valid only for data write breaks */
   MDIUint64      DataMask;   /* valid only for data write breaks */
   MDIUint32      PassCount;  /* Pass count reloaded when hit */
   MDIUint32      PassesToGo; /* Passes to go until next hit */
} MDIBpDataT;
```

Returns:

MDISuccess	No Error, BpData->Id has been set to the handle needed to reference this specific breakpoint.
MDIErrDevice	Invalid device handle.
MDIErrBPType	Invalid breakpoint type. Must be one of the basic 5 types defined.
MDIErrDstResource	Invalid Resource type.
MDIErrUnsupported	The device doesn't support the type of breakpoint requested.
MDIErrRange	Specified range is outside the scope for the given resource.
MDIErrNoResource	The resources needed to implement the request are not available.
MDIErrDuplicateBP	A similar breakpoint has already been defined for this device, or for global breakpoints on any device
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Setup a breakpoint from a full specification, return a unique breakpoint ID that will be used to refer to the breakpoint in other calls. On entry, *BpData* members *Type*, *Enabled*, *Resource*, and *Range.Start* must be initialized for all breakpoint types. *PassCount* must be initialized for all breakpoint types except `MDIBPT_SWOneShot`. For hardware breakpoints with the `MDIBPT_HWFlg_AddrMask` or `MDIBPT_HWFlg_AddrRange` attribute, *Range.End* must be initialized. For data breakpoints with the `MDIBPT_HWFlg_DataWrite` and `MDIBPT_HWFlg_DataValue` attributes, *Data* must be initialized. *PassesToGo* is ignored by `MDISetBp`. If `MDIBPT_HWFlg_DataMask` is also set, *DataMask* must be initialized.

If the breakpoint is set successfully, MDISetBp() will set *BpData->Id* to the breakpoint ID it assigned. No other members of **BpData* will be modified by MDISetBp(). When a breakpoint will trigger on all devices which reference the same memory address (e.g. a software breakpoint) the MDIlib returns an MDIErrDuplicateBp error if another device has a similar breakpoint at the same address.

6.5.2 Set Software Breakpoint

```
MDIInt32
MDISetSWBp ( MDIHandleT Device,
             MDIResourceT Resource,
             MDIOffsetT Offset,
             MDIBpIdT *BpId)
```

Returns:

MDISuccess	No Error, *BpId has been set to the handle needed to reference this specific breakpoint. The breakpoint is set to the enabled state, with PassCount set to 1.
MDIErrDevice	Invalid device handle.
MDIErrDstResource	Invalid Resource type.
MDIErrRange	Specified range is outside the scope for the given resource.
MDIErrNoResource	The resources needed to implement the request are not available.

MDIErrDuplicateBP	A similar breakpoint has already been defined for this device, or for global breakpoints on any device.
MDIErrNoResource	The resources needed to implement the request are not available.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Set up an enabled breakpoint of type `MDIBPT_SWInstruction` with a pass count of one. Since this is expected to be the most common operation, this simpler form of MDISetBp is provided as "syntactic sugar" for the debugger.

If the breakpoint is set successfully, MDISetSWBp will set `*BpId` to the breakpoint ID it assigned. When a software breakpoint will trigger on all devices which execute the same memory address, the MDILib must return an MDIErrDuplicateBP error when another device has a software breakpoint set at the same address.

6.5.3 Clear Breakpoint

```
MDIInt32
MDIClearBp (  MDIHandleT Device,
              MDIBpIdT BpId)
```

Returns:

MDISuccess	No Error, all breakpoints or breakpoint BpId has been removed.
MDIErrDevice	Invalid device handle.
MDIErrBPId	Invalid Breakpoint ID.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Clears a specified breakpoint, or clear all breakpoints on the selected device using a BpId value of MDIAllBpID.

6.5.4 Enable Breakpoint

```
MDIInt32
MDIEnableBp (MDIHandleT Device,
             MDIBpIdT BpId)
```

Returns:

MDISuccess	No Error, breakpoint BpId has been enabled.
MDIErrDevice	Invalid device handle.
MDIErrBPId	Invalid Breakpoint ID.

MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Enables a breakpoint on the selected device. Enabling a previously disabled breakpoint does not affect its PassesToGo value. A *BpId* value of MDIAllBpID will enable all breakpoints.

6.5.5 Disable Breakpoint

```
MDIInt32
MDIDisableBp (   MDIHandleT Device,
                 MDIBpIdT BpId)
```

Returns:

MDISuccess	No Error, breakpoint BpId has been disabled.
MDIErrDevice	Invalid device handle.
MDIErrBPId	Invalid Breakpoint ID.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Disables a breakpoint on a selected device. A disabled breakpoint will not affect target execution and its PassesToGo value will not be decremented, until it is re-enabled. Its current PassesToGo value will remain in effect when it is re-enabled. A *BpId* value of MDIAllBpID will disable all breakpoints.

6.5.6 Query Breakpoints

```
MDIInt32
MDIBpQuery (  MDIHandleT Device,
              MDIInt32 *HowMany,
              MDIBpDataT BpData)
```

Returns:

MDISuccess	No Error, information for a single breakpoint or all breakpoints is returned.
MDIErrDevice	Invalid device handle.
MDIErrBPId	Invalid Breakpoint ID.
MDIErrMore	More breakpoints defined then requested.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.

MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Queries the set of defined breakpoints on a selected device.

If the requested number of breakpoints (**HowMany*) is 0, then the function returns no error (MDISuccess) and **HowMany* is set to the number of defined breakpoints. If **HowMany* is set to fewer breakpoints than there are set, then the **HowMany* value is modified upon return to indicate the number of returned breakpoints. If more breakpoints are set than the value specified in **HowMany*, then MDIErrMore is returned. In this situation, if another MDIBpQuery() call is made before any other calls to the MDILib, more breakpoints are returned as requested by the **HowMany* value.

6.5.7 Hardware Breakpoint Query: Retrieve a list of supported hardware breakpoint types

```
MDIInt32
MDIHwBpQuery (   MDIHandleT Device,
                 MDIInt32 *HowMany,
                 MDIBpInfoT *BpInfo)
```

Structures:

```
typedef struct MDIBpData_struct {
   MDIInt32       Num;
   MDIBpT        Type;
} MDIBpInfoT
```

Type is a bitmap composed of some of these new values:

```
#define MDIBPT_HWType_Exec        0x00000001 // bpt on execute supported
#define MDIBPT_HWType_Data        0x00000002 // bpt on data access supported
#define MDIBPT_HWType_Bus         0x00000004 // bpt on ext h/w access supported
#define MDIBPT_HWType_AlignMask   0x000000F0 // min addr alignment (2^n)
#define MDIBPT_HWType_AlignShift  4
#define MDIBPT_HWType_MaxSMask    0x00003F00 // max size (2^n)
#define MDIBPT_HWType_MaxSShift   8
#define MDIBPT_HWType_VirtAddr    0x00004000 // matches on virtual address
#define MDIBPT_HWType_ASID        0x00008000 // ASID included in virtual address
```

Some breakpoint defines that already exist in MDI are:

```
#define MDIBPT_HWFlg_AddrMask    0x00010000 // address mask supported
#define MDIBPT_HWFlg_AddrRange   0x00020000 // address range supported
#define MDIBPT_HWFlg_DataValue   0x00040000 // data value match supported
#define MDIBPT_HWFlg_DataMask    0x00080000 // data masking supported
#define MDIBPT_HWFlg_DataRead    0x00100000 // bpt on data read supported
#define MDIBPT_HWFlg_DataWrite   0x00200000 // bpt on data write supported
#define MDIBPT_HWFlg_Trigger     0x00400000 // ext trigger output supported
#define MDIBPT_HWFlg_TriggerOnly 0x00800000 // ext trigger only supported
#define MDIBPT_HWFlg_TCMatch     0x01000000 // Set bpt for specified TC
#define MDIBPT_HWFlg_InvertMatch 0x02000000 //"not" match
```

Returns:

MDISuccess	No Error, information for a single breakpoint or all breakpoints is returned.

MDIErrDevice	Invalid device handle.
MDIErrParam	Invalid parameter, *HowMany may not be negative
MDIErrMore	More breakpoints defined then requested.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Description:

Queries the available hardware breakpoint resources of the target device.

If the requested number of breakpoint resource (**HowMany*) is 0, the function returns no error (MDISuccess) and **HowMany* is set to the number of types. If **HowMany* is non-zero on entry, it specifies the number of elements in the *BPInfo* array being passed in. The function fills in the *BPInfo* array with the information for up to **HowMany* breakpoint resources and sets **HowMany* to the number filled in. If there is not enough room in the *BPInfo* array to hold all the available resource, MDIErrMore is returned. If the debugger then calls this function again before any other MDI functions are called, information is returned for the next **HowMany* breakpoint resources.

MDIBpInfoT->Type is a bitmap that specifies the exact type of hardware breakpoint supported, and *MDIBpInfo->.Num* is the number of breakpoints that support this combination of features. If *MDIBpInfoT->Num* has a value of -1, then it supports an infinite number of such breakpoints (as might easily be the case for a simulator).

For hardware breakpoints that support only address masking and not address ranges, the MDILib is encouraged to virtualize support for an address range. In other words, it should generate the smallest mask which surrounds a given address range, and then check the address which causes a data breakpoint and only return control to the debugger if the address is indeed in the originally requested range. This may involve disassembling the faulting instruction to determine the data address.

Example 1: A MIPS 4Kc core with 2 coprocessor 0 data/instruction watchpoints would return:

```
*HowMany =1;

BpInfo[0].Num = 2;
BpInfo[0].Type =(MDIBPT_HWType_Exec                |
                 MDIBPT_HWType_Data                |
                 (3 << MDIBPT_HWType_AlignShift)   |
                 (12 << MDIBPT_HWType_MaxSShift)   |
                 MDIBPT_HWType_VirtAddr            |
                 MDIBPT_HWType_ASID                |
                 MDIBPT_HWFlg_AddrMask             |
                 MDIBPT_HWFlg_DataRead             |
                 MDIBPT_HWFlg_DataWrite);
```

Example 2: A MIPS 4Kc core with 2 data and 4 instruction EJTAG hardware breakpoints would return:

```
*HowMany =2;

BpInfo[0].Num = 2;
BpInfo[0].Type =(MDIBPT_HWType_Data                |
                 (0 << MDIBPT_HWType_AlignShift)   |
                 (31 << MDIBPT_HWType_MaxSShift)   |
                 MDIBPT_HWType_VirtAddr            |
                 MDIBPT_HWType_ASID                |
```

```
                        MDIBPT_HWFlg_AddrMask                |
                        MDIBPT_HWFlg_DataValue               |
                        MDIBPT_HWFlg_DataMask                |
                        MDIBPT_HWFlg_DataRead                |
                        MDIBPT_HWFlg_DataWrite               |
                        MDIBPT_HWFlg_Trigger                 |
                        MDIBPT_HWFlg_TriggerOnly);
    BpInfo[1].Num = 4;
    BpInfo[1].Type =(MDIBPT_HWType_Exec                      |
                     (1 << MDIBPT_HWType_AlignShift)         |
                     (31 << MDIBPT_HWType_MaxSShift)         |
                     MDIBPT_HWType_VirtAddr                  |
                     MDIBPT_HWType_ASID                      |
                     MDIBPT_HWFlg_AddrMask                   |
                     MDIBPT_HWFlg_Trigger                    |
                     MDIBPT_HWFlg_TriggerOnly);
```

Example 3: A simulator that supports an "unlimited" number of hardware breakpoints, with unrestricted address range would return:

```
    *HowMany =1;

    BpInfo[0].Num = -1;
    BpInfo[0].Type =(MDIBPT_HWType_Exec                      |
                     MDIBPT_HWType_Data                      |
                     (0 << MDIBPT_HWType_AlignShift)         |
                     (63 << MDIBPT_HWType_MaxSShift)         |
                     MDIBPT_HWType_VirtAddr                  |
                     MDIBPT_HWType_ASID                      |
                     MDIBPT_HWFlg_AddrRange                  |
                     MDIBPT_HWFlg_DataValue                  |
                     MDIBPT_HWFlg_DataMask                   |
                     MDIBPT_HWFlg_DataRead                   |
                     MDIBPT_HWFlg_DataWrite                  |
                     MDIBPT_HWFlg_TCMatch);
```

Chapter 7

MDILib and Target I/O Command Set

The goal of MDI is to allow interoperability between any debugger written in conformance with this specification and any conforming MDILib implementation; however, no generic API specification can envision and abstract all possible device behavior. There are many possible types of devices (simulators, device resident debug kernels, JTAG/BDM probes, ICE's, etc.) with a wide range of possible capabilities and configuration requirements. To allow for non-standard services and responses in a standard way, MDI provides mechanisms for MDILib specific commands to be executed, and requires the debugger to provide character input and output services to the MDILib. To further support MDILib command parsing and output formatting, the debugger is strongly encouraged to provide expression evaluation and symbolic lookup services to the MDILib.

The required input and output services also serve as a communication channel between the user and the program running on the target device.

7.1 Execute Command: Do the command specified

```
MDIInt32
MDIDoCommand (   MDIHandleT Device,
                 char *Buffer)
```

Returns:

MDISuccess	No Error, command has been executed.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	MDILib has no command parser.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

A single command string is passed to the MDILib for parsing and execution. If an MDILib has no command parser, then it will set the `MDICap_NoParser` flag in *Config->MDICapability* and this function will do nothing; otherwise, the debugger is required to provide a mechanism for the user to provide command lines to be passed to the MDILib via this function without interpretation by the debugger.

Device will be MDINoHandle if the command is not associated with a particular device connection. This would be the case for calls to MDIDoCommand() made before MDIOpen() has been called.

7.2 Display Output: Display the MDILib supplied text to the user

```
MDIInt32
MDICBOutput ( MDIHandleT Device,
              MDIInt32 Type,
              char *Buffer,
              MDIInt32 Count)
```

Returns:

MDISuccess	No Error, output has been displayed.
MDIErrDevice	Invalid device handle.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Structures:

Type:

MDIIOTypeMDIOut	"stdout" from the MDILib
MDIIOTypeMDIErr	"stderr" from the MDILib
MDIIOTypeMDINotify	GUI notification from MDILib
MDIIOTypeTgtOut	"stdout" from the running target program
MDIIOTypeTgtErr	"stderr" from the running target program

Description:

This callback function is implemented by the debugger. Its address is passed to the MDILib in *Config->MDICBOutput* when MDIConnect() is called. The debugger must display the MDILib-supplied text to the user. The debugger may choose to display the various types of output in different ways, for example putting MDILib output and program output in separate windows, or displaying MDILib error output in a pop-up dialog.

This function can be called only when the MDILib is servicing a debugger request; in other words, it cannot be called asynchronously, it is only called recursively after the debugger has made any one of the MDILib calls.

Device will be MDINoHandle if the output is not associated with a particular device connection. *Type* specifies the type of output. Count is the number of characters in Buffer. There is no specific limit to the length of the character data. The data may include LF characters to signal desired line breaks, but no other non-printable ASCII characters are allowed. The data might not end with an LF, for example the MDILib might be displaying a prompt to be followed by a request for input. While the debugger is encouraged to honor line breaks it is not required. The MDIIOTypeMDINotify code specifically requests a GUI to display pop-up warning dialog requiring an acknowledgement (OK) by the user before control is returned to the MDILib. If no GUI is present, treat as MDIIOTypeMDIErr.

7.3 Get Input

```
MDIInt32
MDICBInput ( MDIHandleT Device,
             MDIInt32 Type,
             MDIInt32 Mode,
             char **Buffer,
             MDIInt32 *Count)
```

Returns:

MDISuccess	No Error, input has been obtained.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Debugger does not support non-blocking and/or unbuffered input.

Structures:

Type:

MDIIOTypeMDIIn	"stdin" for the MDILib
MDIIOTypeTgtIn	"stdin" for the running target program

Mode:

MDIIOModeNormal	blocking, line buffered
MDIIOTypeRawBlock	blocking unbuffered
MDIIOTypeRawNoBlock	non-blocking unbuffered (can return *Count == 0)

Description:

This callback function is implemented by the debugger. and its address is passed to the MDILib in *Config->MDICBInput* when MDIConnect() is called. The debugger must get up to a line of character input from the user and deliver it to the MDILib. The characters entered by the user are not to be interpreted or modified by the debugger, except for the end-of-line.

This function can be called only when the MDILib is servicing a debugger request. In other words, it can not be called asynchronously, it is only called recursively after the debugger has made an MDILib call.

Device will be MDINoHandle if the input request is not associated with a particular device connection. The debugger supplies the buffer holding the data, and returns its address to the MDILib in **Buffer*, and returns the number of characters it contains in **Count*. *Type* specifies the type of input. *Mode* specifies the mode of the input. In buffered mode, only a single line is returned per call on MDICBInput, but there is no specific limit to the length of the line. In non-blocking unbuffered mode, the data available at the time of the call is returned. In blocking unbuffered mode, the debugger will return as soon as any input is available (typically one character, but possibly more due to a "paste" event for example).

The debugger is encouraged to support all three modes, but is only required to support MDIIOModeNormal.

7.4 Evaluate Expression

