# Intel® IXP2400/IXP2800 Network Processors

## Intel XScale® Core Support Libraries Reference Manual

*November 2003*

Part Number: 278736-005

## Revision History

| Revision Date | Revision | Description |
|---|---|---|
| 1/2003 | 001 | Version 3.0 SDK Pre Release 6 |
| 6/2003 | 002 | Release for IXA SDK 3.1 Pre release 2. |
| 7/2003 | 003 | Release for IXA SDK 3.1 Pre release 3. |
| 9/2003 | 004 | Release for IXA SDK 3.5 PR-1. |
| 11/2003 | 005 | Updated Intel XScale® core trademark. |

# Contents

## Figures

## Tables

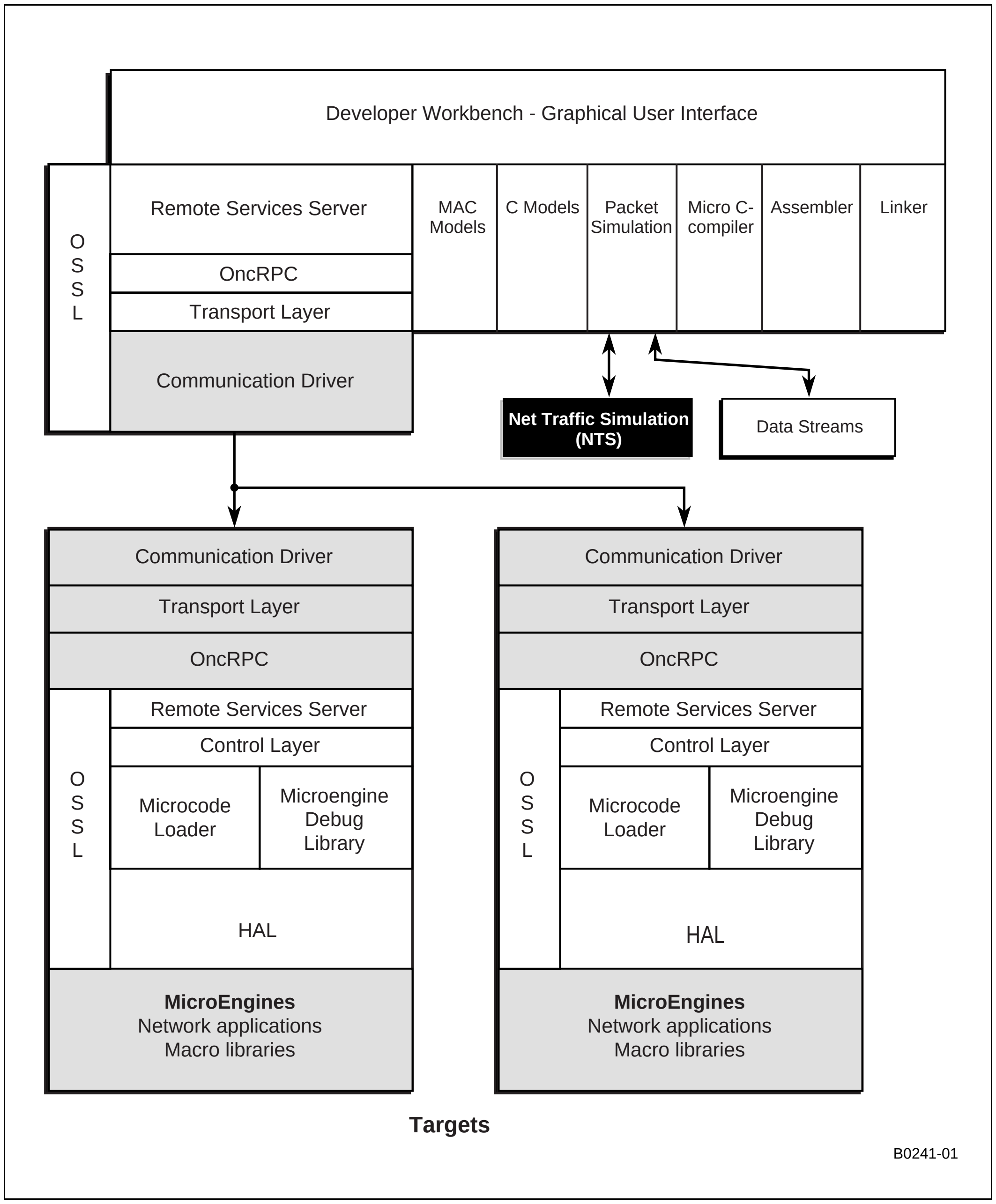
# Overview 1

The software provided with the SDKs is designed to enable you to rapidly develop network processor-based products. The software consists of three groups:

1. Developer’s Workbench. The Developer Workbench enables you to write symbolic microcode, assemble and optimize, build chip configurations, simulate and debug, all from a user-friendly windows interface.
2. Example Designs. The Example Design microcode is sectioned with generic scheduling and queueing mechanisms, enabling you to easily replace or extend the example protocols with your value-added protocols.
3. Libraries. The libraries are modular building blocks that you can use out of the box to construct Intel XScale® core core portions of your code. These building blocks include a Debug library, a uCode Loader, Serial or Ethernet remote access libraries, system messaging libraries, a variety of device drivers, and OS-specific board support packages.

Figure 1-1 shows the overall layout of the MEv2 Development Environment. The upper blocks reflect the Windows 2000* Host and the lower blocks reflect the network processor target system, which may be either VxWorks or Linux. The Host is a Win32 system, and the MEv2 Target systems may be either VxWorks or Embedded Linux based. All of the functionality applies to both VxWorks and eLinux.

**Figure 1-1. Development System Software Overview**



# Core Software API 2

## 2.1 Microcode Loader

The Microcode Loader is used to load microcode images, created by the microcode linker ucld, to the appropriate Microengines. Applications executing on the network processor consist of two distinct parts: one consisting of Microengine images and another of Intel XScale® core compiled code linked against the Microcode Loader Library (uclo.a). This section describes the Microcode Loader and its Application Program Interfaces (API).

