The Meta Debug Interface (MDI)

Revision 1.4
Thursday, July 28, 2011

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202- 3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com
Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other third parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the respective third-party owner. The use herein of a third- party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

Embedded Software and Hardware License Agreement: You can print a copy of the Embedded Software and Hardware License Agreement from: www.mentor.com/eshla

Abstract

The main goal of Meta Debug Interface (MDI) is to define a set of data structures and functions that abstract hardware 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.

History

The original MDI 1.0 interface was co-developed by Embedded Performance Inc. and LSI Logic Corporation. All of Embedded Performance Inc's rights to MDI were sold to Mentor Graphics Corporation. LSI Logic granted a use license to MIPS Technologies, Inc. MIPS renamed the MDI interface to Microprocessor Debug Interface and has since diverged from the MDI 1.0 standard.

Overview

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, lowest-common-denominator interface to the tracing capabilities provided by many target devices.

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. As of this writing, an MDI addendum is available for the MIPS RISC, ARM, and PowerPC architectures.

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.
Device	A specific target system that can be accessed and controlled via MDI. Typically, this is a target board containing a CPU or DSP, or a simulator. In a multi-processor system, each processor 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.

Operation Principles

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 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 an MDI connection session be re-entrant. Therefore the communications path between debugger and MDILib is defined to be single threaded (synchronous) for any one connection. That is, for a given debugger process the same thread must make all MDILib calls relating to a single connection instance. To further clarify this MDI allows each MDIOpen call to be made with a new thread, but all MDI calls made using that handle must also be made with the thread used to open the MDI session.

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` plus currently available architecture specific headers such as `mdimips.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.

MDI Environment Command Set

Version

Retrieve the supported MDI versions for this MDILib implementation.

MDIInt32 MDIVersion (versions)
MDIVersionRangeT * versions;

Returns:

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

Version is a pointer to a structure where the oldest and newest version numbers supported by this MDILib implementation are placed. 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 first release of this specification documents version 0x00010000. The equate MDICurrentRevision will always reflect the newest/highest/current revision number of this specification. For implementations that only support 1 revision of the specification, oldest == newest.

ConnectSetup

The specific configuration/connection data used by an MDI library is library specific. MDIConnectSetup allows for the override of the default data set used by MDIConnect for configuration data. This is an optional interface and a library can choose to either not export the API call or return MDIErrUnsupported.

MDIInt32 MDIConnectSetup (ConnectSetupType, ConnectSetup)

MDIUint32	ConnectSetupType;
char *	ConnectSetup

Returns:

MDISuccess	No Error, handle and configuration have been returned.
MDIErrFailure	An unspecified error occurred, connection was not successful.
MDIErrParam	Invalid parameter
MDIErrUnsupported	MDI libraries not supporting this function return this error value

The ConnectSetupType parameter defines what type of data is in the ConnectSetup parameter. Below is a list of ConnectSetupType defines:

MDICSTypeDefault	Default setup (as if MDIConnectSetup was not called). ConnectSetup is not used.
MDICSTypeFile	ConnectSetup is a path to a file containing the connection setup data
MDICSTypeString	ConnectSetup is a pointer to a string buffer containing the connection setup data.

Connect

Establish a connection to the MDILib using the passed version of MDI. Configure and retrieve the configuration of supported MDI features.

MDIInt32 MDIConnect (MDIVersion, MDIHandle, Config)

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.

MDIVersion 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 MDIVersion() 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.

On input, Config->User contains a character string identifying the debugger to the MDILib. Config->MDICBOutput and Config->MDICBInput are set to the addresses of the required debugger call-back functions for I/O. Config->MDICBEvaluate, Config->MDICBLookup, Config->MDICBPeriodic, and Config->MDICBSync contain the addresses of the corresponding optional debugger call-back functions. or NULL if they are not provided.

On output, the MDILib stores an MDILib-specific identification string in Config->Implementer. Zero or more of the following flag values specifying MDILib capabilities are OR'ed together in Config->MDICapability. The intent is to allow a GUI debugger to disable user interface elements that are not needed for the MDILib it is connecting to.

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

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 particular 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.

Disconnect

Disconnect from the MDILib. Any open Target Groups and Devices associated with this connection will be closed first.

MDIInt32 MDIDisconnect (MDIHandle, Flags)

MDIHandleT	MDIHandle;
MDIUint32	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.

Returns:

MDISuccess	No Error.
MDIErrMDIHandle	Invalid MDI Handle
MDIErrParam	Invalid flags value
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Discovery

Device Discovery was added with MDI 1.3

This service call queries network interfaces for debug devices that are usable by the MDI library instance. It is assumed that any returned debug device at least provides an MDI API interface. It does not provide for any other services that might be defined for a debug device.

Various items can be returned via MDIDDiscoveryListT that allow a debug device to identify itself.

```
typedef struct MDIDiscoveryList_Struct
{
   char      ModelName[32];
   char      SerialNumber[16];
   MDIUint32 IPAddr;
   char      UnitName[32];
   char      QualifiedName[256];
} MDIDiscoveryListT
```

ModelName, SerialNumber are self explainatory. IPAddr is the Internet Protocol address for the debug device. UnitName is a user assigned name for the debug device. Name assignment is consider to be specific to a debug device and outside the scope of the MDI interface. QualifiedName is the fully qualified internet hostname (if any) for the debug device as returned by the network interfaces DNS server.

MDIInt32 MDIDiscovery (*List, MaxCount, *StructSize, *ListCount, *Err, ErrSize)

MDIDiscoveryListT *	List
MDIUint32	MaxCount
MDIUint32 *	StructSize
MDIUint32 *	ListCount
char *	Err
MDIUint32	ErrSize

Returns:

MDISuccess	No Error, FrameCount frames have been returned.
MDIErrMore	Means MDISuccess, but there are more devices than fits in the supplied List object buffer.
MDIErrUnsupported	Debugger does not support this call.
MDIErrRecursive	Recursive call was made.
MDIErrFailure	An error was detected and the a message about it written to the Err parameter (if not NULL).

Comments:

List is a pointer to MaxCount sized array of MDIDiscoverListT objects. StructSize is a pointer to the size of the MDIDiscoveryListT used by the caller. On return, it is set to the size as returned which might be less if the MDI library uses an older version. *list_count is set to the number of debug devices returned in *List (0..MaxCount). Err should be character buffer to write error messages to. ErrSize is the size in characters of Err. Err can be passed in as NULL to ignore error details.

Unlike other calls that return MDIErrMore a second call to MDIErrMore doesn't return the next set of data. Instead, you should recall again, but use a bigger buffer.

CPUInfoQuery

The CPUInfoQuery service was added to the specification with version 1.4.

The service call returns an array of MDICPUInfoT information, names and corresponding number cores, supported by the MDI Library for the given CPU architecture. The array returned is of CPUInfoCount size. Note that this call does not qualify the returned CPU data with the capabilities of any specific probe supported by the library. It only returns the libraries capabilities. The following structure defines the format of the returned data.

```
typedef struct MDICPUInfo_Struct
{
    char        *Name;
    MDIUint32   Cores;
} MDICPUInfoT;
```

MDIInt32 MDICPUInfoQuery (*Arch, **CPUInfo, CPUInfoCount, CPUInfoTypeSize)

char *	Arch
MDICPUInfoT **	CPUInfo
MDIUint32 *	CPUInfoCount
MDIUint32 *	CPUInfoTypeSize

Returns:

MDISuccess	No Error. The pointer passed in CPUInfo is set to point at a CPUInfoCount array of CPUInfoT type data blocks..
MDIErrParam	Means a parameter is in error. An unknown string in Arch, or a NULL value passed in one of the pointers.
MDIErrFailure	Out of memory error or some other unspecific error
MDIErrRecursive	Recursive call was made.

Comments:

Arch is a string matching MDIMIP_FClash, MDIARM_FClose, or MDIPPC_FClass. These definitions are strings defined in architecture specific MDI headers files mdi<arch>.h.

CPUInfo is a pointer to an array of strings. The pointer should be NULL on entry, and on return set to array of CPUInfo Blocks of CPUInfoCount length.

*CPUInfoCount is filled in with the number of CPUInfo Blocks returned.

*CPUInfoTypeSize returns the size of the CPUInfoT structure used by this library. It may larger or smaller than the clients definition if additions to CPUInfoT have occurred over time. The passed in poinrter for this data may be NULL in which case the client assumes a size.

Example Usage:

```
CPUInfoT  *cpu_info = NULL;
MDIUint32 cpu_info_count
MDIUint32 result = MDICPUInfoQuery( MDIMIP_FClass, &cpu_info,
&cpu_info_count, NULL );
```

Target Group Command Set

The MDILib may support the optional capability to perform certain operations of a group of open devices. If so, it will set the MDICAP_TargetGroups flag in Config.MDICapability. If this flag is set, the debugger must use the functions in this group to get a list of target groups and open a selected group before it can query and open a specific device. If the flag is not set, the debugger is required to bypass these functions and directly call MDIDQuery(). For MDILib implementations that do not support group operations, all Target Group functions will return MDIErrUnsupported.

In the case of a debugger that does not support group operations connecting to an MDILib that does, the debugger should open the selected target group with exclusive access to avoid the possibility that devices opened by the debugger could be affected by group commands issued by another debugger.

Target Group Query

Retrieve the names of the defined target groups.

MDIInt32 MDITGQuery (MDIHandle, HowMany, TGData)

MDIHandleT	MDIHandle
MDIInt32 *	HowMany
MDITGDataT *	TGData;

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

Returns:

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

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 may not be interpreted by the debugger.