```
MDIInt32
MDICBEvaluate (  MDIHandleT Device,
                 char *Buffer,
                 MDIInt32 *ResultType,
                 MDIResourceT *SrcResource,
                 MDIOffsetT *SrcOffset,
                 MDIInt32 *Size,
                 void **Value)
```

Returns:

MDISuccess	No Error, expression result has been returned.
MDIErrDevice	Invalid device handle.
MDIErrFailure	Expression could not be evaluated.

Structures:

ResultType:

MDIEvalTypeResource	Address is returned in *SrcResource,*SrcOffset.

MDIEvalTypeChar	Result is a single character.
MDIEvalTypeInt	Result is a signed int of size *Size.
MDIEvalTypeUInt	Result is an unsigned int of size *Size.
MDIEvalTypeFloat	Result is a floating point value of size *Size.
MDIEvalTypeNone	Result of size *Size has no type, or the debugger does not support types.

Description:

This callback function is optionally implemented by the debugger. Its address, or NULL if it is not implemented, is passed to the MDILib in *Config->MDICBEvaluate* when MDIConnect() is called. The purpose of this callback is to allow the MDILib command parser to support expressions which will be evaluated according to the debugger's rules. The debugger is encouraged but not required to provide this service.

This function can be called only when the MDILib is executing a transparent mode command. In other words, it can not be called asynchronously, it is only called recursively after the debugger has called MDIDoCommand(). During the course of evaluating the expression, the debugger may need to access device resources so it may recursively call other MDI functions before returning.

The expression may evaluate to a scalar value, or it may evaluate to an addressable resource. The debugger indicates which by returning the appropriate value in **ResultType*.

If the result is a scalar value, then the debugger stores the value in host byte order in a buffer whose address and size is returned in **Buffer* and **Size*.

7.5 Lookup Resource

```
MDIInt32
MDICBLookup (MDIHandleT Device,
             MDIInt32 Type,
             MDIResourceT SrcResource,
             MDIOffsetT SrcOffset,
             char **Buffer)
```

Returns:

MDISuccess	No Error, string has been returned.
MDIErrDevice	Invalid device handle.
MDIErrLookupNone	Address did not match a symbol or source line.
MDIErrLookupError	Invalid address for look up.

Structures:

Type:

MDILookupNearest	Debugger - returns "sym" on exact match, or "sym+delta", where sym is the nearest symbol with a lower address and delta is the offset from the symbol's address to the requested address, in hex.
MDILookupExact	Debugger - returns "sym" on exact match only.

MDILookupSource	Debugger - returns the source line associated with the resource address, if any. This is intended to be an “exact match” lookup. The debugger should return a source line only for the first of a group of instructions generated by the source line. Support for this lookup is optional.

Description:

This callback function is optionally implemented by the debugger. Its address, or NULL if it is not implemented, is passed to the MDILib in *Config->MDICBLookup* when MDIConnect() is called. The purpose of this callback is to allow the MDILib command parser to decorate command output with symbolic information. The MDILib passes a request type and an address. The debugger generates the requested type of ASCII string into a static buffer, and returns the address of the buffer to the MDILib. The debugger is encouraged but not required to provide this service.

This function can be called only when the MDILib is executing a transparent mode command. In other words, it can not be called asynchronously; only called recursively after the debugger has called MDIDoCommand. It is not expected that the debugger would need to access target resources to perform the lookup, but it is allowed to do so; thus it may recursively call other MDI functions before returning.

The MDILib requests a particular type of symbolic information by passing one of the values for *Type* specified above.

If the lookup is successful, then the debugger returns the address of a buffer containing the resulting NULL terminated ASCII string in **Buffer*. The pointer must remain valid and the contents of the buffer must remain unchanged only until the MDILib calls another callback function or returns from MDIDoCommand(), whichever comes first. The MDILib must not make any further use of the returned pointer after that time.

Chapter 8

Trace Command Set

It is often the case that a device provides some type of trace output reporting on the status of the code being executed. For example, the MIPS EJTAG specification includes the ability to shift out execution status and PC address information as the processor runs. An Instruction Set Simulator could obviously record execution activity, and bus tracing is supported by many ICE vendors.

Since it would be desirable to allow a debugger to display trace information in a well-integrated way, MDI includes an abstraction for tracing services; however, the actual capabilities and features of any particular device that supports tracing will vary widely. It is not possible to create a standard API that will provide full access to all possible tracing systems; therefore, MDI only provides a binary abstraction for the lowest common denominator: a sequence of PC and possibly data addresses and optionally the associated instructions/values. An MDILib can provide its own user interface for extended functions.

Since not all devices will be capable of generating trace information, support for the Trace Data command set is optional in the MDILib. The MDILib will set the `MDICap_TraceOutput` flag in *Config->MDICapability* if it supports the MDITraceClear, MDITraceStatus, MDITraceCount, and MDITraceRead functions. The MDILib will set the `MDICap_TraceCtrl` flag in *Config->MDICapability* if it supports the MDITraceEnable, and MDITraceDisable functions.

If the underlying hardware implements the MIPS PDtrace™ interface, then the MDI library has the option to support the interface required to access this capability. This is indicated by the `MDICap_PDtrace` flag in *Config->MDICapability.* The interface primarily consists of a set of three new trace-related calls that are described at the end of this chapter, from Section 8.7, "Read PDtrace Data" to Section 8.9, "Set PDtrace Mode". In addition to the new subroutine calls, a new include file is needed, mdi_pdtrace.h, which is specified in the Appendix.

Finally, to support the Trace Control Block (TCB) that would attach to one end of a PDtrace interface, two functions that set and get the trigger conditions are provided. The mdi_tcb.h file is provided in the Appendix of this document.

8.1 Enable Tracing

```
MDIInt32
MDITraceEnable (MDIHandleT Device)
```

Returns:

MDISuccess	No Error, tracing has been enabled.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function enables the tracing capabilities of the device. MDI assumes that when tracing is enabled, trace data is captured only when the device is executing code. Thus, it is not necessary for the debugger to explicitly disable tracing after execution stops in order to avoid capturing unwanted data. It is valid for the debugger to enable tracing at the start of the session, and leave it enabled from then on. This means that for devices whose actual tracing capabilities are not tied to execution (e.g. a logic analyzer), must be managed by the MDILib to emulate this "execution tracing".

It is unspecified whether enabling tracing causes any previously captured trace data to be cleared from the device's trace buffer. Further, it is unspecified whether captured trace data is automatically cleared each time device execution begins.

8.2 Disable Tracing

```
MDIInt32
MDITraceDisable (MDIHandleT Device)
```

Returns:

MDISuccess	No Error, tracing has been disabled.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function disables the tracing capabilities of the device. If the device is currently executing code, tracing will be halted immediately, and depending on the capabilities of the tracing system, it may be necessary for the MDILib to temporarily halt execution in order to disable trace capture.

8.3 Clear Trace Data

```
MDIInt32
MDITraceClear (MDIHandleT Device)
```

Returns:

MDISuccess	No Error, the trace buffer has been cleared.
MDIErrDevice	Invalid device handle.
MDIErrTracing	Device is currently tracing.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function causes any previously captured trace data to be cleared from the device's trace buffer. If tracing is enabled and the device is currently executing code, then the debugger must call MDITraceDisable before this function can be called.

8.4 Query Trace Status

```
MDIInt32
MDITraceStatus (MDIHandleT Device,
                MDIUint32 *Status)
```

Returns:

MDISuccess	No Error, the current trace status has been returned.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function returns the current state of the tracing system. Many devices will support mechanisms to qualify tracing, such as beginning or ending capture when a trigger event is detected, or ending capture when the trace buffer becomes full. While MDI can not abstract an interface for configuring such trace capabilities, the debugger should recognize that they may exist. If the debugger supports fetching and displaying trace data while the device is executing, then it should use this function to determine when it is appropriate to do so.

On return, *Status will contain one of the following values:

MDITraceStatusNone	Tracing is not enabled, or the device is not executing.
MDITraceStatusTracing	Tracing underway, with no termination condition.
MDITraceStatusWaiting	Conditional trace capture has not yet begun.
MDITraceStatusFilling	Tracing, with conditional completion expected.
MDITraceStatusStopped	Conditional trace capture has completed.

If no trace conditions are configured, or the device does not support triggered/conditional tracing, then MDITraceStatus will return MDITraceStatusTracing if MDITraceEnable has been called and the device is executing; otherwise it will return MDITraceStatusNone. MDITraceStatusWaiting will be returned when a conditional trigger event has been configured that causes trace capture to begin, and the event has not yet occurred. MDITraceStatusFilling is returned if trace capture has begun, and a conditional trigger event has been configured that can terminate trace capture before the device stops executing. Finally, MDITraceStatusStopped is returned after such a condition has occurred and no more trace data will be captured.

8.5 Query Trace Data

```
MDIInt32
MDITraceCount (  MDIHandleT Device,
                 MDIUint32 *FrameCount)
```

Returns:

MDISuccess	No Error, the frame count has been returned.
MDIErrDevice	Invalid device handle.
MDIErrTracing	Device is currently tracing.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function returns the number of "frames" of trace data currently captured by the device. Although it may be called at any time when the device is not executing, and when tracing is disabled if the device is executing. If tracing is enabled and the device is currently executing code, then the debugger must call MDITraceDisable before this function can be called. A "frame" of trace data describes a single instruction or data access performed by the target. A "frame" of trace data in the PDtrace context is the number of words of trace data. The debugger must call this function before calling MDITraceRead() or MDIPDtraceRead() to transfer actual trace data.

8.6 Read Trace Data

```
MDIInt32
MDITraceRead (   MDIHandleT Device,
                 MDIUint32 FirstFrame,
                 MDIUint32 FrameCount,
                 MDIUint32 IncludeInstructions,
                 MDITrcFrameT *Frames)
```

Structures:

```
typedef struct MDITrcFrame_Struct {
    MDIUint32        Type;
    MDIResourceT     Resource;
    MDIOffsetT       Offset;
    MDIUint64        Value;
} MDITrcFrameT;
```

Type:

MDITTypePC	Resource and Offset give the address of a fetched or executed instruction.
MDITTypeInst	Value contains the instruction whose address is given by Resource and Offset.
MDITTypeRead	Resource and Offset give the address of a loaded data value.
MDITTypeWrite	Resource and Offset give the address of a stored data value.
MDITTypeAccess	Resource and Offset give the address of a loaded or stored data value.
MDITTypeRData_8	Value contains the 8-bit data value read from the address given by Resource and Offset.
MDITTypeWData_8	Value contains the 8-bit data value written to the address given by Resource and Offset.

MDITTypeRData_16	Value contains the 16-bit data value read from the address given by Resource and Offset.
MDITTypeWData_16	Value contains the 16-bit data value written to the address given by Resource and Offset.
MDITTypeRData_32	Value contains the 32-bit data value read from the address given by Resource and Offset.
MDITTypeWData_32	Value contains the 32-bit data value written to the address given by Resource and Offset.
MDITTypeRData_64	Value contains the 64-bit data value read from the address given by Resource and Offset.
MDITTypeWData_64	Value contains the 64-bit data value written to the address given by Resource and Offset.

Returns:

MDISuccess	No Error, FrameCount frames have been returned.
MDIErrDevice	Invalid device handle.
MDIErrInvalidFrames	Requested frame range is invalid.
MDIErrTracing	Device is currently tracing.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function returns the requested range of "frames" of trace data. Although it may be called any number of times after MDITraceCount() has been called, until the next time MDITraceEnable() is called (if MDITraceDisable() had previously been called) or device execution is resumed (if tracing remained enabled). The debugger must call MDITraceCount() before this function can be called after new trace data has been captured. A "frame" of trace data describes a single instruction or data access performed by the target. *Type* specifies how to interpret the rest of the frame data.

Depending on the capabilities of the device, data accesses may not be captured by the tracing system at all, or the values loaded and stored by data accesses may not be available. If the data values are available, they will always be included with the trace data since they would not otherwise be available to the debugger. If the debugger requests instruction values, and the underlying tracing system does not capture them, then the MDILib is required to fetch the instructions from device memory so they can be included in the trace data, if the MDILib is capable of doing so. This is indicated by a capability flag MDICAP_TraceFetchI.

FirstFrame is the frame number of the oldest frame to be returned in this call. Frames are numbered from 1 to N, where N is the total number of frames returned by MDITraceCount() and frame 1 is the oldest frame. *FrameCount* is the number of frames to be returned in **Frames*.

For instruction frames, it may be more efficient for the debugger to read the instruction values from the executable file rather than have the MDILib fetch them over what may be a remote communications link. In that case, the debugger will set *IncludeInstructions* to 0. If *IncludeInstructions* is set 1, then the MDILib will include the instruction values in the trace frame data.

8.7 Read PDtrace Data

```
MDIInt32
MDIPDtraceRead ( MDIHandleT Device,
                 MDITraceFrameNumberT FrameNumber,
                 MDITraceFrameCountT Count,
                 MDIUint32 Instructions,
                 MDITraceFrameT *Data)
```

Structures:

```
typedef struct {
   MDIUint32 Word;     // address of beginning of trace frame in trace memory
   MDIUint32 Bit;      // bit number of beginning of trace frame within trace word.
} MDITraceFrameNumberT;

typedef struct MDITraceFrame_Struct {
   MDITraceFrameNumberT FrameNumber;
   MDIUint32 Type;
   MDIResourceT Resource;
   MDIOffsetT Offset;
   MDIUint64 Value;
} MDITraceFrameT;

#define MDIType_TYPE_MASK       0x00000fff
#define MDIType_MOD_MASK        0xfffff000

/* Expanded trace types obtained using MDIType_TYPE_MASK */

#define MDITTypeOverflow         64    // trace fifo overflowed, information lost
#define MDITTypeTriggerStart     65    // value=trigger cause
#define MDITTypeTriggerEnd       66    // value=trigger cause
#define MDITTypeTriggerAbout     67    // value=trigger cause
#define MDITTypeTriggerInfo      68    // value=trigger cause
#define MDITTypeNotraceCycles    69    // value=number of notrace cycles
#define MDITTypeBackstallCycles  70    // value=number of backstall cycles
#define MDITTypeIdleCyclces      71    // value=number of idle cycles
#define MDITTypeTcbMessage       72    //  addr=TCBcode, value=TCBinfo field
#define MDITTypeModeInit         73    // value = new mode from following table
#define MDITTypeModeChange       74    // value = new mode from following table
                                       // 12:11  ISAM   00 = MIPS32
                                       //               01 = MIPS64
                                       //               10 = MIPS16
                                       //               11 = reserved
                                       // 10:8   MODE   000 = kernel, EXL=0, ERL=0
                                       //               001 = kernel, EXL=1, ERL=0
                                       //               010 = kernel, ERL=1
                                       //               011 = debug mode
                                       //               100 = supervisor mode
                                       //               101 = user mode
                                       //               other = reserved
                                       // 7:0    ASID
#define MDITypeUTM                75   // addr=1(TU1)or 2(TU2) value=user value

/* Expanded trace types obtained using MDIType_MOD_MASK */
#define MDITType_MOD_IM    0x00001000   // instruction cache miss signal
#define MDITType_MOD_LSM   0x00002000   // data cache miss signal
#define MDITType_MOD_FCR   0x00004000   // function call/return instruction
#define MDITType_MOD_CPU   0x00F00000   // which CPU this message applies to
#define MDITType_MOD_TC    0xFF000000   // which TC this message applies to
```

```
/* Extended flags for MDISetBp() */
#define MDIBPT_HWFlg_TraceOnOnly    0x80000000
#define MDIBPT_HWFlg_TraceOffOnly   0x40000000

/* Values for Instructions parameter to MDITrcRead(): */

#define MDITraceReadNoInstructions      0
#define MDITraceReadInstructions        1
```

Returns:

MDISuccess	No Error, FrameCount frames have been returned.
MDIErrDevice	Invalid device handle.
MDIErrInvalidFrames	Requested frame range is invalid.
MDIErrTracing	Device is currently tracing.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function returns the requested range of trace frames from the hardware. Again, since a frame is not easily identified, a numbering scheme is used that rather than being an integer is a composite frame number. This composite consists of the trace word address combined with the bit number of the start of the message. For example, if trace word 12345 has the last part of a trace message that started in 12344, then a complete message, then part of a message that is continued in 12346, then there would be two trace frames 12345.16 and 12345.52.

When requesting trace data, the *FrameNumber* parameter would be this composite with 0 being the oldest trace word being collected and the number returned in *Count* (minus one) being the youngest. The return structure includes the frame number *Data->FrameNumber* since these are no longer sequential.

It is important to note that the caller must allocate **Count+1* for the size of *Data* since one extra frame can be returned under certain circumstances.

8.8 Get PDtrace Mode