### 2.1.1 UcLo - Microcode Loader Library

UcLo is a library of C functions that facilitates the loading of Microengine images, the management of Microengine external variables and functions, and provides the communication link between the Intel XScale® core application and the Microengines.

Release Components include:

- `uclo.h`—the Microcode Loader C header file.
- `uclo.a`, `uclo.lib`—the Microcode Loader C library file.

The `UcLo` API is summarized in Table 2-1.

### 2.1.2 API Summary

**Table 2-1. API Summary**

| API | Function |
|---|---|
| UcLo_InitLib() | Initializes the Loader library. |
| UcLo_InitLibUeng() | Initializes the Loader library. Microengines whose corresponding bit is set in `meMask` are set to the powerup default state. |
| UcLo_LoadObjFile() | Loads microcode object from a file to core memory. |
| UcLo_CopyObjFile() | Copies microcode object from a file to buffer. |
| UcLo_MapObjAddr() | Maps core memory location to microcode object. |
| UcLo_DeleObj() | Removes all references to a microcode object. |
| UcLo_BindSymbol() | Creates an association between a core application value and a microcode symbol. |
| UcLo_MeBindSymbol() | Binds a value to the microcode symbol in the UOF image.. |
| UcLo_WriteUimage() | Writes a specific microcode image to its assigned Microengine. |
| UcLo_WriteUimageAll() | Writes all microcode images to the appropriate Microengines. |
| UcLo_WriteUimagePage() | Writes a page of a specific microcode image to its assigned microengine. |
| UcLo_VerifyFile() | Verifies that the file is an UOF and that it is in the correct byte order. |

**Table 2-1. API Summary**

| API | Function |
|---|---|
| UcLo_VerifyUengine() | Compares the Microengine contents to its assigned image. |
| UcLo_VerifyUimage() | Compares a specific microcode image to the image in the Microengine. |
| UcLo_WriteUword() | Writes a microword to a specific Microengine. |
| UcLo_ReadUword() | Reads a microword from a specific Microengine. |
| UcLo_GetVarMemSegs() | Returns the variable memory segments. |
| UcLo_GetChecksum() | Returns the specified object's CRC checksum. |
| UcLo_GetAssignedMEs | Returns the mask specifying the microengines that are assigned to an image in the UOF. |
| UcLo_GetDebugInfo | Retrieve the udbug information from debug section of an uof. |
| UcLo_LoadIvdFile | Reads the specified file and extract information to be used to initialize import-variables. |
| UcLo_LoafIvdBuf | Same as UcLo_LoadIvdFile but extracts the information from a buffer rather than from a file. |
| UcLo_DelIvd | Remove all the information and resources that were created as a result of calls to UcLo_LoadIvdFile, or UcLo_LoadIvdBuf. |

## 2.1.3 API Functions

### 2.1.3.1 UcLo_InitLib()

This function initializes the uclo library and performs any Microengine driver initialization. It should be called prior to calling any of the other functions in the uclo library.

| | | |
|---|---|---|
| C Syntax | int UcLo_InitLib(void) | |
| Parameters | None | |
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_FILEFAIL | File operation failed due to file related error. |

### 2.1.3.2 UcLo_InitLibUeng()

Initializes the Loader library. Microengines whose corresponding bit is set in `meMask` are set to the powerup default state. The library assumes that those microengines that are not specified are in a reset state.

| | | |
|---|---|---|
| C Syntax | void UcLo_InitLibUeng(unsigned int meMask); | |
| Parameters | meMask [IN/ - A void pointer reference to the loaded object. | |
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_FILEFAIL | File operation failed due to file related error. |

### 2.1.3.3 UcLo_LoadObjFile()

This function loads an object file that was created by the microcode linker ucld to Intel XScale® core memory. The library will allocate and manage the resources to accommodate the object. The user should call UcLo_DeleObj() to remove reference to the object and to free the resources that was allocated by the library.

| | |
|---|---|
| C Syntax | int UcLo_LoadObjFile(void **handle, const char *fileName) |
| Parameters | handle [OUT] - A void pointer reference to the loaded object.<br>fileName [IN] - Character string pointer to the name of the image file that was created by ucld. |

Returns

| | |
|---|---|
| UCLO_SUCCESS | Operation was successful. |
| UCLO_FILEFAIL | File operation failed due to file related error. |
| UCLO_MEMFAIL | Failed allocate memory. |
| UCLO_BADARG | Invalid argument passed to function. |
| UCLO_BADOBJ | Error in object format or checksum. |
| UCLO_UOFVERINCOMPAT | The UOF is incompatible with this version of the Loader library. |
| UCLO_UOFINCOMPAT | The UOF is incompatible with this version of the chip. |

### 2.1.3.4 UcLo_CopyObjFile()

Loads the UOF objects from the file specified by fileName to a buffer. If UCLO_SUCCESS is returned, then a buffer will be allocated; its pointer and size will be returned in objBuf and chunkSize respectively. It is the responsibility of the caller to delete the buffer.

| | |
|---|---|
| C Syntax | int UcLo_CopyObjFile(const char *fileName, char **objBuf, unsigned int *objSize, unsigned int * chunkSize) |
| Parameters | fileName [IN] - Character string pointer to the name of the image file that was created by ucld.<br>objBuf [OUT] - A char pointer to accept the pointer to the object.<br>chunkSize [OUT] - The size of the buffer pointed to by objBuf. |

Returns

| | |
|---|---|
| UCLO_SUCCESS | Operation was successful. |
| UCLO_FILEFAIL | File operation failed due to file related error. |
| UCLO_BADARG | Invalid argument passed to function. |

### 2.1.3.5 UcLo_MapObjAddr()

This function maps the memory location pointed to by the addrPtr parameter to the Ucode Object File (UOF). The library will verify that the mapped object fits within the size specified by the memSize parameter and that the object checksum is valid. If the readOnly parameter is not equal to zero, then the library will copy any region of memory that it needs to modify. Also, the memory region, (addrPtr through addrPtr + memSize + 1), should not be modified by the caller while the image is mapped to the region -- unless through the use of uclo library functions. The user should call UcLo_DeleObj() to remove reference to the object and to free the resources that was allocated by the library.