The string names the target group. It is displayable text so that the debugger may query the user as to which target group he wants to open. The string name may not be more than 80 characters excluding the null terminator.

The target group ID is used in the MDITGOpen() function to select the specific target group.

Target Group Open

Open a target group. The handle returned in *TGHandle is used to reference this target group. Flags is set to MDIExclusiveAccess if the debugger wants exclusive control over any open devices in this target group. Else Flags is set to MDISharedAccess to allow other debuggers to open devices in this target group.

MDIInt32 MDITGOpen (MDIHandle, TGId, Flags, TGHandle);

MDIHandleT	MDIHandle
MDITGIdT	TGId
MDIUint32	Flags MDISharedAccess - Shared Access MDIExclusiveAccess - Exclusive Access
MDIHandleT *	TGHandle

Returns:

MDISuccess	No Error, *TGHandle has be set to the target group handle.
MDIErrFailure	An unspecified error occurred, open was not successful.
MDIErrParam	Invalid parameter.
MDIErrTGId	Invalid TGId.
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.

MDIHandle must be the value returned by the previous MDIConnect() call. MDILib implementations are not required to support shared access to a Target Group. If shared access is not supported, an attempt to open a Target Group already opened by another debugger will return MDIErrNoResource even if both opens specified shared access.

Target Group Close

Close a previously opened target group. Any open devices in the group will be closed automatically.

MDIInt32 MDITGClose (TGHandle, Flags);

MDIHandleT	TGHandle;
MDIUint32	Flags MDICurrentState: Leave in current state MDIResetState: Reset all target devices

Returns:

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

Target Group Execute

Place in execution those target group devices who have been configured for target group control.

```
MDIInt32 MDITGExecute (TGHandle);
        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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Target Group Stop

Stop those target group devices who have been configured for target group control.

```
MDIInt32 MDITGStop (TGHandle);
        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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Device Command Set

The device command set is subdivided into the following sections:

Session Control – Commands used to select and identify the correct device to control, and support debugger event processing and multiple debugger synchronization.

Resource Addresses – Definition of device resources.

Resource Access – Commands to access device resources

Run Control – Commands to control the device.

Breakpoints – Commands to establish and maintain breakpoints within the device.

Additionally, a MAJIC specific API extensions for Option Control is documented in the MAJIC API Extensions chapter.

Session Control

Device Query

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

MDIInt32 MDIDQuery (Handle, HowMany, DData)

MDIHandleT	Handle
MDIInt32 *	HowMany
MDIDDataT *	DData;

```
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 then requested.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

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. DDate->Id is a unique device ID assigned by the MDILib, and used by the debugger to specify the desired device to MDIOpen().

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 (CPU, DSP). DData->FClass further isolates the device type (MIPS, PPC, X86). DData->FPart 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). 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.

Valid values for DData->Family are part of the generic MDI specification. The only values currently specified are MDIFamilyCPU ("CPU") and MDIFamilyDSP ("DSP"). Valid values for DData->FClass and DData->FISA are architecture-specific and are listed in the corresponding Addendum. Valid values for DData->FPart, DData->VFamily, DData->VPart, DData->VPartRev, and DData->VPartData are vendor specific. 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.

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.

Open

Open a device. 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 a second (or third) debugger. Debuggers are 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.

```
MDIInt32 MDIOpen (Handle, DeviceID, Flags, DeviceHandle)
        MDIHandleT          Handle
        MDIDeviceIdT        DeviceID;
        MDIUint32           Flags
                            MDISharedAccess - Shared Access
```

	MDIExclusiveAccess - Exclusive Access
MDIHandleT *	DeviceHandle;

Returns:

MDISuccess	No Error. Device handle is written to DeviceHandle.
MDIErrFailure	An unspecified error occurred, open was not successful.
MDIErrDeviceId	Invalid Device Id.
MDIErrParam	Invalid parameter.
MDIErrFailure	An unspecified error occurred.
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.

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.

Close

Close a device.

MDIInt32 MDIClose (DeviceHandle, Flags)

MDIHandleT	DeviceHandle;
MDIUint32	Flags MDICurrentState: Leave in current state MDIResetState: Reset target device

Returns:

MDISuccess	No Error.
MDIErrFailure	Unable to perform close.
MDIErrParam	Invalid Flags parameter.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Process Events

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().

MDIInt32 MDICBPeriodic (DeviceHandle)

MDIHandleT	DeviceHandle;

Returns:

MDISuccess	No Error.

MDIErrDevice	Invalid Device handle.

Synchronize State

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. The reported device state changes keep the application informed of resource, break point and run state changes that have occurred in the target.

MDIInt32 MDICBSync (Device, SyncType, SyncResource,)

MDIHandleT	Device
MDIInt32	SyncType
	MDISyncBP
	MDISyncState
	MDISyncWrite
MDIResourceT	SyncResource

Returns:

MDISuccess	No Error, Sync information has been noted.
MDIErrDevice	Invalid Device handle.

Comments:

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.

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.

The list for the MIPS architecture is provided in Appendix B.

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 zero-extended to the register size by write operations. For register type resources where ObjectSize is greater than the actual size of the registers being addressed, each register value is zero-extended to ObjectSize bytes by read operations and the high order bytes of each value are ignored by write operations.

Read

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

MDIInt32 MDIRead (Device, SrcResource, SrcOffset, Buffer, ObjectSize, Count)

MDIHandleT	Device
MDIResourceT	SrcResource
MDIOffsetT	SrcOffset
void *	Buffer
MDIUint32	ObjectSize
MDIUint32	Count

Returns:

MDISuccess	No Error, requested data has been returned.
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 resource.
MDIErrSrcOffsetAlignment	SrcOffset is not correctly aligned for the specified ObjectSize.
MDIErrSrcCount	Specified Count and SrcOffset reference space that is outside of the scope for the resource. No objects were returned.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Write

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

MDIInt32 MDIWrite (Device, DstResource, DstOffset, Source, ObjectSize, Count)

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.
MDIErrDevice	Invalid Device handle.
MDIErrDstResource	DstResource is an invalid or unsupported resource type.
MDIErrInvalidDstOffset	DstOffset is invalid for the specified resource.
MDIErrDstOffsetAlignment	DstOffset is not correctly aligned for the specified ObjectSize.
MDIErrDstCount	Specified Count and DstOffset reference space that is outside of the scope for the resource. No objects were written.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Read List

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.

MDIInt32 MDIReadList (Device, ObjectSize, SrcList, ListCount, Buffer)

MDIHandleT	Device
MDIUint32	ObjectSize
MDICRangeT *	SrcList
MDIUint32	ListCount
void *	Buffer

```
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.
MDIErrDevice	Invalid Device handle.
MDIErrSrcResource	Invalid or unsupported resource type in SrcList.
MDIErrInvalidSrcOffset	SrcList[].Offset is invalid for the specified resource.
MDIErrSrcOffsetAlignment	SrcList[].Offset is not correctly aligned for the specified ObjectSize.
MDIErrSrcCount	SrcList[].Count and SrcList[].Offset reference space that is outside of the scope for the resource.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

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.

Write List

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.

MDIInt32 MDIWriteList (Device, ObjectSize, DstList, ListCount, Buffer)

MDIHandleT	Device
MDIUint32	ObjectSize
MDICRangeT *	DstList
MDIUint32	ListCount
void *	Buffer

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

Returns:

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

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.

Move

Move data from one resource to another resource on the device. If Direction is set to MDIMoveForward (Start to End) the move will be done incrementally, starting from the beginning of the ranges. If Direction is set to MDIMoveBackward (End to Start) the move will be done decrementally, starting from the end of the ranges.

MDIInt32 MDIMove (Device, SrcResource, SrcOffset, DstResource, DstOffset, ObjectSize, Count, Direction);

MDIHandleT	Device
MDIResourceT	SrcResource
MDIOffsetT	SrcOffset
MDIResourceT	DstResource
MDIOffsetT	DstOffset
MDIUint32	ObjectSize
MDIUint32	Count

MDIUint32	Direction: MDIMoveForward - Start to End MDIMoveBackward - End to Start

Returns:

MDISuccess	No Error, requested data has been moved.
MDIErrFailure	Unable to perform move 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.
MDIErrSrcCount	Specified Count and SrcOffset reference space that is outside of the scope for the 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 of the scope for the DstResource.
MDIErrAbort	Command was aborted in response to an MDIAbort() call.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Fill

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

MDIInt32 MDIFill (Device, DstResource, DstRange, Buffer, ObjectSize, Count);

MDIHandleT	Device
MDIResourceT	DstResource
MDIRangeT	DstRange
void *	Buffer
MDIUint32	ObjectSize
MDIUint32	Count

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 DstResource.
MDIErrDstOffsetAlignment	DstOffset is not correctly aligned for the specified ObjectSize.

MDIErrAbort	Command was aborted in response to an MDIAbort() call.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Find

Find 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. The search can be forwards or backwards through the specified range. The MDILib is required to support this function only for memory-mapped resources, and up to a maximum Count of 256.