```
MDIInt32
MDIGetPDtraceMode ( MDIHandleT Device,
                    MDITraceModeT *TraceMode)
```

Structures:

```
typedef struct {
   MDIUint32 Mode;        // trace mode (see definitions below)
   MDIUint32 Knob;        // other trace mode knobs (see definitions below)
   MDIUint32 Knob2        // some more trace mode knobs (see defines below)
} MDITraceModeT;
```

Mode is a bitmap composed of values:

```
#define PDtraceMODE_PC            0x00000001 // trace the PC
#define PDtraceMODE_LA            0x00000002 // trace the load address
#define PDtraceMODE_SA            0x00000004 // trace the store address
#define PDtraceMODE_LD            0x00000008 // trace the load data
#define PDtraceMODE_SD            0x00000010 // trace the store data
```

Knob is a bitmap composed of values:

```
#define PDtraceKNOB_Dbg           0x00000001 // trace in debug mode
#define PDtraceKNOB_Exc           0x00000002 // trace in exception and error modes
(EXL or ERL set)
#define PDtraceKNOB_Sup           0x00000004 // trace in supervisor mode
#define PDtraceKNOB_Ker           0x00000008 // trace in kernel mode
#define PDtraceKNOB_Usr           0x00000010 // trace in user mode
#define PDtraceKNOB_ASIDMask      0x00001F70 // if G=0, trace in this process only
#define PDtraceKNOB_ASIDShift        5
#define PDtraceKNOB_G             0x00002000 // trace in all processes
#define PDtraceKNOB_SyPMask       0x0001C000 // Synchronization period
#define PDtraceKNOB_SyPShift      14
#define PDtraceKNOB_TMMask        0x00060000 // On-chip trace 00=traceto,
01=tracefrom
#define PDtraceKNOB_TMShift       17
#define PDtraceKNOB_OfC           0x00080000 // Trace sent to off-chip memory
#define PDtraceKNOB_CA            0x00100000 // cycle-accurate (include idle cycle
records)
#define PDtraceKNOB_IO            0x00200000 // inhibit overflow (stall CPU to
prevent overflow)
#define PDtraceKNOB_AB            0x00400000 // Send PC info for all branches,
predictable or not
#define PDtraceKNOB_CRMask        0x03800000 // Trace clock ratio
#define PDtraceKNOB_CRShift       23
#define PDtraceKNOB_Cal           0x04000000 // 1=calibration mode (test pattern)
#define PDtraceKNOB_EN            0x08000000 // 1=Enable trace initially.  0=don't
generate trace until trace-on event.
#define PDtraceKNOB_debug         0x10000000 // 1=set trace hardware to debug (not
for customer use)
```

Knob2 is a bitmap composed of values:

```
#define PDtraceKNOB2_im        0x00000001;   // trace instr fetch cache miss bit
#define PDtraceKNOB2_lsm       0x00000002;   // trace load/store cache miss bit
#define PDtraceKNOB2_fcr       0x00000004;   // trace instr func. call/return bit
#define PDtraceKNOB2_tlsif     0x00000008;   // record im, lsm, and fcr in trace
#define PDtraceKNOB2_id        0x000000F0;   // processor id to record when trace
is shared among processors
#define PDtraceKNOB2_cpuG      0x00000100;   // enable trace for all CPU's
#define PDtraceKNOB2_cpufilter 0x0001FE00;   // If cpuG=0, trace only this CPU id
#define PDtraceKNOB2_tcG       0x00020000;   // enable trace for all TC's
#define PDtraceKNOB2_tcfilter  0x03FC0000;   // If tcG=0, trace only this TC id
#define PDtraceKNOB2_tracetc   0x04000000;   // record TC info in trace
```

Returns:

MDISuccess	No Error, tracing mode has been obtained.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support tracing.

MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function gets the current tracing mode that is set for the PDtrace functionality.

8.9 Set PDtrace Mode

```
MDIInt32
MDISetPDtraceMode ( MDIHandleT Device,
                    MDITraceModeT TraceMode)
```

Returns:

MDISuccess	No Error, tracing mode has been set.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function sets the current tracing mode to that in the *TraceMode* parameter.

8.10 Get TCB Trigger Information

```
MDIInt32
MDIGetTcbTrigger (  MDIHandleT Device,
                    MDIUint32 TriggerId,
                    MDITcbTriggerT *Trigger)
```

Structures:

```
typedef struct {
    MDIUint32 DebugMode;          // Fire at Debug Mode rising edge
    MDIUint32 ChipTrigIn;         // Fire at Chip Trigger In rising edge
    MDIUint32 ProbeTrigIn;        // Fire at Probe Trigger In rising edge
} MDITcbConditionT;

typedef struct {
    MDIUint32 ChipTrigOut;        // Generate Chip Trigger Out pulse
    MDIUint32 ProbeTrigOut;       // Generate Probe Trigger Out pulse
    MDIUint32 TraceMessage;       // Insert Message in Trace
    MDIUint8  TraceMessageInfo;   // 8-bit info for trace message
} MDITcbActionT;
```

```
typedef struct {
   MDITcbConditionT Condition;    // Conditions for firing trigger
   MDIUint32 Type;                // Type of trigger
   MDIUint32 FireOnce;            // Fire once only
   MDITcbActionT Action;          // Actions to be executed when trigger fires
} MDITcbTriggerT;

/* Action selections for hardware breakpoints */
typedef enum {
TRIGACTION_TRC,        // Single event trace
TRIGACTION_ARM,        // Set ARM condition
TRIGACTION_TON_IF_ARMED,
TRIGACTION_TOFF_IF_ARMED,
TRIGACTION_TRC_IF_ARMED,
TRIGACTION_DISARM      // Clear ARM condition
} MDITcbActionT;
```

Returns:

MDISuccess	No Error, trigger information has been obtained.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function gets the current trigger state that is set in the TCB.

8.11 Set TCB Trigger Information

```
MDIInt32
MDISetTcbTrigger (  MDIHandleT Device,
                    MDIUint32 TriggerId,
                    MDITcbTriggerT *Trigger)
```

Returns:

MDISuccess	No Error, trigger information has been set.
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support tracing.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service cannot be performed at this time because the target program is running.
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This function sets the current TCB trigger state to that in the parameter *Trigger.*

Chapter 9

Multi-Threaded and Multi-Processor Command Set

The functions in this command set augment other MDI functions described elsewhere in this document to provide support for multi-thread and multi-processor debugging.

Since not all devices will be capable of supporting multi-threading, support for these functions is optional in an MDILib. An MDILib will only set the `MDICap_TC` in *Config->MDICapability* if it supports the functions which relate to thread context control. Similarly an MDILib will set the `MDICap_Teams` flag in *Config->MDICapability* if it supports multi-processor teams.

9.1 Multi-Thread Control

9.1.1 Set Thread Context: Sets the current MDI thread context ID

```
MDIInt32
MDISetTC (MDIHandleT Device,
          MDITCIdT TCId)
```

Returns:

MDISuccess	No Error, current TC ID has been set
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support multiple TCs
MDIErrTCId	The specified TC ID is not a valid for this device
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This call sets the current MDI Thread Context (TC) ID to *TCId*, which must be a valid TC ID bound to this device, as returned by MDIGetTC() or MDITCQuery(). Note that TCs assigned to a device need not be contiguous. Upon entering debug mode due to a breakpoint or single-step exception, the MDILib shall automatically set the current MDI TC ID to that of the TC which caused the exception. When entering debug mode asynchronously, for example, because of a call MDIStop(), the current TC ID may be set to that of any TC within the device, including a halted or free TC if the device contains no runnable TCs, or has not yet been activated. However, the TC ID must reference a runnable TC if one is available.

The current MDI TC ID selects the thread context to be used when servicing other MDI functions, in particular those in Section 6.3, "Resource Access" on page 27. For hardware breakpoints it specifies the TC to match if the MDIBPT_HWFlg_TCMatch flag is used, see Section 6.5.1, "Set Full Breakpoint" on page 45 and Section 6.5.7, "Hardware Breakpoint Query: Retrieve a list of supported hardware breakpoint types" on page 49. Software breakpoints which are implemented by writing a breakpoint instruction at the breakpoint address are by definition global, and will be taken by any TC or device which executes the breakpoint instruction. Note that the current MDI TC may be changed spontaneously by a CPU reset since this may cause the previous TC to no longer be bound to the VPE.

9.1.2 Get Thread Context: Returns the current MDI thread context ID

```
MDIInt32
MDIGetTC (MDIHandleT Device,
          MDITCIdT *TCId)
```

Returns:

MDISuccess	No Error, current TC ID has been returned
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support multiple TCs
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This call returns the current MDI Thread Context ID.

9.1.3 Thread Context Query: Retrieves a list of active TCs

```
MDIInt32
MDITCQuery ( MDIHandleT Device,
             MDIInt32 *HowMany,
             MDITCDataT *TCData)
```

Returns:

MDISuccess	No Error
MDIErrDevice	Invalid device handle
MDIErrUnsupported	Device does not support multiple TCs
MDIErrParam	Invalid parameter, *HowMany should not be negative
MDIErrMore	More Thread Contexts exist in the processor than requested
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Structures:

```
typedef struct MDITCData_struct {
   MDITCIdT   TCId;
   MDIUInt32  Status;
} MDITCDataT;
```

Status:

MDITCStatusHalted	TC has been halted and is not capable of running
MDITCStatusFree	TC has not been allocated and is not capable of running
MDITCStatusRunning	TC is runnable and not blocked
MDITCStatusBlockedonWait	TC is runnable, but is blocked by a WAIT instruction waiting for an interrupt
MDITCStatusBlockedonYield	TC is runnable, but is blocked by a YIELD instruction waiting for an external event
MDITCStatusBlockedonGS	TC is runnable, but is blocked waiting for Gating Storage
MDITCStatusSuspended	TC is suspended by debugger (OR’ed with other values)

Description:

If the requested number of Thread Contexts (**HowMany*) is 0, the function returns no error (MDISuccess) and **HowMany* is set to the number of TCs in the processor. If **HowMany* is greater than zero on the call, then this positive value indicates the number of elements in the *TCData* array. The function will then fill in the array with information about the Thread Contexts in the current VPE, ensuring that the first *TCData* entry filled in is the current MDI TC. The **HowMany* return value is set to the number of TC status returned. If the *TCData* array is not large enough to hold all the TCs in the current device, then MDIErrMore is returned along with a filled array. If the debugger then calls MDIQueryTC again before any other MDI function is called, then the *TCData* for the next **HowMany* TCs is returned.

To only retrieve information about the current TC, **HowMany* should be set to 1, and *TCData* should point to a single MDITCDataT structure. The current TC may be a halted or free TC if the device contains no runnable TCs, or has not yet been activated.

A TC's state at the time of the last entry to debug mode can be obtained by AND'ing TCData[].Status with MDITCStatusMask. The value MDITCStatusSuspended may be OR'ed with Status to indicate that the TC was suspended by the debugger using MDISetRunMode(). The MDILib must only set MDITCStatusSuspended if the API version specified to MDIConnect() was 0x0002001E or higher.

9.2 Set Run Mode: Specify behavior when returning to the RUNNING state

```
MDIInt32
MDISetRunMode (  MDIHandleT Device,
                 MDITCIdT TCId,
                 MDIUint32 StepMode,
                 MDIUint32 SuspendMode)
```

Returns:

MDISuccess	No Error, mode has been set.
MDIErrDevice	Invalid device handle.
MDIErrTCId	The specified TC ID value is not a valid for this device
MDIErrTargetRunning	Trying to change execution mode of the thread when it is running
MDIErrUnsupported	Device does not support multiple TCs
MDIErrParam	Invalid values of SSCtl and SuspendCtl
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

*StepMode***:**

MDINoStep	Run normally - no single step
MDIStepInto	Step Into
MDIStepForward	Step Forward
MDIStepOver	Step Over

Description:

This call specifies how a thread context (TC) within a device, or the whole device, should behave after the next call to MDIExecute() or MDITeamExecute(). Each device, and each TC within a multi-threaded device, can be independently programmed to:

1. Remain suspended: The MDI library should "offline" the device or TC before leaving debug mode.
2. Single step: Execute one instruction from the device or TC and take a single-step exception once completed. If more than one TC is selected for single-step, then the first TC to complete an instruction will cause a debug exception and the other TCs may or may not have made any forward progress.
3. Run freely: no single-step or suspension.

When any TC causes a debug exception (breakpoint, single-step, etc.), then all TCs within that device are suspended and may be examined by the debugger until MDIExecute() or MDITeamExecute() is called again.

The *TCid* value specifies a particular TC within a multi-threaded device, or -1 to indicate all TCs within the device. If the device is not multi-threaded then a *TCid* value of -1 defines the execution behavior of the device. After being set by this call, each device or TC's execution mode is *sticky* until changed by another call to this function naming the same *TCid*, or a *TCid* of -1. Upon re-entering debug mode all single-step and suspend modes shall be reset (switched off).

To indicate that a TC or device should take a single-step exception, use a *SSMode* value other than MDINoStep - a value of MDINoStep means that a single-step exception shall not be enabled for the specified TC or device. For a description of the various values of *SSMode*, see Section 6.4.2, "Step: Single steps the device" on page 38.

To indicate that a TC or device should be suspended while the other TCs or devices are running, use a *SuspendMode* value of 1. Using a value of 0 implies that this TC or device will not be suspended, i.e. it will be considered by the processor's policy manager for normal or single-step execution upon leaving debug mode.

An MDILib may return an error of MDIErrParam if the debugger requests a set of single-step and suspend modes which are not compatible with each other (e.g. it may not be possible to support a combination of MDIStepInto and MDIStepForward on different TCs).

The examples below illustrate some commonly desired functionality:

- All TCs to run normally:
```
MDISetRunMode (TCid=-1, SSMode=MDINoStep, SuspendMode=0)
```
- Single-step all TCs in Step Forward mode:
```
MDISetRunMode (TCid=-1, SSMode=MDIStepForward, SuspendMode=0)
```
- Single-step TC 4 in Step Into mode, all other TCs to run freely:
```
MDISetRunMode (TCid=-1, SSMode=MDINoStep, SuspendMode=0)
MDISetRunMode (TCid=4, SSMode=MDIStepInto, SuspendMode=0)
```
- Single-step TC 2 and TC3 in Step Forward mode, while suspending all other TCs :
```
MDISetRunMode (TCid=-1, SSMode=MDINoStep, SuspendMode=1)
MDISetRunMode (TCid=2, SSMode=MDIStepForward, SuspendMode=0)
MDISetRunMode (TCid=3, SSMode=MDIStepForward, SuspendMode=0)
```

9.3 Multi-processor Team Control

The functions in this section can be used to affiliate a number of devices into a multi-processor debugging *team*, so that they stop and start execution together in a synchronized manner. The devices, or *team members*, may be single-threaded CPU cores within a multi-core system, VPEs within a multi-threaded CPU, or some combination of these.

A team is persistent, in that will not be deleted, or have members removed from it, just because a device is closed. The team will vanish only when MDIDestroyTeam() is called, or the last debugger disconnects from the MDILib or group of MDILibs which are maintaining the team.

For a more detailed discussion see Section 3.2, "Multi-processor Debugging" on page 10.

9.3.1 Create Team: Create a new multi-processor debugging team

```
MDIInt32
MDITeamCreate (  MDIHandleT MDIHandle,
                 MDITeamIdT *TeamId);
```

Returns:

MDISuccess	No Error, new empty team created
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams
MDIErrTooManyTeams	The MDILib cannot create another team
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

MDIHandle must be the value returned by a previous MDIConnect() call.

Creates a new empty team and returns its ID in **TeamId*. It is acceptable for an MDILib to limit the number of teams which it can support, including to zero or one, and return MDIErrTooManyTeams when this limit is exceeded.

9.3.2 Teams Query: Retrieves a list of active teams

```
MDIInt32
MDIQueryTeams (  MDIHandleT MDIHandle,
                 MDIInt32 *HowMany,
                 MDITeamIdT *TeamIds)
```

Returns:

MDISuccess	No Error
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams
MDIErrParam	Invalid parameter, *HowMany should not be negative
MDIErrMore	More teams defined than requested
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

MDIHandle must be the value returned by a previous MDIConnect() call.

If the requested number of teams (**HowMany*) is 0, the function returns no error (MDISuccess) and **HowMany* is set to the number of active teams. If **HowMany* is non-zero on entry, it specifies the number of elements in the *TeamID* array being passed in. The function fills in the *TeamIds* array with the IDs for up to **HowMany* teams and sets **HowMany* to the number filled in. If there is not enough room in the *TeamIds* array to hold all the available teams, MDIErrMore is returned. If the debugger then calls this function again before any other MDI functions are called, information is returned for the next **HowMany* teams.

9.3.3 Clear Team: Removes all members from a multi-processor team