| | |
|---|---|
| C Syntax | int UcLo_MapObjAddr(void **handle, void *addrPtr, const int memSize, int readOnly) |
| Parameters | handle [OUT] - A void pointer reference to the mapped object.<br>addrPtr [IN] - Pointer to the memory location where the Ucode Object File image resides.<br>memSize [IN] - An integer indicating the size of the memory region pointed to by the addrPtr parameter.<br>readOnly [IN] - Indicates whether the memory space being pointed to by addrPtr is read-only. |

| Returns | | |
|---|---|---|
| | UCLO_SUCCESS | Operation was successful. |
| | UCLO_BADOBJ | Error in object format or checksum. |
| | UCLO_MEMFAIL | Failed allocate memory. |
| | UCLO_BADARG | Invalid argument specified. |
| | UCLO_UOFVERINCOMPAT | The UOF is incompatible with this version of the Loader library. |
| | UCLO_UOFINCOMPAT | The UOF is incompatible with this version of the chip. |

### 2.1.3.6 UcLo_DeleObj()

This function removes all references to the UOF image -- if one was either loaded or mapped. The memory region of mapped UOF, by calling UcLo_MapImageAddr, will not be deallocated by this function and it is responsible of the caller to explicitly delete it.

| | |
|---|---|
| C Syntax | int UcLo_DeleObj(void *handle) |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object. |

| Returns | | |
|---|---|---|
| | UCLO_SUCCESS | Operation was successful. |
| | UCLO_NOOBJ | No object was mapped or loaded. |

### 2.1.3.7 UcLo_BindSymbol()

This function creates an association between a core application value and a microcode symbol. It initializes all occurrences of the specified symbol in the UOF image to the 32-bit value, or portion of the 32-bit value, as defined by the ucode assembler (uca). See the Assembler, and Linker sections for more information on these tools.

| | |
|---|---|
| C Syntax | int UcLo_BindSymbol(void *handle, const char *ucodeImageName, const char *ucodeSymName, const int value) |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>ucodeImageName [IN] - Pointer to a character string containing the name of the Microengine image.<br>ucodeSymName [IN] - String pointer to the name of the microcode symbol to bind.<br>value [IN] - An unsigned 32-bit value of which to initialize the microcode symbols. |

| Returns | | |
|---|---|---|
| | UCLO_SUCCESS | Operation was successful. |
| | UCLO_BADARG | Invalid argument(s) specified. |

| | |
|---|---|
| UCLO_NOOBJ | No object was either loaded or mapped. |
| UCLO_IMGNOTFND | Image not found. |
| UCLO_SYMNOTFND | Symbol not found. |

### 2.1.3.8 UcLo_MeBindSymbol()

Binds a value to the microcode symbol in the UOF image. Because multiple microengines may be assigned to the same UOF image, the result of this function applies to the specified microengine and to all assigned microengines.

C Syntax: nt UcLo_MeBindSymbol(void *handle, unsigned char me, char *ucodeSymName, int value);

Parameters:
handle [IN] - A void pointer reference to the loaded and mapped object.
me[IN] - Specifies a microengine.
ucodeSymName [IN] - String pointer to the name of the microcode symbol to bind.
value [IN] - An unsigned 32-bit value of which to initialize the microcode symbols.

Returns:

| | |
|---|---|
| UCLO_SUCCESS | Operation was successful. |
| UCLO_BADARG | Invalid argument(s) specified. |
| UCLO_NOOBJ | No object was either loaded or mapped. |
| UCLO_IMGNOTFND | Image not found. |
| UCLO_SYMNOTFND | Symbol not found. |

### 2.1.3.9 UcLo_WriteUimage()

Writes the specified image in the UOF object to the assigned microengines.

C Syntax: int UcLo_WriteUimage (void *handle, const char *ucodeImageName)

Parameters:
handle [IN] - A void pointer reference to the loaded/mapped object.
ucodeImageName [IN] - Pointer to character string containing name of the Microengine image.

Returns:

| | |
|---|---|
| UCLO_SUCCESS | Operation was successful. |
| UCLO_NOOBJ | No object was either loaded or mapped. |
| UCLO_IMGNOTFND | Image not found. |
| UCLO_FAILURE | The operation failed. |
| UCLO_BADARG | Invalid argument specified. |
| UCLO_UNINITVAR | A variable was not initialized. |

### 2.1.3.10 UcLo_WriteUimageAll()

This function writes all the microcode images to one or more appropriate Microengine(s).

C Syntax: int UcLo_WriteUimageAll(void *handle)

Parameters:
handle [IN] - A void pointer reference to the loaded/mapped object.

| | | |
|---|---|---|
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_NOOBJ | No object was either loaded or mapped. |
| | UCLO_IMGNOTFND | Image not found. |
| | UCLO_FAILURE | The operation failed. |
| | UCLO_BADARG | Invalid argument specified. |
| | UCLO_UNINITVAR | A variable was not initialized. |

### 2.1.3.11 UcLo_WriteUimagePage()

Writes the specified page in the UOF object to the assigned microengines.

C Syntax int UcLo_WriteUimagePage (void *handle, const char *ucodeImageName, char pageNum)

Parameters handle [IN] - A void pointer reference to the loaded/mapped object.
ucodeImageName [IN] - Pointer to character string containing name of the Microengine image.
pageNum [IN] - Byte containing the page number to be loaded. Pages are numbered 0 through 255.

| | | |
|---|---|---|
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_NOOBJ | No object was either loaded or mapped. |
| | UCLO_IMGNOTFND | Image not found. |
| | UCLO_FAILURE | The operation failed. |
| | UCLO_BADARG | Invalid argument specified. |
| | UCLO_UNINITVAR | A variable was not initialized. |

### 2.1.3.12 UcLo_VerifyFile()

First, the function verifies that the file is an UOF. If it is, then the UOF format version of the file is returned in the minVer and majVer arguments, and the convEndian argument indicates whether the file is in the correct byte order for the specific system.