MDIInt32 MDIFind (Device, SrcResource, SrcRange, Buffer, MaskBuffer, ObjectSize, Count, FoundOffset, Mode)

MDIHandleT	Device
MDIResourceT	SrcResource
MDIRangeT	SrcRange
void *	Buffer - Array of Count values to compare.
void *	MaskBuffer - Array of Count mask values to apply before comparing, or NULL if no masking is desired.
MDIUint32	ObjectSize
MDIUint32	Count
MDIOffsetT *	FoundOffset - If a match is found, the starting offset of the match is returned in *FoundOffset
MDIUint32	Mode Search mode, one of the following values: MDIMatchForward - Match specified Pattern, searching forward from the start address. MDIMismatchForward - Match anything that is not the specified Pattern, searching forward from the start address. MDIMatchBackward - Match specified Pattern, searching backward from the end address. MDIMismatchBackward - Match 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.
MDIErrAbort	Command was aborted in response to an MDIAbort() call.
MDIErrWrongThread	Call was not made by the connected thread.

MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Query Cache

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

MDIInt32 MDICacheQuery (Device, CacheInfo);

MDIHandleT	Device
MDICacheInfoT	CacheInfo[2]

```
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;
```

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

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.

Cache Flush

Write back and/or invalidate the device's cache. Type is set to MDICacheTypeUnified, MDICacheTypeInstruction, or MDICacheTypeData to specify which cache to operate on. Flags indicates the operation to perform and is set to one or more of MDICacheWriteBack and MDICacheInvalidate, OR'ed together, If Flags specifies both a write back and invalidate, the write back will happen before the invalidate.

MDIInt32 MDICacheFlush (Device, Type, Flags);

MDIHandleT	Device
MDIUint32	Type
MDIUint32	Flags 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 can not be performed.
MDIErrDevice	Invalid Device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Run Control

The debugger requests device execution by calling MDIExecute() , MDIStep(), or MDIStepRange. It then must periodically call MDIRunState() to monitor the target status until execution halts. 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, as it may not be true of some types of target systems such as simulators. 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.

Execute

Place the device into its RUNNING state. 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.

MDIInt32 MDIExecute (Device);

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Step

Single step the device. 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 assures 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. 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.

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.

MDIInt32 MDIStep (Device, Steps, Mode);

MDIHandleT	Device
MDIUint32	Steps
MDIUint32	Mode 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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

StepRange

This function is not required to be supported by an MDI Library instance. If implemented it single steps the device while the PC is within a a given set of instruction ranges. The stepping modes are the same as MDIStep. 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 assures 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. 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.

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.

Ranges is a pointer to a *RangeCount sized array of type MDIInstructionRange. Each range provided is used while stepping as specified by the step Mode. If RangeCount is larger than the size supported by a given MDILibrary, then *RangeCount immediately set to the maximum allowed number of ranges and MDIErrRange is returned. A library specific include file may define a maximum RangeCount in the form of MDI<lib designator>MaxRangeCount.

MDIInt32 MDIStepRange (Device, Steps, Mode, Ranges);

MDIHandleT	Device
MDIUint32	Mode MDIStepInto - Step Into MDIStepForward - Step Forward MDIStepOver - Step Over
MDIInstructionRangeT	*Ranges
MDIUint32	*RangeCount

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.
MDIErrRange	RangeCount is greater than allowed. Maximum supported returned in *RangeCount.
MDIErrUnsupported	MDI libraries not supporting this function return this error value. Additionally, MDIStepOutsideRange is also optionally supported using this same error value to indicate not supported.

Stop

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

MDIInt32 MDIStop (Device);

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.

Abort

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

MDIInt32 MDIAbort (Device);

MDIHandleT	Device

Returns:

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

Reset

Perform a target reset operation.

MDIInt32 MDIReset (Device, Mode);

MDIHandleT	Device	
MDIUint32	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.

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

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.
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. 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.

State

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. 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 number of milliseconds. If the device status changes before the time period expires, MDIRunState() will return the new status immediately.

MDIInt32 MDIRunState (Device, WaitTime, RunState);

MDIHandleT	Device
MDIInt32	WaitTime
MDIRunStateT *	RunState

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

Returns:

MDISuccess	No Error, RunState has been loaded with the device's current state as follows:

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
MDIStatusCommDrop	Not used

MDIErrDevice	Invalid Device handle.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

If the device status has not changed since the last call to MDIRunState(), RunState->Status will be set to MDIStatusNotRunning or MDIStatusRunning. 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 MDIExecute() 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.

MDICommDrop indicates the communication has been lost with the device handling this MDI Connection. Example: A probe losing power or being reset . Clients should proceed to close the connection when this status is returned.

There are also some 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
MDIStatusWasReset	reset was asserted & released
MDIStatusDescription	RunState->Info.ptr points to a descriptive string.
MDIStatusInstructionPending	indicates that a core is in the middle of executing an instruction (the implication is that it was suspended by some sort of multicore synchronization mechanism)

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.

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 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;
```

`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():

`MDIBPT_SWInstruction`	Standard "Software" Instruction execution breakpoint. Execution stops when control reaches the instruction at the address specified by `Resource` and `Range.Start` `PassCount` times. This breakpoint type is usually implemented by inserting a special instruction in memory.
`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.

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.
`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.

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.
`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 address is executed, but with it set to ten the break will be taken every tenth time the address is executed. If `PassCount` is set to zero on entry, MDISetBp() will treat it as a pass count 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.

Set Full Breakpoint

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().

MDIInt32 MDISetBp (Device, BpData)

MDIHandleT	Device
MDIBpDataT *	BpData

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.
MDIErrDstResource	Invalid resource type.
MDIErrUnsupported	The device doesn't support the type of breakpoint requested.
MDIErrRange	Specified range is outside of the scope for the resource.
MDIErrNoResource	The resources needed to implement the request are not available.
MDIErrDuplicateBP	A similar breakpoint was already been defined.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Set Software Breakpoint

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.

MDIInt32 MDISetSWBp (Device, Resource, Offset, BpId)

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 of the scope for the resource
MDIErrDuplicateBP	A similar breakpoint was already been defined.
MDIErrNoResource	The resources needed to implement the request are not available.
MDIErrWrongThread	Call was not made by the connected thread.

MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Clear Breakpoint

Clear a specified breakpoint, or clear all breakpoints.

MDIInt32 MDIClearBp (Device, BpId)

MDIHandleT	Device
MDIBpIdT	BpId MDIAllBpID clears all breakpoints.

Returns:

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

Enable Breakpoint

Enable a breakpoint. Enabling a previously disabled breakpoint does not affect its PassesToGo value.

MDIInt32 MDIEnableBp (Device, BpId)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Disable Breakpoint

Disable a breakpoint. 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.

MDIInt32 MDIDisableBp (Device, BpId)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Query Breakpoints

Query the set of defined breakpoints.

MDIInt32 MDIBpQuery (Device, HowMany, BpData)

MDIHandleT	Device
MDIInt32 *	HowMany
MDIBpDataT	BpData[]

Returns:

MDISuccess	No Error, see comments.
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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

If the requested number of breakpoints (*HowMany) is 0, the function returns no error (MDISuccess) and *HowMany is set to the number of defined breakpoints.

If the requested number of breakpoints is non-zero, it specifies the number of elements in the BpData array being passed in. If BpData[0].Id is not set to MDIAllBpID (-1), the debugger is querying specific breakpoints, and each BpData[i].Id specifies the desired breakpoint. All elements of BpData[] are filled in, and if any of them contain an invalid Id it is set to MDIAllBpID and MDIErrBPId is returned.

If BpData[0].Id is set to MDIAllBpID, the debugger is asking for all currently defined breakpoints to be returned. The function fills in the BpData array with information for up to *HowMany breakpoints and sets *HowMany to the number filled in. If there is not enough room in the BpData array to hold all the defined breakpoints, MDIErrMore is returned. If the debugger then calls MDIBpQuery() again with BpData[0].Id == MDIAllBpID, before any other MDI functions are called, information is returned for the *next* *HowMany breakpoints.

MDI 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.

Execute Command

A single command string is passed to the MDILib for parsing and execution. If an MDILib has no command parser, 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 of at least 80 characters, preferably longer, to be passed to the MDILib via this function without interpretation by the debugger.

MDIInt32 MDIDoCommand (Device, Buffer)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

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.

Display Output

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.

MDIInt32 MDICBOutput (Device, Type, Buffer, Count)

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.

Comments:

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 called an MDILib entry point.

Device will be MDINoHandle if the output is not associated with a particular device connection. 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.

Type specifies the type of output, and will be one of the following:

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

Get Input

This callback function is implemented by the debugger. 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.

MDIInt32 MDICBInput (Device, Type, Mode, Buffer, Count)

MDIHandleT	Device
MDIInt32	Type
MDIInt32	Mode
char **	Buffer
MDIInt32 *	Count

Returns:

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

Comments:

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 called an MDILib entry point.

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. 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).

Type specifies the type of input, and will be one of the following:

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

Mode specifies the mode of input, and will be one of the following:

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

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

Evaluate Expression

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.

MDIInt32 MDICBEvaluate (Device, Buffer, ResultType, SrcResource, SrcOffset, Size, Value)

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.

Comments:

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 one of the following in *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.

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

Lookup Resource

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.

MDIInt32 MDICBLookup (Device, Type, SrcResource, SrcOffset, Buffer)

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.

Comments:

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(). It is not expected that the debugger would need to access target resources to perform the lookup, but it is allowed to do so. So it may recursively call other MDI functions before returning.

The MDILib requests a particular type of symbolic information by passing one of the following values in 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	Depreciated – use MDILookupSourceLine
MDILookupSourceLine	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.
MDILookupSourceInfo	Debugger returns a NULL ended parsable string providing a complete representation of the given code addresses source information. The format of the string is: path="<path to source file>" name="<filename>" line=<startline>[-<endline>]

[column=<start column>[-<end column>]]
Support for this lookup is optional.

If the lookup is successful, the debugger returns the address of a buffer containing the resulting NUL 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.

Trace Data 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.

Enable Tracing

This function enables the tracing capabilities of the device. MDI assumes that, when enabled, trace data is captured only when the device is executing code. So 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), it is up to the MDILib to manage the device 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.

MDIInt32 MDITraceEnable (Device)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Disable Tracing

This function disables the tracing capabilities of the device. If the device is currently executing code, tracing will be halted immediately. 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.

MDIInt32 MDITraceDisable (Device)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Clear Trace Data

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, the debugger must call MDITraceDisable() before this function can be called.

MDIInt32 MDITraceClear (Device)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Query Trace Status

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, 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, MDITraceStatus() will return MDITraceStatusTracing if MDITraceEnable() has been called and the device is executing, else 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.

MDIInt32 MDITraceStatus (Device, Status)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Query Trace Data

This function returns the number of "frames" of trace data currently captured by the device. 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, 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. The debugger must call this function before calling MDITraceRead() to transfer actual trace data.

MDIInt32 MDITraceCount (Device, FrameCount)

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Read Trace Data

This function returns the requested range of "frames" of trace data 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:

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

Where Type specifies how to interpret the rest of the frame data, and is one of the following values:

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.

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, the MDILib is required to fetch the instructions from device memory so they can be included in the trace data.

MDIInt32 MDITraceRead (Device, FirstFrame, FrameCount, IncludeInstructions, Frames)

MDIHandleT	Device
MDIUint32	FirstFrame
MDIUint32	FrameCount
MDIUint32	IncludeInstructions
MDITrcFrameT *	Frames

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 can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

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, the MDILib will include the instruction values in the trace frame data.

Frames are numbered from 1 to N, where N is the total number of frames returned by MDITraceCount(). Frame 1 is the oldest frame.

MAJIC API Extensions

The chapter documents MAJIC specific extensions to the MDI Application Programming Interface. The extensions are designed to be optionally used by either an MDI client or and MDILib implementation.

Alternate Connection Control

The MDIMAJICConnect function described below establishes a connection to the MDILib using the passed version of MDI, and retrieves the configuration and supported MDI features. It mirrors the functionality provided by the normal MDIConnect function, but differs by allowing clients to provide a more integrated MAJIC experience. This is achived via the MDIMAJICConfigT structure which allow a client to specify the details of the actual target connection without having use external MDI configuration tools. The MajicConfig argument allows a client to fill in a simple structure and provide these startup details.

Multiple MDIMAJICConnect calls are made to establish connect information for multiple targets and/or cores. All connection information is thrown out at upon calling MDIDisconnect.

After a successful MDIMajicConnect call is made, an MDI client should proceed as if MDIConnect had just been called. Typically, a client would next call device query (MDIQuery) followed by MDIOpen to establish a MAJIC/target connection.

An MDIMAJICConnect call must not be made using a thread that has already been used to establish a session via MDIOpen.

MAJIC Connect