```
MDIInt32
MDITeamClear (   MDIHandleT MDIHandle,
                 MDITeamIdT TeamId)
```

Returns:

MDISuccess	No Error, team deleted
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams
MDIErrTeamId	Invalid team ID
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

MDIHandle must be the value returned by a previous MDIConnect() call. *TeamId* specifies the id of the team to be cleared - that is all members are removed from the team. All team members currently in FROZEN state must be switched to the RUNNING state; team members in any other state remain unaffected. The team id and associated state remain active however, and new members may be added to the team.

9.3.4 Destroy Team: Destroys a multi-processor team

```
MDIInt32
MDITeamDestroy ( MDIHandleT MDIHandle,
                 MDITeamIdT TeamId)
```

Returns:

MDISuccess	No Error, team deleted
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams
MDIErrTeamId	Invalid team ID
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

MDIHandle must be the value returned by a previous MDIConnect() call. *TeamId* specifies the id of the team to be destroyed. All team members currently in FROZEN state must be switched to the RUNNING state; team members in any other state remain unaffected. The team ID and associated state can then be released and recycled by the MDILib.

9.3.5 Attach Team Member: Add a new member to a team

```
MDIInt32
MDITMAttach (    MDIHandleT MDIHandle,
                 MDITeamIdT TeamId,
                 MDITMDataT *TMData)
```

Returns:

MDISuccess	No Error
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams

MDIErrTeamId	Invalid team ID
MDIErrTGId	Invalid Target Group ID in *TMData
MDIErrDeviceId	Invalid Device ID in *TMData
MDIErrAlreadyMember	The device is already a team member
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Structures:

```
typedef struct MDITMData_struct {
    MDIHandleT   MDIHandle;
    MDITGIDT     TGId;
    MDIDeviceIDT DevId;
} MDITMDataT;
```

Description:

MDIHandle must be the value returned by a previous MDIConnect() call. *TeamId* must be a team id returned by a call to MDICreateTeam() or MDIQueryTeams().

This call adds a single device to an existing team. A device may be a member of only one team at a time, so if it is already a member of this or any other team, then MDIErrAlreadyMember shall be returned.

The ids *TMData->TGId* and *TMData->DevId* specify a device managed by the library whose handle is in *TMData->MDIHandl*e, a value returned by a previous call to MDIConnect(). An MDILib is permitted to return MDIErrMDIHandle if *TMData->MDIHandle* is not the same as the *MDIHandle* argument, but may optionally permit the creation of teams which cross library and probe boundaries. It is not necessary for the new device to have already been opened by this debugger or any other.

Refer to Section 3.2.1, "Multi-processor Teams" on page 11 for a description of the various states associated with devices in a team. If the new device is currently in RUNNING state, and if any existing member of the team is currently HALTED, then the new device must be placed immediately in the FROZEN state. It is permissible to add a currently disabled device to a team, in which case if any existing team member is HALTED, then the new device must be placed in a "pending" FROZEN state, in anticipation of it being enabled. If the new device is currently HALTED, then any existing team members which are RUNNING or disabled must be immediately switched to FROZEN (or pending FROZEN) state. In all other cases the states of the new device and existing team members remain unchanged.

9.3.6 Detach Team Member: Remove a single member from a team

```
MDIInt32
MDITMDetach (    MDIHandleT MDIHandle,
                 MDITeamIdT TeamId,
                 MDITMDataT *TMData)
```

Returns:

MDISuccess	No Error, new empty team created
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams
MDIErrTeamId	Invalid team ID
MDIErrTGId	Invalid target group id in *TMData
MDIErrDeviceId	Invalid device id in *TMData

MDIErrNotMember	The device is not a member of this team
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

MDIHandle must be the value returned by a previous MDIConnect() call. *TeamId* must be a team ID returned by a call to MDICreateTeam() or MDIQueryTeams().

This call removes a single device from the specified team. If the device is not in the team then MDIErrNotAffiliated is returned.

The device is described by the **TMData* which includes the handle of the MDI library module which controls it (usually the same as the *MDIModule* argument), and its Target Group ID and Device ID within that library. It is not necessary for the device to already be opened by this debugger or any other debugger.

Refer to Section 3.2.2, "Disabled Multi-processor Devices" on page 14 for a description of the states associated with devices in a team. If the removed device is currently in HALTED state, and no other team members are in the HALTED state, then any other team member in the FROZEN state must be placed immediately in the RUNNING state if they are enabled. If the removed device is in the FROZEN state, then it should immediately be restarted and placed in the RUNNING state. In all other cases the states of the new device and existing team members remain unchanged.

9.3.7 Team Member Query: Retrieves a list of team members

```
MDIInt32
MDIQueryTM (      MDIHandleT MDIHandle,
                  MDITeamIdT TeamId,
                  MDIInt32 *HowMany,
                  MDITMDataT *TMData)
```

Returns:

MDISuccess	No Error
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams
MDIErrTeamId	Invalid team ID
MDIErrParam	Invalid parameter, *HowMany should not be negative
MDIErrMore	More team members exist than requested
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

MDIHandle must be the value returned by a previous MDIConnect() call. *TeamId* must be a team ID returned by a call to MDICreateTeam() or MDIQueryTeams().

If the requested number of team members (**HowMany*) is 0, the function returns no error (MDISuccess) and **HowMany* is set to the number of team members in *TeamId*. If **HowMany* is non-zero on entry, it specifies the number of elements in the *TMData* array being passed in. The function fills in the *TMData* array with the information for up to **HowMany* team members and sets **HowMany* to the number filled in. If there is not enough room in the *TMData* array to hold all the available members, MDIErrMore is returned. If the debugger then calls this function again before any other MDI functions are called, information is returned for the next **HowMany* team members.

9.3.8 Team Execute: Place all team members into RUNNING state

```
MDIInt32
MDITeamExecute ( MDIHandleT MDIHandle,
                 MDITeamIdT TeamId)
```

Returns:

MDISuccess	No Error
MDIErrMDIHandle	Invalid MDI Handle
MDIErrUnsupported	MDI library does not support teams
MDIErrTeamId	Invalid team ID
MDIErrWrongThread	Call was not made by the connected thread
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

MDIHandle must be the value returned by a previous MDIConnect() call. Places all team members "simultaneously" into the RUNNING state, irrespective of their current state. This call will normally be used only by a multi-processor aware debugger which is controlling all of the team members, for example an SMP operating system kernel debugger. The behavior of each TC and device after returning to RUNNING state is governed by any previous calls to the MDISetRunMode() function, see Section 9.2, "Set Run Mode: Specify behavior when returning to the RUNNING state" on page 73. In a team containing only VPE devices within a single core, all team members must resume execution in precise synchronization. In a team containing multiple cores there may be a small delay between each device restarting, though the MDILib should attempt to make such a delay as short as possible. Any team member entering debug mode while the others are awaiting restart should abort the restart operation and leave all other members in the HALTED or FROZEN state, as appropriate.

Chapter 10

Complex Break and Trigger and StopWatch Timer Command Set

The functions in this command set augment other MDI functions described elsewhere in this document to provide support for debugging breakpoint and trigger conditions that are complex. For example, breakpoints that can be primied or qualified by other breakpoints and conditions, as well as breakpoints that may need to trigger simultaneously (as a tuple for example with other breakpoints).

The commands in this chapter also provides support for StopWatch timers provided in hardware.

10.1 Set a Priming Conditon for the Specified Complex Breakpoint

```
MDIInt32
MDISetBpPrimingCondition ( MDIBpIdT BpId
                           MDIPrimingConditon Cond)
```

Returns:

MDISuccess	No Error, BpId has been set to the priming condition specified.
MDIErrBPId	Invalid Breakpoint ID
MDIErrParam	The index into the priming condition table is bad

Description:

Set the priming condition for the specified breakpoint.

If the breakpoint is set successfully, this will set the breakpoint specified by `BpId` with the priming condition specified by the table entry pointed to by the index specified in the input parameter cond.

10.2 Get the Priming Conditon for the Specified Complex Breakpoint

```
MDIInt32
MDIGetBpPrimingCondition ( MDIBpIdT BpId
                           MDIPrimingConditon *Cond)
```

Returns:

MDISuccess	No Error, priing condition for BpId has been returned successfully
MDIErrBPId	Invalid Breakpoint ID

Description:

Get the priming condition for the specified breakpoint.

10.3 Query Complex Breakpoint and StopWatch Configuration Options

```
MDIInt32
MDICbtConfigQuery (MDIHandleT Device,
          MDICbtConfigTypeT ConfigType,
          MDICbtBkptTypeT BkptType,
          MDIUint32 Index,
          MDIInt32 *HowMany,
          MDICbtConfigT *CbtConfig)
```

Returns:

MDISuccess	No Error
MDIErrDevice	Invalid device handle.
MDIErrResource	Index exceeds hardware resources
MDIErrParam	Invalid parameter, *HowMany should not be negative
MDIErrMore	More data exists than requested
MDIErrUnsupported	Device does not support CBT
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

The caller specifies the configuration type they want to query in ConfigType. as either primed, qualified, tuple, or stopwatch type. The BkptType specifies whether an instruction or data breakpoint configuration is wanted. For the stopwatch type, the stopwatch pair define is used. The Index specifies which instruction or data breakpoint is wanted. For the stopwatch type, this is the index of the pair (0 for 1st, 1 for 2nd, etc.).

If the requested number (*HowMany) is 0, the function returns no error (MDISuccess) and *HowMany is set to the number of possible configuration choices. If *HowMany is non-zero on entry, it specifies the number of elements in the CbtConfig array being passed in. If there is not enough room in the CbtConfig array to hold all available data, MDIErrMore is returned. If the debugger then calls the Query routine again before any other MDI functions are called, information is returned for the next *HowMany devices. CbtConfig is an array of *HowMany elements. It specifies the type (instruction, data, etc.) and the index (for instruction, data, or stopwatch pairs).

The error MDIErrResource is returned if the Index exceeds hardware capability. For example, inst5 is last valid instruction breakpoint in the implementation, so when calling this query function with inst6, this error will be returned

Coding example to retrieve configuration of primed conditions for inst0:

```
*howMany = 0;
if ((err = MDICbtConfigQuery(myHandle, MDICBTCONFIGTYPE_PRIMED,
MDICBTCONFIG_INSTRUCTION, 0, *howMany, NULL) {
      /* check for error here, then continue processing */
}
cbtConfig = malloc(*howMany * sizeof(MDICbtConfigT));
if ((err = MDICbtConfigQuery(myHandle, MDICBTCONFIGTYPE_PRIMED,
MDICBTCONFIG_INSTRUCTION, 0, *howMany, cbtConfig) {
     /* check for error here, then continue processing */
}
```

The pair configuration of a pair of breakpoints used as the start/stop triggers for the StopWatch timer is also obtained using this config query command.

10.4 Get the Current Value of the StopWatch Timer

```
MDIInt32
MDIGetStopWatchValue (MDIHandleT Device,
          MDIUint32 *Value)
```

Returns:

MDISuccess	No Error, current value of StopWatch Timer has been returned
MDIErrDevice	Invalid device handle.
MDIErrUnsupported	Device does not support StopWatch Timer
MDIErrDisabled	Service cannot be performed because the device is disabled
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

This call returns the current value of the StopWatch Timer in the processor.

10.5 Clear the Value of the StopWatch Timer

```
MDIInt32
MDIClearStopWatch (MDIHandleT Device)
```

Returns:

MDISuccess	No Error, Timer cleared
MDIErrMDIHandle	Invalid device handle
MDIErrUnsupported	Device does not support StopWatch Timer
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

Clears the value of the StopWatch Timer.

10.6 Set the Mode of the StopWatch Timer

```
MDIInt32
MDISetStopWatchMode (MDIHandleT Device,
                    MDIStopWatchModeT Mode,
                    MDIUint32 PairIndex,
                    MDIUint32 startInstIndex,
                    MDIUint32 stopInstIndex)
```

Returns:

MDISuccess	No Error, Timer mode set
MDIErrMDIHandle	Invalid device handle
MDIErrUnsupported	Device does not support StopWatch Timer
MDIInvalidParam	Device does not support the specified PairIndex or startInstIndex or stopInstIndex; only valid when Mode is MDIStopWatchPair

MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

Sets the mode of the StopWatch Timer. The currently defined modes are free-running or set to a particular start/stop breakpoint pair. If the latter, then the index of the pair is defined by the PairIndex value, and the breakpoint that starts the time is specified in startInstIndex, and the breakpoint that stops the timer is specified in stopInstIndex. A value of -1 for either of these parameters implies that the current breakpoint settings are to remain unchanged and will be used.

10.7 Get the Mode of the StopWatch Timer

```
MDIInt32
MDIGetStopWatchMode (MDIHandleT Device,
                     MDIStopWatchModeT *Mode,
                     MDIUint32 *PairIndex,
                     MDIUint32 *startInstIndex,
                     MDIUint32 *stopInstIndex)
```

Returns:

MDISuccess	No Error, Timer mode returned
MDIErrMDIHandle	Invalid device handle
MDIErrUnsupported	Device does not support StopWatch Timer
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback

Description:

Returns the mode of the StopWatch Timer. The currently defined modes are free-running or set to a particular start/stop breakpoint pair. If the latter, then the index of the pair is returned in the PairIndex value, the breakpoint that starts the timer is returned in startInstIndex, and the breakpoint that stops the timer is returned in stopInstIndex.

MDI.h Header File

The user should verify the compiler's syntax for a 64-bit signed and unsigned entity, using Microsoft's Visual C++ version 6.0's 64 bit specifiers. The following portion of the specification may be used as a C header file to implement the specification:

```
/* Start of header file for MDI (mdi.h) */
#ifndef MDI_Specification_Definitions
#define MDI_Specification_Definitions

/**
 * To build MDILib:
 * Define MDI_LIB before #include "mdi.h"
 * Include mdi.def in the link on Windows hosts.
 *
 * When building an MDI application (debugger):
 * In one source file only, define MDILOAD_DEFINE before
 * #include "mdi.h" to define pointer variables for the API
 * functions.
 */

typedef unsigned int MDIUint32;
typedef int MDIInt32;

#ifdef _MSC_VER
  typedef unsigned __int64 MDIUint64;
  typedef __int64 MDIInt64;

  #ifndef __stdcall
    #define __stdcall __stdcall
  #endif

#else

  #if __WORDSIZE == 64
    typedef unsigned long MDIUint64;
    typedef long MDIInt64;
  #else
    typedef unsigned long long MDIUint64;
    typedef long long MDIInt64;
  #endif

  #ifndef __stdcall
    #define __stdcall
  #endif

  #ifndef __declspec
    #define __declspec(e)
  #endif

#endif


typedef MDIUint32 MDIVersionT;
typedef struct MDIVersionRange_struct
{
```

```
    MDIVersionT oldest;
    MDIVersionT newest;
} MDIVersionRangeT;

/*
 * Define various revision fields
 */
#define MDIMajor           2
#define MDIMinor           32 /* 32 decimal */
#define MDIOldMajor        1
#define MDIOldMinor        0
#define MDICurrentRevision ((MDIMajor << 16 ) | MDIMinor)
#define MDIOldestRevision  ((MDIOldMajor << 16) | MDIOldMinor)

typedef MDIUint32   MDIHandleT;
#define MDINoHandle ((MDIHandleT)-1)

typedef MDIUint32 MDITGIdT;

typedef struct MDITGData_struct
{
    MDITGIdT TGId;       /* MDI ID to reference this Target Group */
    char     TGName[81]; /* Descriptive string identifying this TG */
} MDITGDataT;

typedef MDIUint32 MDIDeviceIdT;

typedef struct MDIDData_Struct
{
    MDIDeviceIdT Id;    /* MDI ID to reference this device */
    char DName[81];     /* Descriptive string identifying this device */
    char Family[15];    /* Device’s Family (CPU, DSP) */
    char FClass[15];    /* Device’s Class (MIPS, X86, PPC) */
    char FPart[15];     /* Generic Part Name */
    char FISA[15];      /* Instruction Set Architecture */
    char Vendor[15];    /* Vendor of Part */
    char VFamily[15];   /* Vendor Family name */
    char VPart[15];     /* Vendor Part Number */
    char VPartRev[15];  /* Vendor Part Revision Number */
    char VPartData[15]; /* Used for Part Specific Data */
    char Endian;        /* 0 Big Endian, 1 Little Endian */
} MDIDDataT;

/* Valid values for MDIDDataT.Family: */
#define MDIFamilyCPU "CPU"
#define MDIFamilyDSP "DSP"

/* Valid values for MDIDDataT.Endian: */
#define MDIEndianBig    0
#define MDIEndianLittle 1