C Syntax int UcLo_VerifyFile (char *fileHdr, short *minVer, short *majVer, int * convEndian)

Parameters fileHdr [IN] - A handle to the file.
minVer [IN] - The UOF format minor version.
majVer [IN] - The UOF format major version.
convEndian[IN] - Indicates whether the file is in the correct byte order for the system.

| | | |
|---|---|---|
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_FAILURE | The operation failed. |
| | UCLO_UOFINCOMPAT | The UOF is incompatible with this version of the Loader library. |

### 2.1.3.13 UcLo_VerifyUengine()

Compares the content of the specified Microengine to its assigned UOF image that was either mapped or loaded

| | | |
|---|---|---|
| C Syntax | int UcLo_VerifyUengine (void *handle, unsigned char me) | |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>me [IN] - Specifies a microengine. | |
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_NOOBJ | No object was either loaded or mapped. |
| | UCLO_IMGNOTFND | Image not found. |
| | UCLO_FAILURE | The operation failed. |
| | UCLO_BADARG | Invalid argument specified. |

### 2.1.3.14 UcLo_VerifyUimage()

Compares the UOF image that was either loaded or mapped to the content of the assigned Microengine(s).

| | | |
|---|---|---|
| C Syntax | int UcLo_VerifyUimage (void *handle, const char *ucodeImageName) | |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>ucodeImageName [IN] - Pointer to character string containing name of the Microengine image. | |
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_NOOBJ | No object was either loaded or mapped. |
| | UCLO_IMGNOTFND | Image not found. |
| | UCLO_FAILURE | The operation failed. |

### 2.1.3.15 UcLo_WriteUword()

Writes a 32-bit unsigned value to the specified Microengine(s) and micro address.

| | | |
|---|---|---|
| C Syntax | int UcLo_WriteUword (void *handle, unsigned int meMask, unsigned int uAddr, uword_T uWord); | |
| Parameters | Handle [IN] - A void pointer reference to the loaded and mapped object..<br>meMask[IN] - An integer mask specifying the microengines to which to write the value uWord at the address uAddr.<br>uAddr[IN] - An integer value indicating the microstore address to write the value to.<br>uWord [IN] - The value to be written to one or more microengines. | |
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_BADARG | Invalid argument specified. |
| | UCLO_FAILURE | Operation failed. |

### 2.1.3.16 UcLo_ReadUword()

Reads a unsigned integer value from the specified Microengine and microstore address.

| | |
|---|---|
| C Syntax | int UcLo_ReadUword (void *handle, unsigned int meMask, unsigned int uAddr, uword_T uWord); |
| Parameters | Handle [IN] - A void pointer reference to the loaded and mapped object..<br>meMask[IN] - An integer mask specifying one or more microengines.<br>uAddr[IN] - An integer value indicating the microstore address from which to read a value.<br>uWord [OUT] - The value read from the specified microengines at address uAddr. |

| | | |
|---|---|---|
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_BADARG | Invalid argument specified. |
| | UCLO_FAILURE | Operation failed. |

### 2.1.3.17 UcLo_GetVarMemSegs()

Retrieve the locations and sizes of the uC compiler variables memory segments.

| | |
|---|---|
| C Syntax | int UcLo_GetVarMemSegs(void *handle, UcLo_VarMemSeg_T *varMemSeg); |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>varMemSeg [OUT] - A pointer to variable segment structure. |

| | | |
|---|---|---|
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_BADARG | Invalid argument specified. |
| | UCLO_NOOBJ | No object was either loaded or mapped. |

#### 2.1.3.17.1 UcLo_VarMemSeg_S

The UcLo_VarMemSeg_S data structure is used by UcLo_GetVarMemSegs()to return a view of Microengine C compiler variable memory segments.

C Syntax

```
typedef struct UcLo_VarMemSeg_S{
       unsigned int sram0Base;
       unsigned int sram0Size;
        unsigned int sram1Base;
        unsigned int sram1Size;
        unsigned int sram2Base;/
        unsigned int sram2Size;
        unsigned int sram3Base;/
        unsigned int sram3Size;
        unsigned int sdramBase;
       unsigned int sdramSize;/
        unsigned int sdram1Base;/
        unsigned int sdram1Size;
        unsigned int scratchBase;
        unsigned int scratchSize;
}UcLo_VarMemSeg_T;
```

| | |
|---|---|
| Data members | sram0Base -- SRAM0 memory segment base address<br>sram0Size -- SRAM0 segment size bytes<br>sram1Base -- SRAM1 memory segment base address<br>sram1Size -- SRAM1 segment size bytes<br>sram2Base --SRAM2 memory segment base address<br>sram2Size -- SRAM2 segment size bytes<br>sram3Base -- SRAM3 memory segment base address<br>sram3Size -- SRAM3 segment size bytes<br>sdramBase -- SDRAM memory segment base address<br>sdramSize -- SDRAM segment size bytes<br>sdram1Base --SDRAM1 memory segment base address<br>sdram1Size -- SDRAM1 segment size bytes<br>scratchBase -- SCRATCH memory segment base address<br>scratchSize -- SCRATCH segment size bytes |

### 2.1.3.18 UcLo_GetCheckSum()

Returns the specified object's CRC checksum.

| | | |
|---|---|---|
| C Syntax | int UcLo_GetChecksum(void *handle, unsigned int *checksum); | |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>checksum [OUT] - The specified object's checksum. | |
| Returns | UCLO_SUCCESS | Operation was successful. |
| | UCLO_BADARG | Invalid argument specified. |
| | UCLO_NOOBJ | No object was either loaded or mapped. |

### 2.1.3.19 UcLo_GetAssignedMEs()

Returns the mask specifying the microengines that are assigned to an image in the UOF.

| | |
|---|---|
| C Syntax | unsigned int UcLo_GetAssignedMEs(void *handle); |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object. |
| Returns | The mask of the assigned microengines. |