```
MDIInt32 MDIMAJICConnect (MDIVersion, MDIHandle, Config,
    MajicConfig)
        MDIVersionT          MDIVersion;
        MDIHandleT *         MDIHandle;
        MDIConfigT *         Config;
        MDIMAJICConfigT *    MajicConfig;
```

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 have 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.
MDIErrWrongThread	A thread was used that is already in use as a MDIOpen session thread.

MDIVersion, MDIHandle, and Config have the same semantics as they do in the normal MDIConnect call. Please see the MDIConnect description for details on them.

MajicConfig is a pointer to a MDIMajicConfigT structure shown below:

```
typedef struct MDIMAJICConfig_ST
{
   MDIUint32 Speed;             // 0..7
   MDIUint32 ConMode;           // Connection mode (MDI_CM_*)
   char      Port[ 256 ];
   char      CmdFile[ 1024 ];   // default startup cmd file
   char      Endian;            // MDIEndianBig or MDILittleEndian
   char      CPUId[ 16 ];       // a MAJIC supported cpu name
   char      LogFile[ 1024 ];   // optional log file for debugging
   char      ToolPath[ 1024 ];  // path to tools root
} MDIMAJICConfigT;
```

Speed is used to control the default speed of the serial port. Typically this value only matters if a serial device is specified in the Port parameter. But the virtual terminal option speed is also controlled via this parameter. See the MAJIC User's Manual for more details on this option.

ConMode tells MDI what to do once a target connection is established (reset the target, a

```
#define MDI_MAJIC_CM_DEFAULT 0 // No ConnectMode entry specified
#define MDI_MAJIC_CM_RESET   1 // Reset on connection (default)
#define MDI_MAJIC_CM_HALT    2 // Halt target at startup
#define MDI_MAJIC_CM_RUN     3 // Connect and leave target running
```

CmdFile is the path to a MAJIC specific initialization command file. An fully integrated MDI client debugger typically has it's own debugger methodology for initializing the target board environment. If so, this parameter should be left as a empty string.

Endian is must be MDIEndianBig or MDIEndianLittle.

CPUId is an ASCII name for cpu's supported by MAJIC. The list of support changes with every release and service pack. The current list can be displayed with this command:

```
Shell> majic_mon -vh
```

LogFile is the path to a file for logging all MDI call transactions. Normally, this entry should be an empty string.

ToolPath is the path to the root of the normal MDI installation directory. MDI uses this path to load default initialization files, cpu register description files, support files, and help files.

Option Control

The Option Control interface is an optional API in MDI. It provides a rich interface designed to allow MDILib to export a set of control options to a debugger. Typically, a debugger will provide both a GUI option interface and, some kind of command line based interface allowing users to query and set options. A debugger that employs a command language for scripting and startup files will find exporting this option interface into it's own language very useful in gaining full control of a target environment. The Option Control interface includes an option information call providing a full description capability. This allows for the option to be self documenting and limiting the need for an Option Control description in the debuggers User's Manual.

Many times options can be logically grouped into a related set of options and a debugger that knows about these groupings can use the information to present a logical user interface. MDI refers to option groupings as categories. Note that it is allowed for options to belong to more than one category. The call MDIGetOptionCategory queries the categories (names and values) to be queried.

Within the API, Options are identified by a Handle of type MDIOptionHandle. Query functions below allow all the options to be queried, and their values to be read and written. An option information structure called MDIOptionInfoT (below) can be queried and provides information about the option, its usage and meaning.

```
typedef struct MDIOptionInfo_Struct
{
    MDIOptionHandleT  Handle;
    MDIUint32         Type;
    MDIFlags32        Category;    // BitField
    MDIEnumHandleT    EnumHandle;  // Valid if Type is MDIOptionTypeEnum
    MDIFlags32        Flags;       // MDIOptionFlagXXX
    BOOL              Volatile;
    MDIUint32         NumBytes;    // Number of bytes in Value
    MDIUint64         Value[2];    // Not valid if Type is MDIOptionTypeText
    MDIUint64         MinValue;
    MDIUint64         MaxValue;    // Max value allowed (F's if none)
    const char        *Name;       // The option name
    const char        *ShortName;  // Optional- easy to type name
    const char        *Description;// A brief one line description
    const char        *FullDescription; // Detailed explanation, default, range
} MDIOptionInfoT;
```

The Value fields interpretation is based on the option type (MDIOptionType…) and the NumBytes field. For a simple 8, 16, 32 or 64 bit word value, the value[0] field can be cast to the right size size/type to obtain the value. Note that MDIGetOptionValueStr should be used to get the string value of any option of type MDIOptionTypeText. Values of type MDIOptionTypeAddr use Value[2] for the MDIResource.

MDILibs are encouraged to implement all the `MDIOptionInfoT` fields, however some fields (`ShortName, and FullDescription`), are optional. The fields above are mostly self explanatory. Note that option values are queried and set via MDIGetOptionValueStr() and MDISetOptionValue() calls listed below. The Flags field is described below:

```
#define MDIOptionFlagReadOnly  0x1
#define MDIOptionFlagVolatile  0x2
#define MDIOptionFlagPreInit   0x4  // Option available in no connect mode
                                    // (pre initialization)
#define MDIOptionFlagHidden    0x8  // Option not intended to be user visible
                                    // MDI Lib and client setup
```

The Type field is described below.

```
#define MDIOptionTypeEnum     0
#define MDIOptionTypeAddr     1         // (offset, and resource
#define MDIOptionTypeHex      2         // unsigned value
#define MDIOptionTypeDec      3         // signed value
```

```
#define MDIOptionTypeText     4
#define MDIOptionTypeIP       5          // Internet Protocol address
```

Enumerations (`MDIOptionTypeEnum`) can be used by options to provide text names for option states. For any given option that uses enumerations, the enumerations are queried by the call MDIGetOptionEnumNames. Note that these names cannot be documented here as they are specific to an MDILib instance. The structure type below defines an enumeration element. An array of this type would define a set of enumerations for a given EnumHandle as returned by MDIGetOptions.

```
typedef struct MDIOptionEnumInfo_Struct
{
    MDIInt32        Enum;
    const char      *Name;
} MDIOptionEnumInfoT;
```

The structure type below defines categories and their bit field values. Options are allowed to belong to multiple categories and categories provide a means to filter options when an MDILib instance contains a large number of options. Categories are considered an optional interface within the MDI option control interface. MDI Clients should be coded to allow for this to be unavailable.