/* MDI Resources */
typedef MDIUint32 MDIResourceT;

typedef MDIUint64 MDIOffsetT;

typedef struct MDIRange_struct
{
    MDIOffsetT Start;
    MDIOffsetT End;
```

```
} MDIRangeT;

typedef struct MDICRange_struct
{
    MDIOffsetT   Offset;
    MDIResourceT Resource;
    MDIInt32     Count;
} MDICRangeT;

typedef struct MDIConfig_struct
{
    /* Provided By: Other Comments */
    char User[80];             /* Host: ID of caller of MDI */

    char      Implementer[80]; /* MDI ID of who implemented MDI */
    MDIUint32 MDICapability;   /* MDI: Flags for optional capabilities */

    /* Host: CB fn for MDI output */
    MDIInt32 (__stdcall *MDICBOutput) (MDIHandleT Device,
                                       MDIInt32 Type,
                                       char *Buffer,
                                       MDIInt32 Count );

    /* Host: CB fn for MDI input */
    MDIInt32 (__stdcall *MDICBInput) (MDIHandleT Device,
                                      MDIInt32 Type,
                                      MDIInt32 Mode,
                                      char **Buffer,
                                      MDIInt32 *Count);

    /* Host: CB fn for expression eval */
    MDIInt32 (__stdcall *MDICBEvaluate) (MDIHandleT Device,
                                         char *Buffer,
                                         MDIInt32 *ResultType,
                                         MDIResourceT *Resource,
                                         MDIOffsetT *Offset,
                                         MDIInt32 *Size,
                                         void **Value);

    /* Host: CB fn for sym/src lookup */
    MDIInt32 (__stdcall *MDICBLookup) (MDIHandleT Device,
                                       MDIInt32 Type,
                                       MDIResourceT Resource,
                                       MDIOffsetT Offset,
                                       char **Buffer );

    /* Host: CB fn for Event processing */
    MDIInt32 (__stdcall *MDICBPeriodic)(MDIHandleT Device);

    /* Host: CB fn for Synchronizing */
    MDIInt32 (__stdcall *MDICBSync)(MDIHandleT Device,
                                    MDIInt32 Type,
                                    MDIResourceT Resource);

} MDIConfigT;

/* MDIConfigT.MDICapability flag values, can be OR'ed together */
#define MDICAP_NoParser      1    /* No command parser */
#define MDICAP_NoDebugOutput 2    /* No Target I/O */
#define MDICAP_TraceOutput   4    /* Supports Trace Output */
```

```
#define MDICAP_TraceCtrl      8    /* Supports Trace Control */
#define MDICAP_TargetGroups  0x10 /* Supports Target Groups */
#define MDICAP_PDtrace       0x20 /* Supports PDtrace functions */
#define MDICAP_TraceFetchI   0x40 /* Supports Instruction Fetch during Trace */
#define MDICAP_TC            0x80 /* Supports Thread Contexts */
#define MDICAP_Teams         0x100 /* Supports Teams */

typedef struct MDIRunState_struct
{
    /**
     * MdiStatus values
     *
     *  Temporary states:
     *   NotRunning group: Halted, StepsDone, BPHit, UsrBPHit, Exception,
TraceFull,...
     *
     *     These states will show up on the 1st call to RunState().
     *      After the 1st call, the new state will be MDIStatusNotRunning.
     *
     *  States used for transitions:
     *    These state inform the RunLoop() that a specific MdiDevice is requesting
     *    a cpu state (DevState) change.
     *
     *     MdiStatusHalting:
     *       the mdi device wants to stop. The cpu could still be running or
already halted.
     *
     *     MdiStatusStartingRun:
     *       the mdi device would like to run as soon as possible. The cpu might
still be halted by other vdev.
     *
     */
    MDIUint32 Status;
    union u_info
    {
        void *ptr;
        MDIUint32 value;
    } Info;
} MDIRunStateT;

/* Status values:  Info interpretation: */
#define MDIStatusNotRunning  1   /* none */
#define MDIStatusRunning     2   /* none */
#define MDIStatusHalted      3   /* none */
#define MDIStatusStepsDone   4   /* none */
#define MDIStatusExited      5   /* Info.value = exit value */
#define MDIStatusBPHit       6   /* Info.value = BpID */
#define MDIStatusUsrBPHit    7   /* none */
#define MDIStatusException   8   /* Info.value = which exception */
#define MDIStatusTraceFull   9   /* none */
/* 10 is skipped to maintained backward compatibility */
#define MDIStatusDisabled    11  /* Device is not in execution mode */

#define MDIStatusMask        0xff /* Status values are in lowest byte */

/* These can be OR'ed in with MDIStatusRunning and MDIStatusNotRunning
 */
#define MDIStatusReset     0x100 /* currently held reset */
#define MDIStatusWasReset  0x200 /* reset asserted & released */
#define MDIStatusResetMask 0x300 /* reset state mask */
```

```
/* This can also be OR'ed in with MDIStatusHalted */
#define MDIStatusDescription 0x0400 /* Info.ptr = Descriptive string */

typedef struct MDICacheInfo_struct
{
    MDIInt32  Type;
    MDIUint32 LineSize;    /* Bytes of data in a cache line */
    MDIUint32 LinesPerSet; /* Number of lines in a set */
    MDIUint32 Sets;        /* Number of sets */
} MDICacheInfoT;

/* Values for MDICacheInfoT.Type (Cache types): */
#define MDICacheTypeNone        0
#define MDICacheTypeUnified     1
#define MDICacheTypeInstruction 2
#define MDICacheTypeData        3

typedef MDIUint32 MDIBpT;
#define MDIBPT_SWInstruction          1
#define MDIBPT_SWOneShot              2
#define MDIBPT_HWInstruction          3
#define MDIBPT_HWData                 4
#define MDIBPT_HWBus                  5
#define MDIBPT_HWInstructionPrimed    6
#define MDIBPT_HWInstructionQualified 7
#define MDIBPT_HWInstructionTuple     8
#define MDIBPT_HWDataPrimed           9
#define MDIBPT_HWDataQualified        10
#define MDIBPT_HWDataTuple            11

/* Hardware breakpoint types may have one or more of the following */
/* flag bits OR'ed in to specify additional qualifications. */
#define MDIBPT_HWFlg_AddrMask     0x10000
#define MDIBPT_HWFlg_AddrRange    0x20000
#define MDIBPT_HWFlg_DataValue    0x40000
#define MDIBPT_HWFlg_DataMask     0x80000
#define MDIBPT_HWFlg_DataRead     0x100000
#define MDIBPT_HWFlg_DataWrite    0x200000
#define MDIBPT_HWFlg_Trigger      0x400000
#define MDIBPT_HWFlg_TriggerOnly  0x800000
#define MDIBPT_HWFlg_TCMatch      0x1000000
#define MDIBPT_HWFlg_InvertMatch  0x2000000 // 'not' match
#define MDIBPT_HWFlg_TypeQualMask 0xffff0000

#define MDIBPT_TypeMax      MDIBPT_HWDataTuple
#define MDIBPT_TypeMask     0xff
/* Hardware breakpoint types 6 to 11 must specify the index of the instruction or
   data breakpoint in MDIBPT_IndexMask bits */
#define MDIBPT_IndexMask    0xff00

typedef MDIUint32 MDIBpIdT;

#define MDIAllBpID (~(MDIBpIdT)0)

typedef struct MDIBpData_struct
{
  MDIBpIdT Id;
  MDIBpT Type;
```

```
  MDIUint32    Enabled;    /* 0 if currently disabled, else 1 */
  MDIResourceT Resource;
  MDIRangeT    Range;      /* Range.End may be an end addr or mask */
  MDIUint64    Data;       /* valid only for data write breaks */
  MDIUint64    DataMask;   /* valid only for data write breaks */
  MDIUint32    PassCount;  /* Pass count reloaded when hit */
  MDIUint32    PassesToGo; /* Passes to go until next hit */
} MDIBpDataT;

#define MDIBPT_HWType_Exec       1
#define MDIBPT_HWType_Data       2
#define MDIBPT_HWType_Bus        4
#define MDIBPT_HWTYpe_AlignMask  0xf0
#define MDIBPT_HWType_AlignShift 4
#define MDIBPT_HWType_MaxSMask   0x3f00
#define MDIBPT_HWType_MaxSShift  8
#define MDIBPT_HWType_VirtAddr   0x4000
#define MDIBPT_HWType_ASID       0x8000

typedef struct MDIBpInfo_struct
{
    MDIInt32  Num;
    MDIUint32 Type;
} MDIBpInfoT;

/* MDI Trace data type */
typedef struct MDITrcFrame_Struct
{
  MDIUint32    Type;
  MDIResourceT Resource;
  MDIOffsetT   Offset;
  MDIUint64    Value;
} MDITrcFrameT;

#define MDITTypePC       1  /* Instruction address only */
#define MDITTypeInst     2  /* Instruction address and value */
#define MDITTypeRead     3  /* Data Load address only */
#define MDITTypeWrite    4  /* Data Store address only */
#define MDITTypeAccess   5  /* Data Access (Load/Store) address only */
#define MDITTypeRData_8  6  /* Data Load address and 8-bit value */
#define MDITTypeWData_8  7  /* Data Store address and 8-bit value */
#define MDITTypeRData_16 8  /* Data Load address and 16-bit value */
#define MDITTypeWData_16 9  /* Data Store address and 16-bit value */
#define MDITTypeRData_32 10 /* Data Load address and 32-bit value */
#define MDITTypeWData_32 11 /* Data Store address and 32-bit value */
#define MDITTypeRData_64 12 /* Data Load address and 64-bit value */
#define MDITTypeWData_64 13 /* Data Store address and 64-bit value */

/* Values for Flags parameter to MDITGOpen() and MDIOpen(): */
#define MDISharedAccess    0
#define MDIExclusiveAccess 1

/* Values for Flags parameter to MDITGClose() and MDIClose(): */
#define MDICurrentState 0
#define MDIResetState   1

/* Values for SyncType parameter to MDICBSync(): */
#define MDISyncBP    0
#define MDISyncState 1
#define MDISyncWrite 2
```

```
/* Values for Direction parameter to MDIMove(): */
#define MDIMoveForward  0
#define MDIMoveBackward 1

/* Values for Mode parameter to MDIFind(): */
#define MDIMatchForward     0
#define MDIMismatchForward  1
#define MDIMatchBackward    2
#define MDIMismatchBackward 3

/* Values for Mode parameter to MDIStep(): */
#define MDIStepInto    0
#define MDIStepForward 1
#define MDIStepOver    2
#define MDINoStep      ((MDIUint32)~0)

/* "Wait Forever" value for WaitTime parameter to MDIRunState(): */
#define MDIWaitForever -1

/* Values for Mode parameter to MDIReset(): */
#define MDIFullReset       0
#define MDIDeviceReset     1
#define MDICPUReset        2
#define MDIPeripheralReset 3

/* Values for Flags parameter to MDIReset(): */
#define MDINonIntrusive    1

/* Values for Flags parameter to MDICacheFlush(): */
#define MDICacheHit        0
#define MDICacheWriteBack  1
#define MDICacheInvalidate 2
#define MDICacheWBInval    (MDICacheWriteBack|MDICacheInvalidate)
/* 4 is skipped for backward compatibility */
#define MDICacheLock       5
#define MDICacheIndex      0x80

/* Values for Status parameter from MDITraceStatus(): */
#define MDITraceStatusNone    1
#define MDITraceStatusTracing 2
#define MDITraceStatusWaiting 3
#define MDITraceStatusFilling 4
#define MDITraceStatusStopped 5

/* Values for Type parameter to MDICBOutput() and MDICBInput(): */
#define MDIIOTypeMDIIn    1
#define MDIIOTypeMDIOut   2
#define MDIIOTypeMDIErr   3
#define MDIIOTypeTgtIn    4
#define MDIIOTypeTgtOut   5
#define MDIIOTypeTgtErr   6
#define MDIIOTypeMDNotify 7

/* Values for Mode parameter to MDICBInput(): */
#define MDIIOModeNormal 1
#define MDIIORawBlock   2
#define MDIIORawNoBlock 3

/* Values for Type parameter to MDICBEvaluate(): */
```

```
#define MDIEvalTypeResource 1
#define MDIEvalTypeChar     2
#define MDIEvalTypeInt      3
#define MDIEvalTypeUInt     4
#define MDIEvalTypeFloat    5
#define MDIEvalTypeNone     6

/* Values for Type parameter to MDICBLookup(): */
#define MDILookupNearest 1
#define MDILookupExact   2
#define MDILookupSource  3

/* MDI function return values: */
#define MDISuccess                    0  /* Success */
#define MDINotFound                   1  /* MDIFind() did not find a match */
#define MDIErrFailure                -1  /* Unable to perform operation */
#define MDIErrDevice                 -2  /* Invalid Device handle */
#define MDIErrSrcResource            -3  /* Invalid Resource type */
#define MDIErrDstResource            -4  /* Invalid Resource type */
#define MDIErrInvalidSrcOffset       -5  /* Offset is invalid for the specified
                                            resource */
#define MDIErrInvalidDstOffset       -6  /* Offset is invalid for the specified
                                            resource */
#define MDIErrSrcOffsetAlignment     -7  /* Offset is not correctly aligned for the
                                            specified ObjectSize*/
#define MDIErrDstOffsetAlignment     -8  /* Offset is not correctly aligned for the
                                            specified ObjectSize */
#define MDIErrSrcCount               -9  /* Count causes reference outside of
                                            the resource space */
#define MDIErrDstCount               -10 /* Count causes reference outside of
                                            the resource space */
#define MDIErrBPType                 -13 /* Invalid breakpoint type */
#define MDIErrRange                  -14 /* Specified range is outside of the
                                            scope for the resource */
#define MDIErrNoResource             -15 /* Hardware resources not available */
#define MDIErrBPId                   -16 /* Invalid Breakpoint ID */
#define MDIErrMore                   -17 /* More data is available than was
                                            requested */
#define MDIErrParam                  -18 /* A parameter is in error, see
                                            specific instructions */
#define MDIErrTGHandle               -19 /* Invalid Target Group Handle */
#define MDIErrMDIHandle              -20 /* Invalid MDI Environment Handle */
#define MDIErrVersion                -21 /* Version not supported */
#define MDIErrLoadLib                -22 /* MDIInit(): Error loading library */
#define MDIErrModule                 -23 /* MDIInit(): Unable to link required
                                            MDI functions from library */
#define MDIErrConfig                 -24 /* Required callback functions not
                                            present */
#define MDIErrDeviceId               -25 /* Invalid device ID */
#define MDIErrAbort                  -26 /* Command has been aborted */
#define MDIErrUnsupported            -27 /* Unsupported feature */
#define MDIErrLookupNone             -28 /* Address did not match a symbol or
                                            source line. */
#define MDIErrLookupError            -29 /* Invalid address for look up. */
#define MDIErrTracing                -30 /* Can't clear trace buffer while
                                            capturing is in progress */
#define MDIErrInvalidFunction        -31 /* Function pointer is invalid */
#define MDIErrAlreadyConnected       -32 /* MDI Connection has already been made
                                            for this thread */
#define MDIErrTGId                   -33 /* Invalid Target Group ID */
```

```
#define MDIErrDuplicateBP            -34 /* A similar breakpoint has already been
                                            defined for this device, or for global
                                            breakpoints on any device */
#define MDIErrInvalidFrames          -35 /* Range of requested trace frames is
                                            invalid */
#define MDIErrWrongThread            -36 /* Call was not made by the connected
                                            thread */
#define MDIErrTargetRunning          -37 /* Trying to change execution mode of the
                                            thread when it is running */
#define MDIErrRecursive              -38 /* Illegal recursive call from from
                                            MDICDPeriodic */
#define MDIErrSrcObjectSize          -39 /* Invalid ObjectSize for resource */
#define MDIErrDstObjectSize          -40 /* Invalid ObjectSize for resource */
#define MDIErrTCId                   -41 /* TC is not valid for device */
#define MDIErrTooManyTeams           -42 /* Too many teams for MDILib */
#define MDIErrTeamId                 -43 /* Invalid team ID */
#define MDIErrDisabled               -44 /* Device is disabled */
#define MDIErrAlreadyMember          -45 /* Device is already a team member */
#define MDIErrNotMember              -46 /* Device is not a team member */

typedef MDIInt32 MDITCIdT;

typedef struct {
    MDIUint32 TCId;
    MDIUint32 Status;
} MDITCDataT;

#define MDITCStatusHalted         0
#define MDITCStatusFree           1
#define MDITCStatusRunning        2
#define MDITCStatusBlockedOnWait  3
#define MDITCStatusBlockedOnYield 4
#define MDITCStatusBlockedOnGS    5

#define MDITCStatusMask 0xff
#define MDITCStatusSuspended 0x100

typedef MDIInt32 MDITeamIdT;

typedef struct {
    /* MDIHandle is no longer used but it is remained here
       for backword compatibility for FS2 */
    MDIHandleT   MDIHandle;
    MDIHandleT   TGHandle;
    MDIHandleT   DevHandle;
    MDIUint32    Flags;
} MDITMDataT;