### 2.1.3.20 UcLo_GetDebugInfo

Retrieve the udbug information from debug section of an uof. If the UOF was linked with debug information, then memory will allocated and the information will be extracted, and a pointer will be returned in *image* parameter. It's the caller responsibility to delete the debug information. The dbgMeInfo_Image_T is described in dbgMeInfo.h.

| | |
|---|---|
| C Syntax | int UcLo_GetDebugInfo(void *handle, dbgMeInfo_Image_T *image); |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>checksum [OUT] - The specified object's checksum. |
| Returns | UCLO_SUCCESS, UCLO_FAILURE, UCLO_BADARG, UCLO_NOOBJ |

### 2.1.3.21 UcLo_LoadIvdFile

Reads the specified file and extract information to be used to initialize import-variables. The file format is the following space-separated values:

<chipName> <imageName> <symbolName> <value>

Where:

| | |
|---|---|
| <chipName> | Is the name of the chip. This value is only used in the simulation mode and can be an empty string. |
| <imageName> | Is the name of the UOF image -- typically the name of the list-file unless implicitly defined by the .image_name directive. |
| <symbolName> | Is the name of the import-variable symbol to be initialized. |
| <value> | Is the value to be assigned to the import-variable. |
| C Syntax | int UcLo_LoadIvdFile(char *filename); |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>checksum [OUT] - The specified object's checksum. |
| Returns | UCLO_SUCCESS, UCLO_FAILURE |

### 2.1.3.22 UcLo_LoadIvdBuf

Same as UcLo_LoadIvdFile but extracts the information from a buffer rather than from a file.

| | |
|---|---|
| C Syntax | int UcLo_LoadIvdBuf(char *ivdBuf, unsigned int ivdBufLen); |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>checksum [OUT] - The specified object's checksum. |
| Returns | UCLO_SUCCESS, UCLO_FAILURE |

### 2.1.3.23 UcLo_DelIvd

Remove all the information and resources that were created as a result of calls to UcLo_LoadIvdFile, or UcLo_LoadIvdBuf.

| | |
|---|---|
| C Syntax | void UcLo_DelIvd(); |
| Parameters | handle [IN] - A void pointer reference to the loaded/mapped object.<br>checksum [OUT] - The specified object's checksum. |
| Returns | none |

## 2.2 Hardware Abstraction Layer (HAL)

The Intel XScale® core and the microengines share access to the same Intel® IXP2400 and IXP2800 functional units—SRAM, SDRAM and the FBI (IX Bus). An Intel XScale® core application interfaces to these units in one of three ways:

- Through memory-mapped locations in the XScale™ core address space when running on the hardware
- Using the simIO and XACT API calls to the Transactor when running as a simulation
- Using the APIs provided by the Hardware Abstraction Layer, known as HAL

The HAL generates code to interface either to the Intel XScale® core hardware or the Transactor. An XScale™ core application that uses HAL can run in hardware mode or simulation mode without changes to the code that accesses the functional units. This increases portability of the application code when moving between the simulation environment of the Transactor and the XScale™ core hardware.

This section describes the HAL API calls. The calls for each of the shared Intel® IXP2400 and IXP2800 functional units are partitioned into the following categories:

- Instantiation function
- Memory access functions
- CSR and atomic functions

The instantiation function returns a pointer to an instance of the functional unit class and is used as a reference to other functional unit API calls.

Memory access functions are used to read and write to functional unit memory—including SDRAM, SRAM, and scratchpad memory. The memory access functions—through the definition of the `IOSTYLE` compiler directive—map to the correct underlying operation.

Hardware memory functions are available only when `IOSTYLE` is defined as either `HARDWARE` or `XACTIO`. These functions include access to control status registers (CSR) as well as advanced memory operations such as SRAM push and pop.

The Hardware Abstraction Layer provides an abstract programming layer between the tools/application and the IXP microengines. The Intel® XScale™ core and microengines share access to the same IXP functional units—microengines, SRAM and DRAM memory, media switch fabric, and so on. These HAL libraries provide the Microengine Debug Library and Intel XScale® core applications access to the IXP functional modules regardless of whether they are running on the hardware or on the simulator.

The microengine HAL consists of macros and C-language APIs that are defined in `hal_mev2.h` and `halMev2Api.h`. The macros provide access to the microengine CSRs without any error checking of the parameters. The C-language APIs utilize the macros to provide microengine CSR access and provide parameter other error checking. Both the macros and the C-language functions allow seamless access either on the hardware or simulator.

## 2.3 HAL Register Macros

Table 2-2 shows the macros used for accessing microengine control status registers (CSR) and transfer registers (XFER). The `GET_ME_xxx` macros return the value of the register while the `SET_ME_xxx` ones assigns a value to the register.

**Table 2-2. HAL Microengine Register API**

| Name | Description |
|---|---|
| SET_ME_CSR() | Writes a value to a microengine CSR. |
| GET_ME_CSR() | Reads a microengine CSR. |
| SET_ME_SXFER() | Writes a value to a microengine SRAM XFER register. |
| GET_ME_SXFER() | Reads a microengine SRAM XFER register. |
| SET_ME_DXFER() | Writes a value to a microengine DRAM XFER register. |
| GET_ME_DXFER() | Reads a microengine DRAM XFER register. |

### 2.3.1 API Macros

#### 2.3.1.1 `SET_ME_CSR()`

Writes a value to a microengine CSR.

##### Microengine Assembler Syntax

```
SET_ME_CSR(me, csr, val);
```

##### Input

| | |
|---|---|
| `me` | Specifies the microengine whose CSR is to set. |
| `csr` | The offset of the CSR relative to the base of the microengine CSR address. |
| `val` | An unsigned value to be written to the register. |

##### Microengine Assembler Usage Example

```
SET_ME_CSR(me, csr, val);
```

#### 2.3.1.2 `GET_ME_CSR()`

Reads a microengine CSR.