```
typedef struct MDIOptionCategoryInfo_Struct
{
    MDIFlags32      Category;       // BitField
    const char      *Name;
} MDIOptionCategoryInfoT;
```

Options API

Common Option Arguments

MDIHandleT Handle
: Each of the MDI options related functions takes a Handle as the first argument. This handle must be a device handle as returned by MDIOpen.

MDIOptionHandleT Option
: Option is a handle of type MDIOptionHandle returned by a previous call to MDIGetOptions

MDIEnumHandleT EnumHandle
: MDIEnumHandle must be the value returned in the Handle member of the MDIOptionsInfoT array returned by a previous MDIGetOptions() call.

Options Query

Retrieve a list of options that are members of the the given category.

MDIInt32 MDIGetOptions (Handle, HowMany, Category, OptionInfo)

MDIHandleT	Handle
MDIInt32 *	HowMany
MDIFlags32	Category
MDIOptionInfoT *	OptionInfo

Returns:

MDISuccess	No error, command has been executed.
MDIErrMore	More options defined then requested.
MDIErrDevice	Invalid device handle.
MDIErrParam	Invalid parameter

MDIErrWrongThread	Call was not made by the connected thread.
MDIErrUnsupported	MDILib does not support Option Control Interface.

Comments:

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

The Category parameter can be determined from calls to MDIGetOptionCategories or it can be passed set to -1 (0xffffffff) which matches all categories. Note that Category is a bit field and can be composed of any set of valid options categories.

OptionInfo is a pointer to an array of type MDIOptionInfoT. It should be sized big enough to hold all the options requested by *HowMany.

Option Info Query

Retrieve option information for a given option.

MDIInt32 MDIGetOptionInfo (Handle, Option, OptionInfo)

MDIHandleT	Handle
MDIOptionHandleT	Option
MDIOptionInfoT *	OptionInfo

Returns:

MDISuccess	No error, command has been executed.
MDIErrDevice	Invalid device handle.
MDIErrParam	Invalid parameter
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrUnsupported	MDILib does not support Option Control Interface.
MDIErrFailure	Out of memory.

Comments:

The function fills in the OptionInfo structure with information about the selected Option.

Option Query Categories

Retrieve the next option category value and name.

MDIInt32 MDIGetOptionCategories (Handle, HowMany, CategoryInfo)

MDIHandleT	Handle
MDIInt32 *	HowMany
MDIOptionCategoryInfoT *	CategoryInfo

Returns:

MDISuccess	No Error, command has been executed.
MDIErrDevice	Invalid Device handle.
MDIErrMore	More options defined then requested.
MDIErrParam	Invalid parameter.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrUnsupported	MDILib does not support Option Control.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

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

CategoryInfo is a pointer to an array of type MDIOptionCategoryInfoT. It should be sized big enough to hold all the categories requested by *HowMany.

Option Query Enumeration Names

Retrieve the next enumeration name for the given option handle.

MDIInt32 MDIGetOptionEnumInfo (Handle, EnumHandle, HowMany, EnumInfo)

MDIHandleT	Handle
MDIEnumHandleT	EnumHandle
MDIInt32 *	HowMany
MDIOptionEnumInfoT *	EnumInfo

Returns:

MDISuccess	No Error, command has been executed.
MDIErrMore	More options defined then requested.
MDIErrDevice	Invalid Device handle.
MDIErrParam	Invalid parameter.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrUnsupported	MDILib does not support Option Control.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

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

EnumInfo is a pointer to an array of type MDIOptionEnumInfoT. It should be sized big enough to hold all the enumerations requested by *HowMany.

Option Get Value String

Retrieves the value of the passed Option as an ASCII string into ValueStr.

MDIInt32 MDIGetOptionValueStr (Handle, Option, ValueStr)

MDIHandleT	Handle
MDIOptionHandleT	Option
char *	ValueStr

Returns:

MDISuccess	No Error, command has been executed.
MDIErrDevice	Invalid Device handle.
MDIErrParam	Invalid parameter.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrUnsupported	MDILib does not support Option Control.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

ValueStr should be allocated as 256 characters array. MDI does not allow for longer value strings. The returned string is suitable as input to the MDISetOptionValueStr function.

Note:

If the option is of type MDIOptionTypeAddr then the option's value string contents is displayed in the MDILib's ASCII representation of address and MDIResource (space) information.

Option Set Value String

Sets the given Option to the passed value string in ValueStr.

MDIInt32 MDISetOptionValueStr (Handle, Option, ValueStr, ErrText)

MDIHandleT	Handle
MDIOptionHandleT	Option
char *	ValueStr
char *	ErrText

Returns:

MDISuccess	No Error, command has been executed.
MDIErrDevice	Invalid Device handle.
MDIErrParam	Invalid parameter.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrUnsupported	MDILib does not support Option Control.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

ValueStr should not exceed 256 bytes as MDI does not allow for longer value strings. Case sensitivity of enumerations is not defined and left to the MDILib desires. ErrText is returned as non NULL if an error in the ValueStr is encountered.

Note:

If the option is of type MDIOptionTypeAddr then the string contents is parsed for address and MDIResource (space) information in the native format of the MDILib implementation. Clients that have their own resource representations should use MDISetOptionValue to update options.

Option Set Value

Sets the given Option to the passed value passed in Value.

MDIInt32 MDISetOptionValue (Handle, Option, Value, ErrText)

MDIHandleT	Handle
MDIOptionHandleT	Option
void *	Value
char *	ErrText

Returns:

MDISuccess	No Error, command has been executed.
MDIErrDevice	Invalid Device handle.
MDIErrParam	Invalid parameter.
MDIErrWrongThread	Call was not made by the connected thread.
MDIErrUnsupported	MDILib does not support Option Control.
MDIErrTargetRunning	Service can not be performed at this time because the target program is running.
MDIErrRecursive	Recursive call was made during an MDICBPeriodic() callback.

Comments:

Value is a pointer to a type that is dependent upon the option type, ErrText is returned as non NULL if an error in the ValueStr is encountered.

Option Pre-Initialization.

Many of the MAJIC options can be thought of as options that need to be setup for a particular environment and then left alone. A user would typically want to setup such options at target configuration time. The options interface supports this concept by allowing either a disconnected (to MAJIC) connection via MDIMajicConnect() and MDIOpen() call sequence, or similar connected to MAJIC, but not to target call sequence. The latter approach is preferred as it allows the connection to the unit to qualify options as being valid and pertinent to the MAJIC model and setup environment. However, in multi-core environments, a connected option setup as described here would potentially effect the global option values of potentially running sessions.

Method 1: Disconnected Option Setup

To connect in this mode two members in the `MDIMAJICConfigT` parameter of your MDIMAJICConnect() call must be set as follow:

`CmdFile`	Set to empty (“\0”).
`ConMode`	Must be set to `MDI_MAJIC_CM_NO_CONNECTION`.

The rest of the parameters should be passed per normal making sure the CPUId field of `MDIMAJICConfigT` is correct for the configured instance and valid callbacks are configured for the MDIConfigT parameter. Now `MDIOpen` can be called without it actually making a physical connection to a MAJIC probe.

This connect mode will only return any options that are setup with the MDIOptionFlagPreInit flag set.

Note that the lack of a real connection to a MAJIC probe does not allow for the probe to qualify any options as being valid for the given probe model and setup. Consequently, some options may be visible in a

configuration dialog, but not valid on startup. Options such as these can be simply ignored (e.g, not updated upon a real connection).

Remember that MDIDisconnect should only be called when all your MDIOpen

Method 2: Connected Option Setup (Preferred)

This mode is similar to the above Method 1, but allows for options to be qualified as valid for the connected environment. This is not full proof as some options might be further qualified by connection to target hardware which is purposefully not done here. There few examples of this and they are typically modify range values, as opposed to option itself being valid.

To connect in this mode two members in the `MDIMAJICConfigT` parameter to your MDIMAJICConnect() call must be set as follow:

`CmdFile`	Set to empty ("\0").
`ConMode`	Must be set to `MDI_MAJIC_CM_RUN`.

The rest of the parameters should be passed per normal making sure the CPUId field of `MDIMAJICConfigT` is correct for the configured instance and valid callbacks are configured for the MDIConfigT parameter. `MDIOpen` can now be called to make a connection to the configured MAJIC probe. In this mode MDIGetOptions returns all options that are valid for the environment. This will include many options that should not appear in a pre-initialized dialog. For such a case, the caller is expected to filter the options based on the MDIOptionFlagPreInit type.

Option Playback

Below is the recommend sequence of operations to do when starting a debug connection/session. This allows the options to set and qualified before startup file playback.

- Configure CmdFile option as empty.
- MDIMAJICConnection()
- MDIOpen()
- Option Playback – MDISetOptionValue(), or MDIDoCommand("eo <option name> = <value>")
- MDIDoCommand("fr c <startup file if any>")

Appendix A – MDI.H Header File

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

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

#ifndef MDI_Specification_Definitions
#define MDI_Specification_Definitions


/*
    When building MDILib:
        Define MDI_LIB before #include "mdi.h"

    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;
#else
typedef unsigned long long MDIUint64;
typedef long long          MDIInt64;

#ifndef __stdcall
#define __stdcall
#endif

#endif

typedef MDIUint32          MDIVersionT;

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

#define MDIMajor           0x0001
#define MDIMinor           0x0000
#define MDIOldMajor        0x0001
#define MDIOldMinor        0x0000
#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 MDIInstructionRange_struct
{
    MDIRangeT       Range;
    MDIResourceT    Space;
} MDIInstructionRangeT;

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 */
   MDIInt32  (__stdcall *MDICBOutput)
         (MDIHandleT Device, MDIInt32 Type,
         char *Buffer, MDIInt32 Count );

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