/* Cond parameter to MDISetBpPrimingCondition(): */
typedef MDIUint32 MDIPrimingConditionT;

/* Values for ConfigType parameter to MDICbtConfigQuery(): */
typedef MDIUint32 MDICbtConfigTypeT;
#define MDICBT_ConfigType_Primed    0
#define MDICBT_ConfigType_Qualified 1
#define MDICBT_ConfigType_Tuple     2
#define MDICBT_ConfigType_StopWatch 3

/* Values for BkptType parameter to MDICbtConfigQuery(): */
typedef MDIUint32 MDICbtBPTypeT;
```

```
#define MDICBT_BPType_StopWatchPair 0
#define MDICBT_BPType_Instruction   1
#define MDICBT_BPType_Data          2

/* Type and Type2 parameters to MDICbtConfigQuery(): */
typedef MDIUint32 MDICbtConfigItemTypeT;

/* Index parameter to MDICbtConfigQuery(): */
typedef MDIUint32 MDICbtIndexT;

/* Values for CbtConfig parameter to MDICbtConfigQuery(): */
typedef struct MDICbtConfig_struct
{
    MDICbtConfigItemTypeT Type;//type of configuration item
    MDICbtIndexT Index;//index for item
    MDICbtConfigItemTypeT Type2;//used for stopwatch pairs only
    MDICbtIndexT Index2;// used for stopwatch pairs only
} MDICbtConfigT;

/* Values for MDICbtConfigT.Type and .Type2 (Config types): */
#define MDICBT_Config_Bypass      0  //used for primed
#define MDICBT_Config_Instruction 1
#define MDICBT_Config_Data        2

/* Value parameter to MDIGetStopWatchValue(): */
typedef MDIUint32 MDIStopWatchValueT;

/* Values for Mode parameter to MDISetStopWatchMode(): */
typedef MDIUint32 MDIStopWatchModeT;
#define MDICBT_StopWatch_FreeRun 0  // in free run mode
#define MDICBT_StopWatch_Pair    1  // pair defined in PairIndex

/* PairIndex parameter to MDISetStopWatchMode(): */
typedef MDIUint32 MDIPairIndexT;

/* StartInstIndex parameter to MDISetStopWatchMode(): */
typedef MDIUint32 MDIStartIndexT;

/* StopInstIndex parameter to MDISetStopWatchMode(): */
typedef MDIUint32 MDIStopIndexT;

/* Function Prototypes */
#ifdef __cplusplus
extern "C" {
#endif

#if defined( MDI_LIB )          /* MDILib, do extern function declarations */
  #define yf(str) extern int __declspec(dllexport) __stdcall str

#elif defined( MDILOAD_DEFINE ) /* debugger, do function pointer definitions */
  #define yf(str) int (__stdcall *str)

#else                           /* debugger, do extern function pointer
                                   declarations */
  #define yf(str) extern int (__stdcall *str)
#endif

    /* 0 */
    yf(MDIVersion)   (MDIVersionRangeT *);
    yf(MDIConnect)   (MDIVersionT, MDIHandleT*, MDIConfigT*);
```

```
    yf(MDIDisconnect)(MDIHandleT,  MDIUint32);
    yf(MDITGQuery)   (MDIHandleT,  MDIInt32*, MDITGDataT*);
    yf(MDITGOpen)    (MDIHandleT,  MDITGIdT,  MDIUint32, MDIHandleT *);

    /* 5 */
    yf(MDITGClose)  (MDIHandleT, MDIUint32);
    yf(MDITGExecute)(MDIHandleT);
    yf(MDITGStop)   (MDIHandleT);
    yf(MDIDQuery)   (MDIHandleT, MDIInt32*,    MDIDDataT *);
    yf(MDIOpen)     (MDIHandleT, MDIDeviceIdT, MDIUint32, MDIHandleT *);

    /* 10 */
    yf(MDIClose)    (MDIHandleT, MDIUint32);
    yf(MDIRead)     (MDIHandleT, MDIResourceT, MDIOffsetT, void*, MDIUint32,
MDIUint32);
    yf(MDIWrite)    (MDIHandleT, MDIResourceT, MDIOffsetT, void*, MDIUint32,
MDIUint32);
    yf(MDIReadList) (MDIHandleT, MDIUint32,    MDICRangeT*, MDIUint32, void*);
    yf(MDIWriteList)(MDIHandleT, MDIUint32,    MDICRangeT*, MDIUint32, void*);

    /* 15 */
    yf(MDIMove)   (MDIHandleT, MDIResourceT, MDIOffsetT,  MDIResourceT,
                   MDIOffsetT, MDIUint32,    MDIUint32,   MDIUint32);
    yf(MDIFill)   (MDIHandleT, MDIResourceT, MDIRangeT,   void*, MDIUint32,
MDIUint32);
    yf(MDIFind)   (MDIHandleT, MDIResourceT, MDIRangeT,   void*, void*,
                   MDIUint32,  MDIUint32,    MDIOffsetT*, MDIUint32);
    yf(MDIExecute)(MDIHandleT);
    yf(MDIStep)   (MDIHandleT, MDIUint32,    MDIUint32);

    /* 20 */
    yf(MDIStop)      (MDIHandleT);
    yf(MDIReset)     (MDIHandleT, MDIUint32);
    yf(MDICacheQuery)(MDIHandleT, MDICacheInfoT*);
    yf(MDICacheFlush)(MDIHandleT, MDIUint32, MDIUint32);
    yf(MDIRunState)  (MDIHandleT, MDIInt32,  MDIRunStateT *);

    /* 25 */
    yf(MDISetBp)    (MDIHandleT, MDIBpDataT*);
    yf(MDISetSWBp)  (MDIHandleT, MDIResourceT , MDIOffsetT , MDIBpIdT*);
    yf(MDIClearBp)  (MDIHandleT, MDIBpIdT);
    yf(MDIEnableBp) (MDIHandleT, MDIBpIdT);
    yf(MDIDisableBp)(MDIHandleT, MDIBpIdT);

    /* 30 */
    yf(MDIBpQuery)     (MDIHandleT, MDIInt32*, MDIBpDataT*);
    yf(MDIDoCommand)   (MDIHandleT, char*);
    yf(MDIAbort)       (MDIHandleT);
    yf(MDITraceEnable) (MDIHandleT);
    yf(MDITraceDisable)(MDIHandleT);

    /* 35 */
    yf(MDITraceClear) (MDIHandleT);
    yf(MDITraceStatus)(MDIHandleT, MDIUint32 *);
    yf(MDITraceCount) (MDIHandleT, MDIUint32 *);
    yf(MDITraceRead)  (MDIHandleT, MDIUint32, MDIUint32, MDIUint32, MDITrcFrameT
*);
    yf(MDISetTC)      (MDIHandleT, MDITCIdT);

    /* 40 */
```

```
    yf(MDIGetTC)        (MDIHandleT, MDITCIdT*);
    yf(MDITCQuery)      (MDIHandleT, MDIInt32*,  MDITCDataT*);
    yf(MDISetRunMode)(MDIHandleT, MDITCIdT,   MDIUint32,   MDIUint32);
    yf(MDITeamCreate)  (MDIHandleT, MDITeamIdT*);
    yf(MDIQueryTeams)  (MDIHandleT, MDIInt32*, MDITeamIdT*);

    /* 45 */
    yf(MDIHwBpQuery)(MDIHandleT, MDIInt32*, MDIBpInfoT*);
    yf(MDICacheOp)   (MDIHandleT, MDIResourceT, MDIInt32, MDIResourceT, MDIOffsetT,
MDIUint32 );
    yf(MDICacheSync) (MDIHandleT, MDIResourceT, MDIOffsetT, MDIUint32 );
    yf(MDICacheInfo) (MDIHandleT, MDIResourceT, MDICacheInfoT*);
    yf(MDITeamClear)   (MDIHandleT, MDITeamIdT);

    /* 50 */
    yf(MDITeamDestroy) (MDIHandleT, MDITeamIdT);
    yf(MDITMAttach) (MDIHandleT, MDITeamIdT, MDITMDataT*);
    yf(MDITMDetach) (MDIHandleT, MDITeamIdT, MDITMDataT*);
    yf(MDIQueryTM)  (MDIHandleT, MDITeamIdT, MDIInt32*, MDITMDataT*);
    yf(MDITeamExecute) (MDIHandleT, MDITeamIdT);

    /* 55 */
    yf(MDISetBpPrimingCondition) (MDIHandleT, MDIBpIdT, MDIPrimingConditionT);
    yf(MDIGetBpPrimingCondition) (MDIHandleT, MDIBpIdT, MDIPrimingConditionT*);
    yf(MDICbtConfigQuery) (MDIHandleT, MDICbtConfigTypeT, MDICbtBPTypeT,
MDICbtIndexT, MDIInt32*, MDICbtConfigT*);
    yf(MDIGetStopWatchValue) (MDIHandleT, MDIStopWatchValueT*);
    yf(MDIClearStopWatch) (MDIHandleT);
    yf(MDISetStopWatchMode) (MDIHandleT, MDIStopWatchModeT, MDIPairIndexT,
MDIStartIndexT, MDIStopIndexT);
    yf(MDIGetStopWatchMode) (MDIHandleT, MDIStopWatchModeT*, MDIPairIndexT*,
MDIStartIndexT*, MDIStopIndexT*);

#undef yf

#ifdef __cplusplus
}
#endif
```

Example Code to Setup an MDILib Connection

```
/*****************************************************************
This may serve as a starting point to connect to MDI.dll
The mdiinit.c is used to find and link the MDI.dll
*****************************************************************/

#if defined(_WIN32) || defined(__CYGWIN32__)
#include <windows.h>
#else
typedef void *HMODULE;
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MDI_ALLOCATE
#include <mdi.h>
#include <mdimips.h>
#include <mdiinit.h>

MDIHandleT MDIhandle;
MDIHandleT TGhandle;
MDIHandleT Devhandle;

MDIDDataT DeviceData;

MDIConfigT config;

#define ec(str) {str, #str}

struct errorcodes_struct {
   int errorcode;
   char *str;
   } errorcodes[] = {
   ec( MDIErrFailure              ),
   ec( MDIErrDevice               ),
   ec( MDIErrSrcResource          ),
   ec( MDIErrDstResource          ),
   ec( MDIErrInvalidSrcOffset     ),
   ec( MDIErrInvalidDstOffset     ),
   ec( MDIErrSrcOffsetAlignment   ),
   ec( MDIErrDstOffsetAlignment   ),
   ec( MDIErrSrcCount             ),
   ec( MDIErrDstCount             ),
   ec( MDIErrBPType               ),
   ec( MDIErrRange                ),
   ec( MDIErrNoResource           ),
   ec( MDIErrBPId                 ),
   ec( MDIErrMore                 ),
   ec( MDIErrParam                ),
   ec( MDIErrTGHandle             ),
   ec( MDIErrMDIHandle            ),
   ec( MDIErrVersion              ),
```

```
    ec( MDIErrLoadLib              ),
    ec( MDIErrModule               ),
    ec( MDIErrConfig               ),
    ec( MDIErrDeviceId             ),
    ec( MDIErrAbort                ),
    ec( MDIErrUnsupported          ),
    ec( MDIErrLookupNone           ),
    ec( MDIErrLookupError          ),
    ec( MDIErrTracing              ),
    ec( MDIErrInvalidFunction      ),
    ec( MDIErrAlreadyConnected     ),
    ec( MDIErrTGId                 ),
    ec( MDIErrDuplicateBP          ),
    ec( MDIErrInvalidFrames        ),
    ec( MDIErrWrongThread          ),
    ec( MDIErrTargetRunning        ),
    ec( MDIErrRecursive            ),
      { 0, "Undefined"             },
       };

/*****************************************************************
ChkMDIerr  If errno is != 0, Display the MDI error on the console
Returns 0 if errno is MDISuccess, otherwise -1.

*****************************************************************/

int ChkMDIerr(int errno)
{
    int i;

    if (errno)
       {
       for (i = 0;
            errorcodes[i].errorcode && errorcodes[i].errorcode != errno;
            i++)
           {
           }
       fprintf(stderr,
               "\nMDI Error (%d) %s\n", errno, errorcodes[i].str);
       return -1;
       }

    return 0;
}

/*****************************************************************
SelectDevice
Returns -1 if no devices are present otherwise, index of selected device in device
array.
*****************************************************************/

int
SelectDevice(MDIDDataT *base, int number)
{
   int i;
   char buffer[81];
   int value;

   if (!number)
      {
```

```
        return (-1);
        }
    if (number == 1)
        {
        return (0);
        }
    do
        {
        fprintf(stdout, "Select Device:\n");
        for (i = 0; i < number ; i++ )
            {
            fprintf(stdout, "   %02d) %s\n", i + 1, base[i].DName);
            }
        fprintf(stdout, "Enter Number (1-%d) >", number);
        fgets(buffer, sizeof (buffer), stdin);
        value = atoi(buffer);
        }
    while (value < 1 || value > number);

    return (value - 1);
}

/******************************************************************
SelectTarget
Returns -1 if no Target groups are present otherwise, index of selected target
group in target group array.
******************************************************************/

int
SelectTarget(MDITGDataT *base, int number)
{
    int i;
    char buffer[81];
    int value;

    if (!number)
        {
        return (-1);
        }
    if (number == 1)
        {
        return (0);
        }
    do
        {
        fprintf(stdout, "Select Target Group:\n");
        for (i = 0; i < number ; i++ )
            {
            fprintf(stdout, "   %02d) %s\n", i + 1, base[i].TGName);
            }
        fprintf(stdout, "Enter Number (1-%d) >", number);
        fgets(buffer, sizeof (buffer), stdin);
        value = atoi(buffer);
        }
    while (value < 1 || value > number);

    return (value - 1);
}

/******************************************************************
```

```
OpenDev
Creates an array of the available devices in the target group, but if more than 1,
it queries the user as to which device it wants to connect.
Returns
                 If successful on device open, then DevHandle is set and 0 is
returned
If error, then a number < 0 is returned to indicate the error
******************************************************************/
int
openDev(void)
{
   MDIDDataT temp;
   MDIDDataT *tempbase;
   int NumDevices;
   int retval;
   int SelectedDevice;


   NumDevices = 0;
   retval = MDIDQuery(TGhandle, &NumDevices, &temp);
   if (ChkMDIerr(retval))
      {
      return retval;
      }

   tempbase = (MDIDDataT *)malloc(NumDevices * sizeof (MDIDDataT));
   retval = MDIDQuery(TGhandle, &NumDevices, tempbase);
   if (ChkMDIerr(retval))
      {
      free (tempbase);
      return retval;
      }

   SelectedDevice = SelectDevice(tempbase, NumDevices);

   if (SelectedDevice < 0)
      {
      free (tempbase);
      return (-5000);
      }

   memmove(&DeviceData, &tempbase[SelectedDevice], sizeof (MDIDDataT));

   free (tempbase);

   retval = MDIOpen(TGhandle, DeviceData.Id, MDIExclusiveAccess, &Devhandle);

   ChkMDIerr(retval);

   return retval;
}

/******************************************************************
openTG
If the MDI DLL does not support target groups, then set the TGhandel to the
MDIhandle and return 0; otherwise, create an array of the available target groups
If more than 1, query the user as to which target group it wants to connect.
Returns
If successful on target group open, then TGhandle is set and 0is returned
If error, then a number < 0 is returned to indicate the error
```

```
*****************************************************************/
int
openTG(void)
{
   int retval;
   MDITGDataT temp;
   MDITGDataT *tempbase;
   int SelectedTarget;
   int NumTargets;

   /* If the MDI DLL we're connecting to, does not do target groups,
      then just use the MDIhandle for the TGhandle */

   if (!(config.MDICapability & MDICAP_TargetGroups))
      {
      TGhandle = MDIhandle;
      return 0;
      }

   NumTargets = 0;
   retval = MDITGQuery(MDIhandle, &NumTargets, &temp);
   if (ChkMDIerr(retval))
      {
      return retval;
      }

   tempbase = (MDITGDataT *)malloc(NumTargets * sizeof (MDITGDataT));
   if (!tempbase)
      {
      return -5000;
      }
   retval = MDITGQuery(MDIhandle, &NumTargets, tempbase);
   if (ChkMDIerr(retval))
      {
      free(tempbase);
      return retval;
      }
   if (NumTargets > 1)
      {
      SelectedTarget = SelectTarget(tempbase, NumTargets);
      }
   else
      {
      SelectedTarget = 0;
      }

   if (SelectedTarget < 0)
      {
      free(tempbase);
      return -5001;
      }

   retval = MDITGOpen(MDIhandle, tempbase[SelectedTarget].TGId,
                      MDIExclusiveAccess, &TGhandle);
   free(tempbase);