#### Microengine Assembler Syntax

```
GET_ME_CSR(me, csr);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine whose CSR is to read. |
| `csr` | The offset of the CSR relative to the base of the microengine CSR address. |

#### Microengine Assembler Usage Example

```
csrValue = GET_ME_CSR(meNum, csrOffset);
```

### 2.3.1.3 `SET_ME_SXFER()`

Writes a value to a microengine SRAM XFER register.

#### Microengine Assembler Syntax

```
SET_ME_SXFER(me, reg, val);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine whose SRAM XFER register is to set. |
| `reg` | The microengine-relative register number. |
| `val` | An unsigned value to be written to the register. |

#### Microengine Assembler Usage Example

```
SET_ME_SXFER(me, reg, val);
```

### 2.3.1.4 `GET_ME_SXFER()`

Reads a microengine SRAM XFER register.

#### Microengine Assembler Syntax

```
GET_ME_SXFER(me, reg);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine whose SRAM XFER register is to read. |
| `reg` | The microengine-relative register number. |

### Microengine Assembler Usage Example

```
xferRegValue = GET_ME_SXFER(me, reg);
```

## 2.3.1.5 `SET_ME_DXFER()`

Writes a value to a microengine DRAM XFER register.

### Microengine Assembler Syntax

```
SET_ME_DXFER(me, reg, val);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine whose DRAM XFER register is to set. |
| `reg` | The microengine-relative register number. |
| `val` | An unsigned value to be written to the register. |

### Microengine Assembler Usage Example

```
SET_ME_DXFER(me, reg, val);
```

## 2.3.1.6 `GET_ME_DXFER()`

Reads a microengine DRAM XFER register.

### Microengine Assembler Syntax

```
GET_ME_DXFER(me, reg);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine whose DRAM XFER register is to read. |
| `reg` | The microengine-relative register number. |

### Microengine Assembler Usage Example

```
xferRegValue = GET_ME_DXFER(me, reg);
```

## 2.4 HAL Microengine C Functions

***Note:*** The HAL physical address mapping has changed from a constant; therefore, simIo_Init then halMe_Init must be called prior to using any of the HAL macros or APIs, and your application/foreign-model linked with the halMev2, utils.lib, ossl.lib, and simio_lib.lib libraries. If you are linking with loader_dll.lib, then the simio_lib.lib must precede it. Also, your foreign-model code must be compiled as “Multithreaded DLL” (see Visual Studio setting “Project Settings->C/C++->Category:Code Generation, Use run-time library”) in order to link properly.

**Table 2-3. `halME` API**

| Name | Description |
|---|---|
| halMe_Init() | Initializes the microengines and take them out of reset. |
| halMe_DelLib() | Restores the system resources allocated by the halMe_Init function. |
| halMe_IntrEnable() | Enables breakpoint, parity, or thread interrupts. |
| halMe_IntrDisable() | Disables breakpoint, parity, or thread interrupts. |
| halMe_SpawnIntrCallbackThd() | Invokes specified callback function for subsequent occurrences of interrupts without blocking. |
| HalMeIntrCallback() | Defines an interrupt callback function. |
| halMe_TerminateCallbackThd() | Cancels a request initiated by the halMe_SpawnIntrCallbackThd() function. |
| halMe_GetMeState() | Determines if the specified microengine is valid, whether it is initialized, or whether it is in or out of reset. |
| halMe_IntrPoll() | Waits for one or more interrupts to occur. |
| halMe_GetMeState() | Determines if the specified microengines is valid or whether it’s initialized, in or out of reset. |
| halMe_SetUstoreFreeMem() | Defines a region of micro-store that’s unused. |
| halMe_GetUstoreFreeMem() | Retrieves the starting address and size of the micro-store free region. |
| halMe_IsMeEnabled() | Determines if the specified microengine is active. |
| halMe_Start() | Sets the wake-event voluntary and start the microengines context, specified by the me and `startCtx` parameters, from it’s current PC. |
| halMe_Stop() | Stop the microengines contexts that have a corresponding bit set in the `ctxMask` parameter. |
| halMe_Reset() | Resets the specified microengines. |
| halMe_ClrReset() | Takes the specified micro-engines out of the reset state. |
| halMe_ResetTimestamp() | Stops the timestammp clock and zeroes the timestamps of all specified microengines, then restarts the timestamp clock. |
| halMe_GetCtxArb() | Reads the microengines context arbitration-control register and return the value in the `ctxArbCtl` parameter. |
| halMe_PutCtxArb() | Writes a long-word ctxArbCtl value to the microengines context arbitration-control register. |
| halMe_PutCtxStatus() | Writes the context status CSR for specified contexts. |
| halMe_GetCtxStatus() | Reads the context status CSR for specified contexts. |
| halMe_GetMeCsr() | Reads the microengines CSR indicated by `csr` and return the value in the value parameter. |
| halMe_PutMeCsr() | Writes the long-word value to microengines CSR indicated by csr parameter. |
| halMe_PutMeLmMode() | Sets the local-memory mode to relative or global. |
| halMe_PutMeCtxMode() | Sets the microengines context-mode to either four or eight. |

**Table 2-3. `halME` API (Continued)**

| Name | Description |
|---|---|
| halMe_PutMeNnMode() | Sets the microengines next-neighbor mode to either write to itself (mode=0), or to its neighbor (mode=1). |
| halMe_GetCtxWakeupEvents() | Reads the state of the microengines context wakeup-event signals. |
| halMe_PutCtxWakeupEvents() | Writes the long-word events to the microengines context-wakeup-events CSR. |
| halMe_GetCtxSigEvents() | Reads the signal-events of specified microengines context. |
| halMe_PutCtxSigEvents() | Writes the long-word value of the event parameter to the signal-events of the specified microengines contexts. |
| halMe_PutPC() | Sets the indicated microengines context program-counters to the value specified by the `upc` parameter. |
| halMe_GetPC() | Reads the program-counter of the specified microengines context. |
| halMe_PutUwords() | Writes a number of uwords to the specified microengines. |
| halMe_GetUwords() | Reads a number of uwords from specified microengines micro-store. |
| halMe_PutRelDataReg() | Writes a longword to the relative microengine general purpose register. |
| halMe_GetRelDataReg() | Reads a longword to the relative microengine general purpose register. |
| halMe_PutAbsDataReg() | Writes a longword to the absolutemicroengine general purpose register. |
| halMe_GetAbsDataReg() | Reads a longword to the absolute microengine general purpose register. |
| halMe_PutLM() | Writes a long-word value to the specified microengines's local-memory. |
| halMe_GetLM() | Reads a long-word value from the specified microengines local-memory. |
| halMe_VerifyMe() | Checks that the value specified by the `me` parameter is a valid microengine number. |
| halMe_VerifyMeMask() | Checks that the value specified by the `meMask` parameter indicates valid microengine number. |
| halMe_SysMemVirtSize() | Returns the virtual memory region sizes. |
| halMe_GetVirAddr() | Translates an NPU physical address to a virtual address. |
| halMe_PutCAM() | Writes the longword data to the specified microengine CAM entry and mark it as the MRU (Most Recently Used). |
| halMe_GetCAMState() | Reads the contents of the microengine CAM and store the tag value, status in the appropriate parameters. |
| halMe_IsInpStateSet() | Determines if the microengine is in the state specified by `inpState`. |
| halMe_PutTbuf() | Writes a longword from value to `TBUF` starting at the address location specified by `tbufByte`. |
| halMe_GetRbuf()f | Reads a longword from the `RBUF` location and store it. |
| halMe_PutTbufEleCntl() | Writes the `eleCtlA`, and `eleCtlB` long-words to the `TBUF` element control specified by `eleNum`. |
| halMe_GetVirXaddr() | Return the starting virtual address of the spcified physical region. |
| halMe_GetPhyXaddr | Return the starting physical address of the spcified virtual region . |
| halMe_PutCtxIndrCsr() | Write a 32-bit value to the micro-engine context(s) indirect CSR. |
| halMe_GetCtxIndrCsr() | Read a micro-engine context indirect CSR. |

## 2.4.1 Defined Types, Enumerations, and Data Structures

### 2.4.1.1 `Hal_IntrMasks_T`

The data structure used to specify an interrupt mask.

#### C Syntax