```
                                    /* Host:  CB fn for MDI input */
   MDIInt32  (__stdcall *MDICBEvaluate)
         (MDIHandleT Device, char *Buffer,
         MDIInt32 *ResultType, MDIResourceT *Resource,
         MDIOffsetT *Offset, MDIInt32 *Size, void **Value);
                                    /* Host:  CB fn for expression eval */
   MDIInt32  (__stdcall *MDICBLookup)
         (MDIHandleT Device, MDIInt32 Type,
         MDIResourceT Resource, MDIOffsetT Offset,
         char **Buffer );
                                    /* Host:  CB fn for sym/src lookup */
   MDIInt32  (__stdcall *MDICBPeriodic)(MDIHandleT Device);
                                    /* Host:  CB fn for Event processing */
   MDIInt32  (__stdcall *MDICBSync)(MDIHandleT Device, MDIInt32 Type,
         MDIResourceT Resource);
                                    /* Host:  CB fn for Synchronizing */
} 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 */


typedef struct MDIRunState_struct
{
    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 */
#define MDIStatusMask           0x00ff  /* Status values are in lowest byte */

/* These can be OR'ed in with MDIStatusRunning and MDIStatusNotRunning */

#define MDIStatusReset          0x0100  /* currently held reset */
#define MDIStatusWasReset       0x0200  /* reset asserted & released */
#define MDIStatusResetMask      0x0300  /* reset state mask */
/* This can also be OR'ed in with MDIStatusHalted */
#define MDIStatusDescription    0x0400  /* Info.ptr = Descriptive string */
/* This can OR'ed in with MDIStatusNotRunning to indicate that a core is in the
middle of executing an instruction (the implication is that it was suspended
by some sort of multicore synchronization mechanism)  */
#define MDIStatusInstructionPending   0x0800
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_TypeMax           MDIBPT_HWBus
#define MDIBPT_TypeMask          0x0000ffff

/* Hardware breakpoint types may have one or more of the following */
/* flag bits OR'ed in to specify additional qualifications.   */

#define MDIBPT_HWFlg_AddrMask    0x00010000
#define MDIBPT_HWFlg_AddrRange   0x00020000
#define MDIBPT_HWFlg_DataValue   0x00040000
#define MDIBPT_HWFlg_DataMask    0x00080000
#define MDIBPT_HWFlg_DataRead    0x00100000
#define MDIBPT_HWFlg_DataWrite   0x00200000
#define MDIBPT_HWFlg_Trigger     0x00400000
#define MDIBPT_HWFlg_TriggerOnly 0x00800000
#define MDIBPT_TypeQualMask      0xffff0000

typedef MDIUint32       MDIBpIdT;
#define MDIAllBpID      -1

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;


/* Following flags and struct are part of MIPS' MDI 2.2 specification */
#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  9
#define MDIBPT_HWType_VirtAddr   0x4000
#define MDIBPT_HWType_ASID       0x8000

typedef struct MDIBpInfo_struct
{
    MDIInt32 Num;
    MDIBpT 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

/* Bit Values for MDIStepRange Mode Flag orable in with normal Step mode value
above*/

#define MDIStepInRange         0x0000
#define MDIStepOutsideRange 0x1000

/* "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 MDICacheFlush(): */

#define MDICacheWriteBack    0x01
#define MDICacheInvalidate   0x02

/* 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

/* 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, also used in option queries */

#define MDIErrFailure               -1
   /* Unable to perform operation. */
#define MDIErrDevice                -2
   /* Invalid Device handle. */
#define MDIErrSrcResource           -3
   /* Invalid Resource type. */
#define MDIErrDstResource           -4
   /* 2nd Resource has invalid type. */
#define MDIErrInvalidSrcOffset      -5
   /* Offset is invalid for the specified resource. */
```

```
#define MDIErrInvalidDstOffset      -6
   /* 2nd Offset is invalid for the 2nd resource. */
#define MDIErrSrcOffsetAlignment    -7
   /* Offset is not correctly aligned. */
#define MDIErrDstOffsetAlignment    -8
  /* 2nd Offset is not correctly aligned for the specified ObjectSize */
#define MDIErrSrcCount              -9
   /* Count causes reference outside of the resources space */
#define MDIErrDstCount              -10
   /* Count causes reference outside of 2nd resources 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 are unavailable. */
#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 MDIErrDeviceHandle          -34
#define MDIErrDevicesOpen           -35
#define MDIErrInvalidData           -36
#define MDIErrDuplicateBP           -37
#define MDIErrInvalidFrames         -38
   /* Range of requested trace frames is invalid */
#define MDIErrWrongThread           -39
#define MDIErrTargetRunning         -40
#define MDIErrRecursive             -41
   /* Illegal recursive call from from MDICDPeriodic */
#define MDIErrObjectSize            -42
   /* Invalid Object Size for Resource */


/* Function Prototypes */
```

```
#ifdef __cplusplus
extern "C" {
#endif

#ifndef yf  // allow caller to provide their own yf if needed
    #ifdef _WIN32
        #ifndef MDI_DLL_EXPORT
            #define MDI_DLL_EXPORT __declspec( dllexport )
        #endif
    #else  // some flavor of unix/linux
        #ifndef MDI_DLL_EXPORT
            #define MDI_DLL_EXPORT // linux doesn't require anything special
here
        #endif
    #endif
    #if defined( MDI_LIB )
        // MDILib, do extern function declarations
        #define yf(str) extern MDI_DLL_EXPORT int __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 MDI_DLL_EXPORT int (__stdcall *str)
    #endif
#endif

yf(MDIVersion)(MDIVersionRangeT *);
yf(MDIConnect)(MDIVersionT, MDIHandleT *, MDIConfigT *);
yf(MDIDisconnect)(MDIHandleT, MDIUint32);
yf(MDITGQuery)(MDIHandleT, MDIInt32 *HowMany, MDITGDataT *);
yf(MDITGOpen)(MDIHandleT, MDITGIdT, MDIUint32, MDIHandleT *);
yf(MDITGClose)(MDIHandleT, MDIUint32);
yf(MDITGExecute)(MDIHandleT);
yf(MDITGStop)(MDIHandleT);
yf(MDIDQuery)(MDIHandleT, MDIInt32 *HowMany, MDIDDataT *);
yf(MDIOpen)(MDIHandleT, MDIDeviceIdT, MDIUint32, MDIHandleT *);
yf(MDIClose)(MDIHandleT, MDIUint32);
yf(MDIRead)(MDIHandleT, MDIResourceT SrcR, MDIOffsetT SrcO,
        void * Buffer, MDIUint32 ObjectSize, MDIUint32 Count);
yf(MDIWrite)(MDIHandleT, MDIResourceT DstR, MDIOffsetT DstO,
        void * Buffer, MDIUint32 ObjectSize, MDIUint32 Count);
yf(MDIReadList)(MDIHandleT, MDIUint32 ObjectSize, MDICRangeT *SrcList,
        MDIUint32 ListCount, void * Buffer );
yf(MDIWriteList)(MDIHandleT, MDIUint32 ObjectSize, MDICRangeT *DstList,
        MDIUint32 ListCount, void * Buffer );
yf(MDIMove)(MDIHandleT, MDIResourceT SrcR, MDIOffsetT SrcO,
        MDIResourceT DstR, MDIOffsetT DstO, MDIUint32 ObjectSize,
        MDIUint32 Count, MDIUint32 Flag);
yf(MDIFill)(MDIHandleT, MDIResourceT DstR, MDIRangeT DstRng,
        void * Buffer, MDIUint32 ObjectSize, MDIUint32 Count);
yf(MDIFind)(MDIHandleT, MDIResourceT SrcR, MDIRangeT SrcRng,
        void * Buffer, void * MaskBuffer, MDIUint32 ObjectSize,
        MDIUint32 Count, MDIOffsetT *FoundOffset, MDIUint32 Mode);
yf(MDIExecute)(MDIHandleT);
yf(MDIStep)(MDIHandleT, MDIUint32 Steps, MDIUint32 Mode);
yf(MDIStepRange)(MDIHandleT, MDIUint32 Mode,MDIInstructionRangeT *Ranges,
        MDIUint32 *RangeCount);
yf(MDIStop)(MDIHandleT);
yf(MDIReset)(MDIHandleT, MDIUint32 Flag);
yf(MDICacheQuery)(MDIHandleT, MDICacheInfoT *CacheInfo);
yf(MDICacheFlush)(MDIHandleT, MDIUint32 Type, MDIUint32 Flag);
yf(MDIRunState)(MDIHandleT, MDIInt32 WaitTime, MDIRunStateT *runstate);
yf(MDISetBp)(MDIHandleT, MDIBpDataT *);
yf(MDISetSWBp)(MDIHandleT , MDIResourceT , MDIOffsetT , MDIBpIdT *);
yf(MDIClearBp)(MDIHandleT, MDIBpIdT);
```

```
yf(MDIEnableBp)(MDIHandleT, MDIBpIdT);
yf(MDIDisableBp)(MDIHandleT, MDIBpIdT);
yf(MDIBpQuery)(MDIHandleT, MDIInt32 *HowMany, MDIBpDataT *);
yf(MDIDoCommand)(MDIHandleT, char *Buffer);
yf(MDIAbort)( MDIHandleT );
yf(MDITraceEnable)(MDIHandleT);
yf(MDITraceDisable)(MDIHandleT);
yf(MDITraceClear)(MDIHandleT);
yf(MDITraceStatus)(MDIHandleT, MDIUint32 *);
yf(MDITraceCount)(MDIHandleT, MDIUint32 *);
yf(MDITraceRead)(MDIHandleT, MDIUint32, MDIUint32, MDIUint32, MDITrcFrameT *);

/* HWBP capability query function from MDI 2.2 */
yf(MDIHwBpQuery) (MDIHandleT, MDIInt32*, MDIBpInfoT*);

#undef yf

#ifdef __cplusplus
}
#endif

#endif

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

Appendix B – MIPS Addendum

This Addendum to the MDI specification provides details for MDI implementations targeting CPUs following any MIPS architecture.