   ChkMDIerr(retval);

   return retval;
```

```
}
/*******************************************************************
MDIDbgOutput
Required MDI output routine.  Just send buffers along to stderr and stdout
Returns MDISuccess
*******************************************************************/
int __stdcall
MDIDbgOutput( MDIHandleT handle, MDIInt32 Type, char *Buffer, MDIInt32 Count )
{
   if (Type == MDIIOTypeMDIErr || Type == MDIIOTypeTgtErr)
      fwrite( Buffer, Count, 1, stderr );
   else
      fwrite( Buffer, Count, 1, stdout );
   return( MDISuccess );
}
/*******************************************************************
MDIDbgInput
Required MDI input routine.  Just get a line from the console and send it in.
Returns MDISuccess

*******************************************************************/
int  __stdcall
MDIDbgInput( MDIHandleT handle, MDIInt32 Type, MDIInt32 Mode,
       char **Buffer, MDIInt32 *Count )
{
   static charlinebuf[ 1024 ];

   *Buffer = fgets( linebuf, sizeof (linebuf), stdin );
   *Count = strlen( linebuf );
   return( MDISuccess );
}
/*******************************************************************
opendevice
Load MDI dll through MDIInit.
Connect to MDI dll through MDIConnect.
Open a Target Group
Open the device we want to drive.
Returns MDISuccess if succesful number < 0 if error
*******************************************************************/
int
opendevice(void)
{
   int retval;
   MDIVersionT version;
   HMODULE h;

   retval = MDIInit(0, &h);

   if (ChkMDIerr(retval))
      {
      return retval;
      }

   version = MDICurrentRevision;

   memset(&config, 0, sizeof (config));
   config.MDICBOutput = MDIDbgOutput;
   config.MDICBInput = MDIDbgInput;
```

```
   retval = MDIConnect(version, &MDIhandle, &config);
   if (ChkMDIerr(retval))
      {
      return retval;
      }

   if (openTG())
      {
      retval = MDIDisconnect(MDIhandle, 0);
      ChkMDIerr(retval);
      return -5000;
      }

   if (openDev())
      {
      retval = MDITGClose(TGhandle, 0);
      ChkMDIerr(retval);
      retval = MDIDisconnect(MDIhandle, 0);
      ChkMDIerr(retval);
      return -5001;
      }
   return 0;
}
/*******************************************************************
closedevice
Close down the resources that were used in opendevice
Returns MDISuccess if succesful number < 0 if error
*******************************************************************/
int
closedevice(void)
{
    int closeerror;
    int retval;

    retval = MDIClose(Devhandle, 0);
    closeerror = retval;
    ChkMDIerr(retval);
    retval = MDITGClose(TGhandle, 0);
    closeerror |= retval;
    ChkMDIerr(retval);
    retval = MDIDisconnect(MDIhandle, 0);
    closeerror |= retval;
    ChkMDIerr(retval);
    return closeerror;
}

int
main(int argc, char *argv[])
{

   if (opendevice())
      {
      return (-1);
      }

   /* Application Code */

   if (closedevice())
      {
      return (-1);
```

```
        }
    return (0);
}
```

Appendix C

An MDI Addendum for MIPS32® and MIPS64® Architectures

C.1 Abstract

The MIPS architecture-specific resource objects of the MIPS Debug Interface (MDI) are described in this appendix.

C.2 MIPS MDIDDataT Fields

Valid values for the *MDIDDataT.FFamily* and *MDIDDataT.FISA* fields returned by MDIDQuery() are architecture specific. For MIPS, *MDIDDataT.FFamily* must be set to MDIMIP_FClass ("MIPS"). Valid values for *MDIDDataT.FISA* are:

MDIMIP_FISA_M1	"MIPSI"
MDIMIP_FISA_M2	"MIPSII"
MDIMIP_FISA_M3	"MIPSIII"
MDIMIP_FISA_M4	"MIPSIV"
MDIMIP_FISA_M5	"MIPSV"
MDIMIP_FISA_M32	"MIPS32"
MDIMIP_FISA_M64	"MIPS64"

C.3 MIPS Exception Codes

When MDIRunState() returns a *RunState.Status* value of MDIStatusException, the meaning of *RunState.Info.value* is architecture-specific. For MIPS processors, the value returned are the contents of the ExcCode field of the CP0 Cause register.

C.4 MIPS16e Instructions

For MIPS processors, it is necessary for the MDILib to know if a software breakpoint is being set via the MDIBpSet and MDISWBpSet (functions are on a normal 32-bit instruction or a MIPS16e instruction). Also, it is necessary for the debugger to know whether an instruction trace frame returned by MDITraceRead() is a MIPS16e instruction or not. For both cases, MIPS16e instructions are signaled by setting the low order bit in the corresponding address offset to 1. `mdimips.h` defines the name MDIMIP_Flg_MIPS16 for this purpose.

C.5 MIPS Resources

The "Programming Mnemonic" is the macro name defined in the header file `mdimips.h`, made available with this MDI addendum. As a minimum, all MIPS MDILib implementations are required to support the following encodings: MDIMIPCPU, MDIMIPPC, MDIMIPHILO, MDIMIPCP0, MDIMIPPHYSICAL, and MDIMIPGVIRTUAL. MDIMIPGVIRTUAL support may be limited to the physically mapped segments. If the target processor includes floating point hardware, the MDILib implementation is also required to support MDIMIPCP1, MDIMIPCP1C, MDIMIPFP, MDIMIPFPR, and MDIMIPDFP (if double precision is available).

For register type resources, if the size of the object being written to that register is smaller than the width of the register, then the register is written into the low-order bits and sign-extended. If the size is smaller and the register is being read, then the low-order bits of the register supply the value. When the size of the object being read is larger than the register width, then the register value is sign-extended to the desired width. If the size is larger than the register being written, then the high-order bits are ignored.

It is strongly recommended that MIPS MDILib implementations support all encodings for resources that the target system actually provides. Table C-1 lists the specific resource encodings (address spaces) defined for the MIPS architecture:

Table C-1 : MIPS32/MIPS64 Resource Definition

MIPS Resource	MDI Mnemonic	Offset Definition
CPU General Registers	MDIMIPCPU	Offset is the register number, 0-31
PC Pseudo Register	MDIMIPPC	Offset will be 0. If it is a MIPS16e instruction, then bit 0 is set to value one.
HI/LO Registers	MDIMIPHILO	Offset is 0 for register HI, 1 for register LO, 2 for ACX, 3 for HI1, 4 for LO1, 5 for ACX1, 6 for HI2, 7 for LO2, 8 for ACX2, 9 for HI3, 10 for LO3, and 11 for ACX3.
Coprocessor General Registers	MDIMIPCP*x* *x* = 0, 1, 2, or 3	Each CP*x* general register set consists of up to 256 banks of 32 registers. Offset(bits 12:5) select the bank. Offset(bits 4:0) select the register. Progamatically, ((bank << 5) + register = Offset). CP*x* general registers are those accessed by the MTC*x*/MFC*x* instructions.
Coprocessor Control Registers	MDIMIPCP*x*C *x* = 0, 1, 2, or 3	Each CP*x* control register set consists of up to 256 banks of 32 registers. Offset(bits 12:5) select the bank. Offset(bits 4:0) select the register. Progamatically, ((bank << 5) + register = Offset). CP*x* control registers are those accessed by the CTC*x*/CFC*x* instructions.
CPU Single-precision FP Pseudo Registers	MDIMIPFP	Offset is 0 to *n*-1, where *n* is the number of single-precision registers available: 16 MIPS I, MIPS II. 32 MIPS III, MIPS IV, MIPS V, MIPS32, MIPS64. These are the single precision (32-bit) floating-point values implemented in floating-point general purpose registers (FGRs). Offsets 0-15 map to FGRs[0,2,4,…] in MIPS I and MIPS II processors, since the odd numbered FGRs can not hold a single-precision value.
CPU Double-precision FP Pseudo Registers	MDIMIPDFP	Offset is 0 to *n*-1, where *n* is the number of double-precision registers available: 16 MIPS I, MIPS II, MIPS32. 32 MIPS III, MIPS IV, MIPS V, MIPS64. These are the double precision (64-bit) floating-point values implemented in floating-point general purpose registers (FGRs). Offsets 0-15 map to FGRs[0,2,4,…] in MIPS I, MIPS II and MIPS32 processors, since it takes two 32-bit FGRs to hold each double precision value.

Table C-1 : MIPS32/MIPS64 Resource Definition

FP registers access via software model	MDIMIPFPR	If implemented, this resource provides a software model of the FP register without the debugger requiring a detailed knowledge of how the hardware is implemented. In essense, it provides the abstraction of the ValueFPR() and StoreFPR() pseudo-code defined in the second volume of the MIPS64 Architecture Programming Manual. The details for how this resource works is shown in Table C-2, Table C-3, and Table C-4.
192 bit Accumulator	MDIMIP192ACC	The 192-bit accumulator register is addressed as three 64-bit registers. Offset is 0 for the high 64 bits, 1 for the middle 64 bits, and 2 for the low-order 64 bits.
Primary Instruction and Unified Cache Tags	MDIMIPPICACHET MDIMIPPUCACHET	This space is organized as an array of cache tag entries each of TagSize / 4 registers, where TagSize is returned by `MDICacheQuery() in` MDICacheInfoT. Each cache tag entry consists of cache tag registers followed by cache parity registers. For processors that do not support cache parity bits, writes to the cache parity registers are ignored and reads return zero. Offset is 0 through *n*-1, where *n* is the total number of registers in all cache tag entries. For multi-set caches, all of the cache tag entries for set 0 are followed by all of the cache tag entries for set 1, etc.
Primary Data Cache Tags	MDIMIPPDCACHET	See Primary Instruction and Unified Cache Tags' offset definition above.
Secondary Instruction and Unified Cache Tags	MDIMIPSICACHET MDIMIPSUCACHET	See Primary Instruction and Unified Cache Tags' offset definition above.
Secondary Data Cache Tags	MDIMIPSDCACHET	See Primary Instruction and Unified Cache Tags' offset definition above.
Tertiary Instruction and Unified Cache Tags	MDIMIPTICACHET MDIMIPTUCACHET	See Primary Instruction and Unified Cache Tags' offset definition above.
Tertiary Data Cache Tags	MDIMIPTDCACHET	See Primary Instruction and Unified Cache Tags' offset definition above.
Primary Instruction and Unified Cache Data	MDIMIPPICACHE MDIMIPPUCACHE	Offset is the byte offset within the cache. For multi-set caches, set 0 comes first in the address space, immediately followed by set 1, etc.
Primary Data Cache Data	MDIMIPPDCACHE	See Primary Instruction and Unified Cache offset definition above.
Secondary Instruction and Unified Cache Data	MDIMIPSICACHE MDIMIPSUCACHE	See Primary Instruction and Unified Cache offset definition above.
Secondary Data Cache Data	MDIMIPSDCACHE	See Primary Instruction and Unified Cache offset definition above.
Tertiary Instruction and Unified Cache Data	MDIMIPTICACHE MDIMIPTUCACHE	See Primary Instruction and Unified Cache offset definition above.
Tertiary Data Cache Data	MDIMIPTDCACHE	See Primary Instruction and Unified Cache offset definition above.

Table C-1 : MIPS32/MIPS64 Resource Definition

Translate Lookaside Buffers	MDIMIPTLB	This space is organized as an array of TLB entries. Offset is 0 through *n*-1 where *n* is two or four times the number of TLB entries available in the MMU, depending on type: For MIPS1 style single entry MMUs, a TLB entry consists of two registers, EntryLo followed by EntryHi. For MIPS3 style double entry MMUs, a TLB entry consists of four registers, EntryLo0, EntryLo1, EntryHi, and PageMask.
Physical Memory	MDIMIPPHYSICAL	Offset is the physical byte address.
Global Virtual Memory	MDIMIPGVIRTUAL	Offset is the virtual byte address.
ASID Virtual Memory	MDIMIPVIRTUAL + *asid*	Offset is the byte address within the virtual address space specified by the given ASID value. The MDIMIPVIRTUAL equate is set to 0x1000. Specific ASID spaces can then be referenced as MDIMIPVIRTUAL + *asid*.
EJTAG Memory	MDIMIPEJTAG	For processors that implement the MIPS EJTAG specification, this resource refers to the memory-mapped EJTAG registers. Offset is the byte offset from the beginning of register bank, as specified in the EJTAG specification.
Release 2 Shadow Register Set	MDIMIPSRS	For processors that implement Release 2 of the MIPS32 or MIPS64 architecture and include shadow register sets. The architectural maximum limit for n is 16. The number of the shadow register set is specified by the offset field. The SRS bank number and register number are combined using (set*32)+regno.
DSPControl register (used by the MIPS DSP ASE)	MDIMIPDSP	For processors that implement the MIPS DSP ASE
ITC Memory	MDIMIPITC	For processors that implement the MIPS MT ASE, this defines the ITC memory. Offset is the byte offset from the start of the ITC region.
Release 2 Hardware registers	MDIMIPHWR	Registers accesssed using the RDHWR Release 2 instruction. This is a read-only resource.

Table C-2 : MDIMIPFPR Resource Details for MIPS III, IV, & MIPS64, or MIPS32 with 64-bit FP

Data Size	FP32 Registers Mode	Offset	Read	Write
4	n/a	Even & Odd	VALUE <- FPR[OFFSET]$_{31..0}$	FPR[OFFSET] <- VALUE$_{31..0}$
8	FR=1	Even & Odd	VALUE <- FPR[OFFSET]$_{63..0}$	FPR[OFFSET] <- VALUE$_{63..0}$
8	FR=0	Even	VALUE <- (FPR[OFFSET+1]$_{31..0}$ << 32) || FPR[OFFSET]$_{31..0}$	FPR[OFFSET] <- VALUE$_{31..0}$ FPR[OFFSET+1] <- VALUE$_{63..32}$
8	FR=0	Odd	MDIErrSrcOffsetAlignment	MDIErrDstOffsetAlignment

Table C-3 : MDIMIPFPR Resource Details for MIPS32 (32-bit FP)

Data Size	Offset	Read	Write
4	Even & Odd	VALUE <- FPR[OFFSET]$_{31..0}$	FPR[OFFSET] <- VALUE$_{31..0}$
8	Even	VALUE <- (FPR[OFFSET+1]$_{31..0}$ << 32) || FPR[OFFSET]$_{31..0}$	FPR[OFFSET] <- VALUE$_{31..0}$ FPR[OFFSET+1] <- VALUE$_{63..32}$
8	Odd	MDIErrSrcOffsetAlignment	MDIErrDstOffsetAlignment

Table C-4 : MDIMIPFPR Resource Details for MIPS I & II

Data Size	Offset	Read	Write
4	Even	VALUE <- FPR[OFFSET/2]$_{31..0}$	FPR[OFFSET/2]$_{31..0}$ <- VALUE$_{31..0}$
4	Odd	VALUE <- FPR[OFFSET/2]$_{63..32}$	FPR[OFFSET/2]$_{63..32}$ <- VALUE$_{31..0}$
8	Even	VALUE <- FPR[OFFSET]$_{63..0}$	FPR[OFFSET] <- VALUE$_{63..0}$
8	Odd	MDIErrSrcOffsetAlignment	MDIErrDstOffsetAlignement

C.6 MIPS-Specific Breakpoint Implementation

C.6.1 MDISetBP() and MDISetSWBp() Function Calls

With respect to the MDISetBP() function call, when initializing the Range parameter in the MDIBpDataT data structure, if the instruction is MIPS16e, then bit 0 of range.start should have a value of 1.

For the MDISetSWBp() function call, the offset must be odd if it is a MIPS16e instruction.

C.6.2 Implementation of MDISetSWBp()

MDIBPT_SWInstruction is implemented in the MIPS architecture using the BREAK or SDBBP instruction. The hardware breakpoints, for example MDIBPT_HWInstruction, is implemented using either the coprocessor 0 Watch registers, or the EJTAG hardware breakpoint registers.

C.7 MIPS Specific Header File

The following header file, `mdimips.h`, may be used as a C header file to implement the specification for MIPS architectures:

```
/* Start of header file for MIPS Specific MDI  (MDImips.h) */

#ifndef MDI_MIPS_Specification_Definitions
#define MDI_MIPS_Specification_Definitions

/* Valid values for MDIDDataT.FClass: */
#define MDIMIP_FClass       "MIPS"
/* Valid values for MDIDDataT.FISA: */
#define MDIMIP_FISA_M1      "MIPSI"
#define MDIMIP_FISA_M2      "MIPSII"
#define MDIMIP_FISA_M3      "MIPSIII"
#define MDIMIP_FISA_M4      "MIPSIV"
#define MDIMIP_FISA_M5      "MIPSV"
#define MDIMIP_FISA_M32     "MIPS32"
#define MDIMIP_FISA_M64     "MIPS64"