```
typedef struct {
   unsigned int attn_bkpt_mask;
   unsigned int attn_parity_mask;
   unsigned int thd_a_mask[4];
   unsigned int thd_b_mask[4];
} Hal_IntrMasks_T;
```

#### Members

| | |
|---|---|
| attn_bkpt_mask | Mask of the microengines breakpoint interrupts. |
| attn_parity_mask | Mask of the microengines parity interrupts. |
| thd_a_mask | Mask of threads. |
| thd_b_mask | Mask of threads. |

### 2.4.1.2 `Hal_Me_CSR_T`

This enumeration lists the microengine control and status register (CSR) addresses—as described in the network processor's Programmer's Reference Manual. These values are used to describe the register being acted upon by CSR related functions and macros.

#### C Syntax

```
typedef enum{
   USTORE_ADDRESS                 = 0x000,
   USTORE_DATA_LOWER              = 0x004,
   USTORE_DATA_UPPER              = 0x008,
   USTORE_ERROR_STATUS            = 0x00c,
   ALU_OUT                        = 0x010,
   CTX_ARB_CNTL                   = 0x014,
   CTX_ENABLES                    = 0x018,
   CC_ENABLE                      = 0x01c,
   CSR_CTX_POINTER                = 0x020,
   CTX_STS_INDIRECT               = 0x040,
   ACTIVE_CTX_STATUS              = 0x044,
   CTX_SIG_EVENTS_INDIRECT        = 0x048,
   CTX_SIG_EVENTS_ACTIVE          = 0x04c,
   CTX_WAKEUP_EVENTS_INDIRECT     = 0x050,
   CTX_WAKEUP_EVENTS_ACTIVE       = 0x054,
   CTX_FUTURE_NUM_INDIRECT        = 0x058,
   CTX_FUTURE_NUM_ACTIVE          = 0x05c,
    LM_ADDR_0_INDIRECT              = 0x060,
    LM_ADDR_0_ACTIVE                = 0x064,
    LM_ADDR_1_INDIRECT              = 0x068,
    LM_ADDR_1_ACTIVE                = 0x06c,
    BYTE_INDEX                      = 0x070,
    ACTIVE_LM_ADDR_0_BYTE_INDEX     = 0x0e0,
    INDIRECT_LM_ADDR_0_BYTE_INDEX   = 0x0e4,
    INDIRECT_LM_ADDR_1_BYTE_INDEX   = 0x0e8,
    ACTIVE_LM_ADDR_1_BYTE_INDEX     = 0x0ec,
    T_INDEX_BYTE_INDEX              = 0x0f4,
    T_INDEX                         = 0x074,
    FUTURE_NUM_SIGNAL_INDIRECT      = 0x078,
    FUTURE_NUM_SIGNAL_ACTIVE        = 0x07c,
    NN_PUT                          = 0x080,
    NN_GET                          = 0x084,
    TIMESTAMP_LOW                   = 0x0c0,
    TIMESTAMP_HIGH                  = 0x0c4,
    INSTRUCTION_SIGNATURE           = 0x0c8,
    NEXT_NEIGHBOR_SIGNAL            = 0x100,
    PREV_NEIGHBOR_SIGNAL            = 0x104,
    SAME_ME_SIGNAL                  = 0x108,
    CRC_REMAINDER                   = 0x140,
    PROFILE_COUNT                   = 0x144,
    PSEUDO_RANDOM_NUMBER            = 0x148,
    LOCAL_CSR_STATUS                = 0x180,
    NULL_CSR                        = 0x3fc
}Hal_Me_CSR_T;
```

#### 2.4.1.3 Register Types

The following symbolic constants are defined to specifiy register types.

| Symbolic Constant | Definition |
|---|---|
| IXP_NO_DEST | No register |
| IXP_GPA_REL | Bank-A Relative General Purpose Register |
| IXP_GPA_ABS | Bank-A Absolute General Purpose Register |
| IXP_GPB_REL | Bank-B Relative General Purpose Register |
| IXP_GPB_ABS | Bank-B Absolute General Purpose Register |
| IXP_SR_REL | SRAM Transfer Register |
| IXP_SR_RD_REL | Relative SRAM Read-Transfer Register |
| IXP_SR_WR_REL | Relative SRAM Write-Transfer Register |
| IXP_SR_ABS | SRAM Absolute Register |
| IXP_SR_RD_ABS | Absolute SRAM Read-Transfer Register |
| IXP_SR_WR_ABS | Absolute SRAM Write-Transfer Register |
| IXP_SR0_SPILL | SRAM Channel 0 Spill Register |
| IXP_SR1_SPILL | SRAM Channel 1 Spill Register |
| IXP_SR2_SPILL | SRAM Channel 2 Spill Register |

| Symbolic Constant | Definition |
|---|---|
| IXP_SR3_SPILL | SRAM Channel 3 Spill Register |
| IXP_SR0_MEM_ADDR | SRAM Channel 0 Memory Address |
| IXP_SR1_MEM_ADDR | SRAM Channel 1 Memory Address |
| IXP_SR2_MEM_ADDR | SRAM Channel 2 Memory Address |
| IXP_SR3_MEM_ADDR | SRAM Channel 3 Memory Addres |
| IXP_DR_REL | DRAM Transfer Register |
| IXP_DR_RD_REL | Relative DRAM Read-Transfer Register |
| IXP_DR_WR_REL | Relative DRAM Write-Transfer Register |
| IXP_DR_ABS | DRAM Absolute Register |