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 as follows:

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"

MIPS Exception Codes

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

MIPS16 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 is on a normal 32-bit instruction or a MIPS16 instruction. Also, it is necessary for the debugger to know whether an instruction trace frame returned by MDITraceRead() is a MIPS16 instruction or not. For both cases, MIPS16 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.

MIPS Resources

The following table lists the specific resource encodings (address spaces) defined for the MIPS architecture. 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, and MDIMIPDFP (if double precision is available).

It is strongly recommended that MIPS MDILib implementations support all encodings for resources that the target system actually provides.

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.

MIPS Resource	MDI Mnemonic	Offset Definition
HI/LO Registers	MDIMIPHILO	Offset is 0 for register HI, 1 for register LO
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.
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 or Unified Cache Tags	MDIMIPPICACHET MDIMIPPUCACHET	This space is organized as an array of cache tag entries. Each cache tag entry consists of two registers, cache tag followed by cache parity. 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 twice the total number of cache tag entries. For multi-set caches,

MIPS Resource	MDI Mnemonic	Offset Definition
		all of the cache tag entries for set 0 are followed by all of the cache tag entries for set 1, etc.
Secondary Instruction or Unified Cache Tags	MDIMIPSICACHET MDIMIPSUCACHET	See Primary Instruction or Unified Cache Tags' offset definition above.
Primary Data Cache Tags	MDIMIPPDCACHET	See Primary Instruction or Unified Cache Tags' offset definition above.
Secondary Data Cache Tags	MDIMIPSDCACHET	See Primary Instruction or Unified Cache Tags' offset definition above.
Primary Instruction or Unified Cache	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.
Secondary Instruction or Unified Cache	MDIMIPSICACHE MDIMIPSUCACHE	See Primary Instruction or Unified Cache offset definition above.
Primary Data Cache	MDIMIPPDCACHE	See Primary Instruction or Unified Cache offset definition above.
Secondary Data Cache	MDIMIPSDCACHE	See Primary Instruction or Unified Cache offset definition above.
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.

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      5
#define MDIMIPPDCACHET      6
#define MDIMIPSICACHET      7
#define MDIMIPSUCACHET      7
#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       20
#define MDIMIPPDCACHE       21
#define MDIMIPSICACHE       22
#define MDIMIPSUCACHE       22
#define MDIMIPSDCACHE       23
#define MDIMIPPHYSICAL      24
#define MDIMIPGVIRTUAL      25
#define MDIMIPEJTAG         26

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

/*
** For MDISetBp(), MDISetSWBp(), and MDITraceRead(), setting the low
** order address bit to 1 means that the addressed instruction is a
```

```
** MIPS16 instruction.
*/
#define MDIMIP_Flg_MIPS16   1

#endif

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

Appendix C – PowerPC Addendum

This Addendum to the MDI specification provides details for MDI implementations targeting CPUs following the PowerPC architectures.

PowerPC MDIDDataT Fields

Valid values for the MDIDDataT.FFamily and MDIDDataT.FISA fields returned by MDIDQuery() are architecture specific. For PowerPC, MDIDDataT.FFamily must be set to MDIPPC_FClass ("PPC"). Valid values for MDIDDataT.FISA are as follows:

MDIPPC_FISA_3BOOK	"3BOOK"
MDIPPC_FISA_BOOKE	"BOOKE"

PowerPC Exception Codes

When MDIRunState() returns RunState.Status = MDIStatusException, the meaning of RunState.Info.value is architecture-specific. For PowerPC does not use this field.

PowerPC Resources

The following table lists the specific resource encodings (address spaces) defined for the PowerPC architecture. The "Programming Mnemonic" is the macro name defined in the header file `mdippc.h`, made available with this MDI addendum.

As a minimum, all PowerPC MDILib implementations are required to support the checked encodings: show in the table below.

It is strongly recommended that PPC MDILib implementations support all encodings for resources that the target system actually provides.

PPC Resource	**MDI Mnemonic**		**Offset Definition**
CPU General Registers	MDIPPC_GPR	√	Offset is the register number, 0-31
Special Purpose Registers	MDIPPC_SPR	√	Offset is the register number (0..1024)
Device Control Registers	MDIPPC_DCR	√	Offset is the register number (0..1024)
Floating Point Data Registers	MDIPPC_FPR	√	Offset is the register number (0-31). Note: Always 64 bit precision format.
Program Counter Register	MDIPPC_PC	√	Offset must be 0
Condition Reg1ister	MDIPPC_CR	√	Offset must be 0
Machine State Register	MDIPPC_MSR		Offset must be 0
Floating Point Status Register	MDIPPC_FPSCR		Offset must be 0
TLB Registers	MDIPPC_TLB		Offset is entry number * (words/entry) + word number
SLB Registers	MDIPPC_SLB		Offset is entry number * 2 + (word number, 0 or 1)
Vector Status and Control Register	MDIPPC_VSCR		Offset must be 0, value is 32 bits wide

PPC Resource	MDI Mnemonic		Offset Definition
Vector Registers	MDIPPC_VECTOR		Offset = register number 0..31. Each register is 128 bits
Physical memory	MDIPPCPHYSICAL	√	Offset is the physical byte address.
Global Virtual Data Memory	MDIPPCGDVIRTUAL		Offset is the data virtual byte address
Global Virtual Instruction Memory	MDIPPCGIVIRTUAL		Offset is the instruction virtual byte address
Virtual Address Space 0, PID value *XXXXXXXX*	MDIPPCVIRTUAL0		Offset is address space 0, PID value *XXXXXXXX*
Virtual Address Space 1, PID value *XXXXXXXX*	MDIPPCVIRTUAL1		Offset is address space 1, PID value *XXXXXXXX*
Virtual memory addresses	MDIPPCVIRTUAL		Offset is effective byte address