/* Valid values for Resource */
#define MDIMIPCPU           1
#define MDIMIPPC            2
#define MDIMIPHILO          3
#define MDIMIPTLB           4
#define MDIMIPPICACHET      5
#define MDIMIPPUCACHET      MDIMIPPICACHET
#define MDIMIPPDCACHET      6
#define MDIMIPSICACHET      7
#define MDIMIPSUCACHET      MDIMIPSICACHET
#define MDIMIPSDCACHET      8
#define MDIMIP192ACC        9
#define MDIMIPCP0           10
#define MDIMIPCP0C          11
#define MDIMIPCP1           12
#define MDIMIPCP1C          13
#define MDIMIPCP2           14
#define MDIMIPCP2C          15
#define MDIMIPCP3           16
#define MDIMIPCP3C          17
#define MDIMIPFP            18
#define MDIMIPDFP           19
#define MDIMIPPICACHE       20
#define MDIMIPPUCACHE       MDIMIPPICACHE
#define MDIMIPPDCACHE       21
#define MDIMIPSICACHE       22
#define MDIMIPSUCACHE       MDIMIPSICACHE
#define MDIMIPSDCACHE       23
#define MDIMIPPHYSICAL      24
#define MDIMIPGVIRTUAL      25
#define MDIMIPEJTAG         26
#define MDIMIPSRS           27
#define MDIMIPFPR           28
#define MDIMIPDSP           29
#define MDIMIPTICACHET      30
#define MDIMIPTUCACHET      MDIMIPTICACHET
// 31 skipped for backwards compatibility with Spec rev 2.11
#define MDIMIPTDCACHET      32
#define MDIMIPTICACHE      33
#define MDIMIPTUCACHE      MDIMIPTICACHE
// 34 skipped for backwards compatibility with spec rev 2.11
#define MDIMIPTDCACHE      35
#define MDIMIPITCVIRTUAL   36
#define MDIMIPITC          MDIMIPITCVIRTUAL // for backwards compatibility
```

```
#define MDIMIPHWR           37
#define MDIMIPCURPC         38

#define MDIMIPVIRTUAL       0x00001000  /* 0x10xx: 0x1000+ASID value */

/* FS2: enables host-based memory cache when combined with MDIMIPPHYSICAL,
MDIMIPGVIRTUAL, or MDIMIPVIRTUAL */
#define MDIMIPHOSTCACHE 0x80000000

/*
** For MDISetBp(),MDISetSWBp(), MDITraceRead(), and for the MDIMIPPC
** resource, setting the low order address bit to 1 means that
** the addressed instruction is a MIPS16e instruction.
*/
#define MDIMIP_Flg_MIPS16   1

#endif

/* End of header file for MIPS Specific MDI (MDImips.h) */
```

MDI_PDtrace.h Header File

```
/* Start of header file for PDtrace (mdi_PDtrace.h) */

#ifndef MDITRACE_Specification_Definitions
#define MDITRACE_Specification_Definitions

/*
  This is the trace extensions for the MDI specification.  Upon approval,
  this header file will be merged into mdi.h.
*/

/*
  From mdi.h:

    To build MDILib:
        Define MDI_LIB before #include "mdi.h"
        Include mdi.def in the link on Windows hosts.

    To build an MDI application (debugger):
        Compile mdiinit.c and include it in your link
        Make a call to
           int MDIInit(char *MDIdllpathandname, HMODULE *handle)
        to explicitly load the specified MDILib before making any other MDI calls.
*/

#include "mdi.h"      //need standard defines

/* Trace Resources */

typedef MDIUint32   MDITraceFrameCountT;

/* MDI Trace data type */

typedef struct {
  MDIUint32 Word;     // address of beginning of trace frame in trace memory
  MDIUint32 Bit;      // bit number of beginning of trace frame within trace word.
} MDITraceFrameNumberT;

typedef struct MDITraceFrame_Struct {
  MDITraceFrameNumberT FrameNumber;
  MDIUint32 Type;
  MDIResourceT Resource;
  MDIOffsetT Offset;
  MDIUint64 Value;
} MDITraceFrameT;


typedef struct {
  MDIUint32 Mode;        // trace mode (see definitions above)
  MDIUint32 Knob;        // other trace mode knobs (see definitions below)
  MDIUint32 Knob2;       // more trace mode knobs (see defines below)
} MDITraceModeT;

/* Values for Mode member of MDITraceMode: */
```

```
#define PDtraceMODE_PC          0x00000001 // trace the PC
#define PDtraceMODE_LA          0x00000002 // trace the load address
#define PDtraceMODE_SA          0x00000004 // trace the store address
#define PDtraceMODE_LD          0x00000008 // trace the load data
#define PDtraceMODE_SD          0x00000010 // trace the store data

/* Values for Knob member of MDITraceMode: */

#define PDtraceKNOB_Dbg         0x00000001 // trace in debug mode
#define PDtraceKNOB_Exc         0x00000002 // trace in exception and error modes
(EXL or ERL set)
#define PDtraceKNOB_Sup         0x00000004 // trace in supervisor mode
#define PDtraceKNOB_Ker         0x00000008 // trace in kernel mode
#define PDtraceKNOB_Usr         0x00000010 // trace in user mode
#define PDtraceKNOB_ASIDMask    0x00001F70 // if G=0, trace in this process only
#define PDtraceKNOB_ASIDShift   5
#define PDtraceKNOB_G           0x00002000 // trace in all processes
#define PDtraceKNOB_SyPMask     0x0001C000 // Synchronization period
#define PDtraceKNOB_SyPShift    14
#define PDtraceKNOB_TMMask      0x00060000 // On-chip trace 00=traceto, 01=tracefrom
#define PDtraceKNOB_TMShift     17
#define PDtraceKNOB_OfC         0x00080000 // Trace sent to off-chip memory
#define PDtraceKNOB_CA          0x00100000 // cycle-accurate (include idle cycle
records)
#define PDtraceKNOB_IO         0x00200000 // inhibit overflow (stall CPU to prevent
overflow)
#define PDtraceKNOB_AB          0x00400000 // Send PC info for all branches,
predictable or not
#define PDtraceKNOB_CRMask      0x03800000 // Trace clock ratio
#define PDtraceKNOB_CRShift     23
#define PDtraceKNOB_Cal         0x04000000 // 1=calibration mode (test pattern)
#define PDtraceKNOB_EN          0x08000000 // 1=Enable trace initially.  0=don't
generate trace until trace-on event.
#define PDtraceKNOB_debug      0x10000000 // 1=set trace hardware to debug (not for
customer use)

/* Values for Knob2 member of MDITraceMode: */
#define PDtraceKNOB2_im         0x00000001;   // trace instr fetch cache miss bit
#define PDtraceKNOB2_lsm        0x00000002;   // trace load/store cache miss bit
#define PDtraceKNOB2_fcr        0x00000004;   // trace instr func. call/return bit
#define PDtraceKNOB2_tlsif      0x00000008;   // record im, lsm, and fcr in trace
#define PDtraceKNOB2_id         0x000000F0;   // processor id to record when trace
is shared among processors
#define PDtraceKNOB2_cpuG       0x00000100;   // enable trace for all CPU's
#define PDtraceKNOB2_cpufilter 0x0001FE00;   // If cpuG=0, trace only this CPU id
#define PDtraceKNOB2_tcG        0x00020000;   // enable trace for all TC's
#define PDtraceKNOB2_tcfilter   0x03FC0000;   // If tcG=0, trace only this TC id
#define PDtraceKNOB2_tracetc    0x04000000;   // record TC info in trace

#define MDIType_TYPE_MASK       0x00000fff
#define MDIType_MOD_MASK        0xfffff000

/* Expanded trace types */

#define MDITTypeOverflow          64    // trace fifo overflowed, information lost
#define MDITTypeTriggerStart      65    // value=trigger cause
#define MDITTypeTriggerEnd        66    // value=trigger cause
#define MDITTypeTriggerAbout      67    // value=trigger cause
#define MDITTypeTriggerInfo       68    // value=trigger cause
```

```
#define MDITTypeNotraceCycles    69    // value=number of notrace cycles
#define MDITTypeBackstallCycles  70    // value=number of backstall cycles
#define MDITTypeIdleCyclces      71    // value=number of idle cycles
#define MDITTypeTcbMessage       72    // addr=TCBcode, value=TCBinfo field
#define MDITTypeModeInit         73    // value = new mode from following table
#define MDITTypeModeChange       74    // value = new mode from following table
                                       // 12:11  ISAM   00 = MIPS32
                                       //               01 = MIPS64
                                       //               10 = MIPS16
                                       //               11 = reserved
                                       // 10:8   MODE   000 = kernel, EXL=0, ERL=0
                                       //               001 = kernel, EXL=1, ERL=0
                                       //               010 = kernel, ERL=1
                                       //               011 = debug mode
                                       //               100 = supervisor mode
                                       //               101 = user mode
                                       //               other = reserved
                                       // 7:0    ASID
#define MDITypeUTM                 75  // addr=1(TU1) or 2(TU2) value=user value

/* Expanded trace types obtained using MDIType_MOD_MASK */
#define MDITType_MOD_IM    0x00001000   // instruction cache miss signal
#define MDITType_MOD_LSM   0x00002000   // data cache miss signal
#define MDITType_MOD_FCR   0x00004000   // function call/return instruction
#define MDITType_MOD_CPU   0x00F00000   // which CPU this message applies to
#define MDITType_MOD_TC    0xFF000000   // which TC this message applies to

/* Extended flags for MDISetBp() */
#define MDIBPT_HWFlg_TraceOnOnly    0x80000000
#define MDIBPT_HWFlg_TraceOffOnly   0x40000000

/* Values for Instructions parameter to MDITrcRead(): */

#define MDITraceReadNoInstructions      0
#define MDITraceReadInstructions        1

/* Function Prototypes */

#ifdef __cplusplus
extern "C" {
#endif

#if defined( MDI_LIB )
/* MDILib, do extern function declarations */
#define yf(str) extern int __stdcall str
#elif defined( MDILOAD_DEFINE )
/* mdiinit.c, do function pointer defintions */
#define yf(str) int (__stdcall *str)
#else
/* debugger, do extern function pointer declarations */
#define yf(str) extern int (__stdcall *str)
#endif

/* MDIPDtraceRead: caller must allocate '*Count+1' for 'Data' since one extra frame
is returned
   under certain circumstances. */
yf(MDIPDtraceRead)(MDIHandleT Device, MDITraceFrameNumberT FrameNumber,
MDITraceFrameCountT *Count, MDIUint32 Instructions, MDITraceFrameT *Data);

yf(MDIGetPDtraceMode)(MDIHandleT Device, MDITraceModeT *TraceMode);
```

```
yf(MDISetPDtraceMode)(MDIHandleT Device, MDITraceModeT TraceMode);

#undef yf

#ifdef __cplusplus
}
#endif

#endif

/* End of header file for MDITRACE (mditrace.h) */
```

mdi_tcb.h Header File

```
/* Start of header file for Win32 MDI  (mdi.h) */

#ifndef MDITCB_Specification_Definitions
#define MDITCB_Specification_Definitions

/*
  This is the FS2 specific TCB extensions.  These are not supported by
  MDI but are made available to implementers if useful.
*/

/*
  From mdi.h:

    To build MDILib:
        Define MDI_LIB before #include "mdi.h"
        Include mdi.def in the link on Windows hosts.

    To build an MDI application (debugger):
        Compile mdiinit.c and include it in your link
        Make a call to
           int MDIInit(char *MDIdllpathandname, HMODULE *handle)
        to explicitly load the specified MDILib before making any other MDI calls.
*/

#include "mdi.h"  //need standard defines

typedef unsigned int       MDIUint8;

/* Values for DebugMode member of MDITcbConditionT: */

#define MDIDebugModeRisingEdge       0
#define MDINoDebugModeRisingEdge     1

/* Values for ChipTrigIn member of MDITcbConditionT: */

#define MDIChipTrigInRisingEdge      0
#define MDINoChipTrigInRisingEdge    1

/* Values for ProbeTrigIn member of MDITcbConditionT: */

#define MDIProbeTrigInRisingEdge     0
#define MDINoProbeTrigInRisingEdge   1

/* Values for ChipTrigOut member of MDITcbActionT: */

#define MDIChipTrigOutPulse          0
#define MDINoChipTrigOutPulse        1

/* Values for ProbeTrigOut member of MDITcbActionT: */

#define MDIProbeTrigOutPulse         0
#define MDINoProbeTrigOutPulse       1
```

```
/* Values for TraceMessage member of MDITcbActionT: */

#define MDIInsertTraceMessage        0
#define MDIDontInsertTraceMessage    1

/* Values for Type member of MDITcbTriggerT: */
#define MDITcbTypeInfo               0         // Do nothing or Generate Trace
message only
#define MDITcbTypeStart              1         // Start Trace
#define MDITcbTypeStop               2         // Stop Trace
#define MDITcbTypeAbout              3         // Stop Trace delayed

/* Values for FireOnce member of MDITcbTriggerT: */
#define MDIFireOnce                  0
#define MDIDontFireOnce              1

typedef struct {
  MDIUint32 DebugMode;            // Fire at Debug Mode rising edge
  MDIUint32 ChipTrigIn;           // Fire at Chip Trigger In rising edge
  MDIUint32 ProbeTrigIn;          // Fire at Probe Trigger In rising edge
} MDITcbConditionT;

typedef struct {
  MDIUint32 ChipTrigOut;          // Generate Chip Trigger Out pulse
  MDIUint32 ProbeTrigOut;         // Generate Probe Trigger Out pulse
  MDIUint32 TraceMessage;         // Insert Message in Trace
  MDIUint8  TraceMessageInfo;     // 8-bit info for trace message
} MDITcbActionT;

typedef struct {
  MDITcbConditionT condition;     // Conditions for firing trigger
  MDIUint32 Type;                 // Type of trigger
  MDIUint32 FireOnce;             // Fire once only
  MDITcbActionT Action;           // Actions to be executed when trigger fires
} MDITcbTriggerT;

/* Action selections for hardware breakpoints */
typedef enum {
TRIGACTION_TRC,         // Single event trace
TRIGACTION_ARM,         // Set ARM condition
TRIGACTION_TON_IF_ARMED,
TRIGACTION_TOFF_IF_ARMED,
TRIGACTION_TRC_IF_ARMED,
TRIGACTION_DISARM       // Clear ARM condition
} MDITcbActionT;

#define MAX_TCBTRIG      8


#if defined( MDI_LIB )
/* MDILib, do extern function declarations */
#define yf(str) extern int __stdcall str
#elif defined( MDILOAD_DEFINE )
/* mdiinit.c, do function pointer defintions */
#define yf(str) int (__stdcall *str)
#else
/* debugger, do extern function pointer declarations */
#define yf(str) extern int (__stdcall *str)
#endif
```

```
yf(MDIGetTcbTrigger)(MDIHandleT Device, MDIUint32 TriggerId, MDITcbTriggerT
*Trigger);
yf(MDISetTcbTrigger)(MDIHandleT Device, MDIUint32 TriggerId, MDITcbTriggerT
*Trigger);


#endif

/* End of header file for MDITCB (mditcb.h) */
```

Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself.

Version	Date	Comments
1.00	15 October 2001	Initial release
2.00	15 July 2003	Revisions include: • Syntax, typos, grammar • Additions for PDtrace/TCB tracing methodology
2.10	30 December 2004	Additional cleanup, additions to support MT ASE, DSP ASE, and multi-core debug.
2.11	24 January 2005	Resolved some open issues and incorporated Ernie and Nigel's comments
2.12	19 July 2005	Additional cleanups
2.20	29 March 2006	Clean ups and updates mainly around the MP and MT chapters and concepts
2.30	4 June 2007	• Added MDITCStatusSuspended to MDITCQuery return • Added the Complex Break Trigger (CBT) chapter • Added InvertMatch flag, valid for hardware data breakpoints
2.31	20 July 2007	Typo fixes in 2.30, no functionality change
2.32	21 August 2007	Cleaned up mdi.h and mdimips.h--added complex break and trigger, rename Bkpt to BP for consistent naming, updated version number to 2.32, made consistent upper/lower case usage in #defines, added back a missing #define MDIMIPHOSTCACHE and added a new reset define for Eclipse Nav IDE group (specifically for FS2 use).
2.40	19 October 2007	Clean up of mdimips.h for backwards compatibility with rev 2.11
2.41	22 September 2009	• Removed unused MDIErrDeviceHandle, MDIErrDevicesOpen, and MDIErrInvalidData • Added invalid source and destination ObjectSize return values MDIErrSrcObjectSize and MDIErrDstObjectSize • Clarification on MDICacheSync() example: global or ASID virtual memory are both valid resources for MIPS32 Release 2 • Added MDINonIntrusive flag to MDIReset() • Clarified the Size parameter in MDICacheOp() • Added TagSize to MDICacheInfoT. Specified the MIPS cache tag entry structure