PowerPC Specific Header File

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

```
/* Start of header file for PowerPC Specific MDI  (MDIppc.h) */

#ifndef MDI_PPC_Specification_Definitions
#define MDI_PPC_Specification_Definitions

/* Valid values for MDIDDataT.FClass: */
#define MDIPPC_FClass       "PPC"
/* Valid values for MDIDDataT.FISA: */
#define MDIPPC_FISA_3BOOK  "3BOOK"      /* As specified by Books I-III */
#define MDIPPC_FISA_BOOKE  "BOOKE"      /* As specified by Book E */

/* Valid values for Resource */

    /* General Purpose Registers. Offset == register number (0-31) */
#define MDIPPC_GPR            1
    /* Special Purpose Registers. Offset == register number (0-1023)
     * Note: the register number is the 10-bit SPR field of the mtspr/mfspr
     * instruction.  The existence and architectural meaning of each
     * register depends on the specific target processor.
     */
#define MDIPPC_SPR            2
    /* Device Control Registers. Offset == register number (0-1023)
     * Note: the register number is the 10-bit DCR field of the mtdcr/mfdcr
     * instruction.  The existence and architectural meaning of each
     * register depends on the specific target processor.
     * Note: some PPC implementations include additional device control
     * registers that are accessed by writing an address (register selector)
     * to a DCR and then reading/writing the datum via a second
     * DCR.  The MDI specification intentionally does not provide
     * resource encodings for direct access to these IMRs (Indirectly
     * Mapped Registers). Debuggers should access them by manipulating the
     * appropriate address and data DCRs, just as target code would do.
     */
#define MDIPPC_DCR            3
    /* Floating Point data registers. Offset == register number (0-31)
     * Note: FPRs are always 64-bit double precision format.
     */
#define MDIPPC_FPR            4
```

```
    /* Location counter pseudo-register.  Offset == 0 */
#define MDIPPC_PC          5
    /* Condition Reg1ister.  Offset == 0 */
#define MDIPPC_CR          6
    /* Machine State Register.  Offset == 0 */
#define MDIPPC_MSR             7
    /* Floating Point Status Register. Offset == 0 */
#define MDIPPC_FPSCR       8
    /* "Book E" compliant MMU implementations have a software accessible
     * Translation Lookaside Buffer.  The number of entries and the
     * size and format of each entry are implementation specific, as
     * are the semantics of the read/write instructions.
     * Conventionally, the tlbre/tlbwe instructions divide each TLB
     * entry into a set of GPR sized words and transfer the words between
     * the entry and the GPRs. So the TLB space is modeled as an array of
     * GPR-sized registers consisting of the N words for entry 0 followed by
     * the N words for entry 1, etc.  The contents of each word match the
     * layout of the GPRs documented by the implementation for the
     * tlbre/tlbwe instructions.
     */
    /* TLB registers.  offset = (entry number) * (words/entry) + (word number)
*/
#define MDIPPC_TLB             9
    /* "Book III" compliant MMU implementations have a software accessible
     * Segment Lookaside Buffer containing an implementation-dependent
     * number of 94 bit entries.  The slbmte and slbmfev/slbmfee instructions
     * map the fields of each entry onto a pair of 64-bit GPRs.  So the SLB
     * space is modeled as as an array of 64-bit registers consisting of the
     * 2 words for entry 0 followed by the 2 words for entry 1, etc.  For
     * each entry, the first (even) word corresponds to the slbmte
     * instruction's "RS" register (VSID) and the second (odd) word
     * corresponds to the slbmte instruction's "RB" register (ESID).
     */
    /* SLB registers.  offset = (entry number) * 2 + (word number, 0 or 1) */
#define MDIPPC_SLB             10

    /* Physical memory addresses (bypassing MMU).  Offset = Real Address */
#define MDIPPCPHYSICAL      11

    /* Virtual memory addresses.
     * The resource encodings needed for virtual addressing are heavily
     * dependent on the MMU architecture of the target processor.  For
     * PPC, there are two main MMU architectures defined.
     *
     * The MMU architecture specified by "Book E" ("Enhanced PowerPC") has
     * a software-accessible TLB and a software-driven reload model with
     * no specified in-memory page table format.  In addition to the
     * 32- or 64-bit "Effective Address", TLB lookups are always qualified
     * by a separate one-bit "Address space" for Instruction fetch and
     * Load/Store (data) accesses.  Also, lookups can further qualified by
     * an implementation-defined "Process ID" (PID) of up to 32 bits.
     * A PID value of 0 signifies a "global" entry where PID matching is
     * suppressed.
     * The following to resources are used to access global (PID=0) Data and
     * Instruction memory via the corresponding current Address Space bit
     * in the Machine State register.
     */
    /* Global Virtual Data memory addresses.  Offset = Effective Address */
#define MDIPPCGDVIRTUAL        12
    /* Global Virtual Instruction memory addresses.  Offset = Effective Address
*/
#define MDIPPCGIVIRTUAL        13
    /*
     * The following two equates define resource ranges to provide access
     * to virtual addresses with explicit values for the "Address Space"
     * flag and PID value.  The defined ranges only support PIDs up to 24
     * bits, not the full 32 bits allowed by Book E.  This is not expected to
```

```
     * be a problem in practice, since most implementations use an 8 bit PID.
     */
    /* Address Space 0, PID value xxxxxxxx */
#define MDIPPC_VSCR           14
    /* Vector Registers.  Offset = register number 0..31.  Each register is 128
    * bits wide
    */
#define MDIPPC_VECTOR     15
    /*
     * The following two equates define resource ranges to provide access
     * to virtual addresses with explicit values for the "Address Space"
     * flag and PID value.  The defined ranges only support PIDs up to 24
     * bits, not the full 32 bits allowed by Book E.  This is not expected to
     * be a problem in practice, since most implementations use an 8 bit PID.
     */
    /* Address Space 0, PID value xxxxxxxx */#define MDIPPCVIRTUAL0
0x80000000  /* 0x80xxxxxx: xxxxxx == PID value */
    /* Address Space 1, PID value xxxxxxxx */
#define MDIPPCVIRTUAL1       0xC0000000  /* 0xC0xxxxxx: xxxxxx == PID value */

    /*  "Book III"
     * ("Classic PowerPC") specifies a three-stage MMU with a table of
     * software accessible Segment descriptors, an inaccessible TLB,
     * and hardware driven in-memory page tables.  There are no address
     * space qualifiers applied to the original "Effective Address", so
     * a single resource encoding is all that's needed.
     * Debuggers that don't support multiple address spaces should also
     * use this encoding
     */
    /* default Virtual memory addresses.  Offset = Effective Address */
#define MDIPPCVIRTUAL     12

#endif
/* End of header file for PowerPC Specific MDI (MDIppc.h) */
```

Appendix D – ARM Addendum

This Addendum to the MDI specification provides details for MDI implementations targeting CPUs following the ARM architecture.

ARM 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 ("ARM"). Valid values for MDIDDataT.FISA are as follows:

MDIARM_Fclass "ARM"

Thumb Instructions

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

ARM Resources

The following table lists the specific resource encodings (address spaces) defined for the ARM architecture. The "Programming Mnemonic" is the macro name defined in the header file `mdiarm.h`, made available with this MDI addendum.

As a minimum, all ARM MDILib implementations are required to support the following encodings: MDIARMCURRENT, MDIARMSTATUS, MDIARMPC MDIARMPHYSICAL, and MDIARMGVIRTUAL. MDIARMGVIRTUAL support may be limited to the physically mapped segments..

It is strongly recommended that MIPS MDILib implementations support all encodings for resources that the target system actually provides.

ARM Resource	**MDI Mnemonic**	**Offset Definition**
CPU General Registers	MDIARMCURRENT	Offset is a register number, 0-15
CPU User Mode Registers	MDIARMUSER	Offset is a register number: 0..15*
CPU SVC Mode Registers	MDIARMSVC	Offset is a register number: 0..15*
CPU IRQr Mode Registers	MDIARMIRQ	Offset is a register number: 0..15*
CPU FIQ Mode Registers	MDIARMFIQ	Offset is a register number: 0..15*
CPU ABORT Mode Registers	MDIARMABORT	Offset is a register number: 0..15*
CPU User Mode Registers	MDIARMUNDEF	Offset is a register number: 0..15*
PC Pseudo Register	MDIARMPC	Offset will be 0.
Status/Control Registers	MDIARMSTATUS	ARM v4-7: cpsr, spsr, spsr_svc, spsr_abort, spsr_undef, spsr_irq, spsr_fiq ARM v7m parts:

ARM Resource	MDI Mnemonic	Offset Definition
		apsr..control. (llist of registers as documented in ARM v7m architecture manual – Section titled: Special register encoding use in ARMv7-M system instructions)..
CPU Coprocessor Registers	MDIARMCP*x* *x* = 0 ..15	For 32 bit registers:: The MCR/MRC instructions include four fields that can be combined by the CP to select a register: Op1 (b23:21, CP specific opcode bits), Rn (b19:16, the CP src/dest reg), Op2 (b7:5, CP specific opcode bits), and Rm (b3:0, the "additional * CP register"). The 16384 possible registers are addressed as follows: Offset == Op1 << 11 | Rm << 7 | Op2 << 4 | Rn For 64-bit registers: the MCRR/MRRC instructions include two fields that can be combined by the CP to select a register: Op (b7:4, CP specific opcode bits), and Rm (b3:0, the CP src/dest reg). The 128 possible registers addressed as follows: Offset == Op << 4 | Rm Offset is register number: 0..15 *
Physical Memory	MDIARMPHYSICAL	Offset is the physical byte address.
Global Virtual Memory	MDIARMGVIRTUAL	Offset is the virtual byte address.

* Resource spaces with this mark are not available on ARMv7m parts.

Appendix E – Basic Connection to MDI example

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

#include <windows.h>
#include <stdio.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( MDIErrDeviceHandle         ),
   ec( MDIErrDevicesOpen          ),
   ec( MDIErrInvalidData          ),
   ec( MDIErrDuplicateBP          ),
   ec( MDIErrInvalidFrames        ),
   ec( MDIErrWrongThread          ),
   ec( MDIErrTargetRunning        ),
   ec( MDIErrRecursive            ),
   ec( MDIErrObjectSize           ),
      0, "Undefined"
      };

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

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);
      }
retry:
   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);
   gets(buffer);
   value = atoi(buffer);
   if (value < 1 || value > number)
      {
      goto retry;
      }

   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);
      }
retry:
   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);
   gets(buffer);
```

```
   value = atoi(buffer);
   if (value < 1 || value > number)
      {
      goto retry;
      }

   return (value - 1);
}


/*****************************************************************
*
*
*   openDev
*           Creates an array of the available devices in the target
*           group.  If more than 1, it queries the user as to which
*           device it wants to connect.
*
*   Returns If successful on device open, DevHandle is set and 0
*           is returned
*           If error, than a number < 0 is returned to indicate
*           the error
*
*
*****************************************************************/
int
openDev()
{
   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, TGhandle is set and 0
*           is returned
*           If error, than a number < 0 is returned to indicate the
*           error
*
*
*****************************************************************/
int
openTG()
{
   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 char linebuf[ 1024 ];

    *Buffer = fgets( linebuf, 1024, 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 successful
*           number < 0 if error
*
*
*****************************************************************/
int
opendevice()
{
   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 successful
*           number < 0 if error
*
*
*****************************************************************/
int
closedevice()
{
    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);

}
```

Support Information

If you have questions about this API, please log in to SupportNet. You may search thousands of technical solutions, view documentation, or open a Service Request online at:

http://supportnet.mentor.com/

If your site is under current support and you do not have a SupportNet login, you may easily register for SupportNet by filling out the short form at:

http://supportnet.mentor.com/user/register.cfm

All customer support contact information can be found on our web site at:

http://supportnet.mentor.com/contacts/supportcenters/

Index of MDI Functions

Revision History

Revision	Date	Comments
1.0	10/31/2000	Original Release
1.1	05/14/2007	Additions – Options API, MDIMAJICConnect, PowerPC
1.2	08/24/2009	Additions – StepRange, MDIGetOptionInfo
1.3	05/01/2011	Additions – Discovery, ConnectSetup, ARM and PowerPC resource descriptions,
1.4	6/23/2011	Additions - CPUIdsQuery