| IXP_DR_RD_ABS | Absolute DRAM Read-Transfer Register |
| IXP_DR_WR_ABS | Absolute DRAM Write-Transfer Register |
| IXP_DR_MEM_ADDR | DRAM Memory Address |
| IXP_LMEM | Local Memory Register |
| IXP_LMEM0 | Local Memory 0 Register |
| IXP_LMEM1 | Local Memory 1 Register |
| IXP_LMEM_SPILL | Local Memory Spill Register |
| IXP_LMEM_ADDR | Local Memory Address |
| IXP_NEIGH_REL | Relative Neighbor Register |
| IXP_NEIGH_INDX | Neighbor Index Register |
| IXP_SIG_REL | Relative Signal Register |
| IXP_SIG_INDX | Signal Index Register |
| IXP_SIG_DOUBLE | Double Signal Register |
| IXP_SIG_SINGLE | Single Signal Register |
| IXP_SCRATCH_MEM_ADDR | Scratch Memory Address |
| IXP_ANY_REG = 0xffff | Any of the registers in this table. |

## 2.4.2 API Functions

### 2.4.2.1 `halMe_Init()`

Initializes the microengines and takes them out of reset. Microengines with their corresponding bit set in `meMask` are initialized to the following state:

- Context mode is set to eight
- Program counters are set to zero
- Next context to run is set to zero
- All contexts are disabled
- `cc_enable` is set to `0x2000`

- Wakeup events are set to one
- The signal event is set to zero

All other microengines remain untouched—thought they are taken out of reset. This function should be called prior to calling most of the `halME` functions.

### C Syntax

```
int halMe_Init(uint meMask);
```

### Input

| | |
|---|---|
| `meMask` | The bits correspond to the microengine number and address of microengines to be initialized. For example, the first microengine in the second cluster corresponds to bit 16. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful |

## 2.4.2.2 `halMe_DelLib()`

Restores the system resources allocated by the halMe_Init() function. No arguments are required and a `void` value is returned.

### C Syntax

```
void halMe_DelLib(void);
```

## 2.4.2.3 `halMe_IntrEnable()`

Enables breakpoint, parity, or thread interrupts, and enables interrupt processing by the HAL. If the HAL interrupt processing is not enabled, then the application is free to manage the particular interrupts in an OS dependent manner..

### C Syntax

```
void halMe_IntrEnable(uint type_mask);
```

### Input

| | |
|---|---|
| `type_mask` | Specifies the mask type—must be one or more of the following values {`HALME_INTR_ATTN_BKPT_MASK`, `HALME_INTR_ATTN_PARITY_MASK`, `HALME_INTR_THD_A_MASK`, `HALME_INTR_THD_B_MASK`}. |

### 2.4.2.4 halMe_IntrDisable()

Disables breakpoint, parity, or thread interrupts.

#### C Syntax

```
void halMe_IntrDisable(uint type_mask);
```

#### Input

| | |
|---|---|
| `type_mask` | Specifies the mask type—must be one or more of the following values {`HALME_INTR_ATTN_BKPT_MASK`, `HALME_INTR_ATTN_PARITY_MASK`, `HALME_INTR_THD_A_MASK`, `HALME_INTR_THD_B_MASK`}. |

### 2.4.2.5 halMe_IntrPoll()

Waits until one or more of the interrupts specified by `type_mask` occurs, or returns immediately if there are any outstanding interrupts of the specified type. Interrupts must be enabled by halMe_IntrEnable() or else any attempt to poll returns `HALME_DISABLED`.

Upon return, the mask parameter contains a bit mask of the microengines or threads which have an interrupt of the type or types specified in the poll request. The masks may be zero in the case of a halMe_IntrDisable() request—presumably from a different thread. The interrupt is cleared before the call returns.

In the event that multiple threads were blocked on the same interrupt, when that interrupt occurs, only one poll call completes. There is no guarantee which of the calls completes.

#### C Syntax

```
int halMe_IntrPoll(uint type_mask, Hal_IntrMasks_T  *masks);
```

#### Input

| | |
|---|---|
| `type_mask` | Specifies the mask type—must be one or more of the following values {`HALME_INTR_ATTN_BKPT_MASK`, `HALME_INTR_ATTN_PARITY_MASK`, `HALME_INTR_THD_A_MASK`, `HALME_INTR_THD_B_MASK`}. |

#### Input/Output

| | |
|---|---|
| `masks` | On input specifies the location where the mask is to be returned. On output the returned mask.<br>See Hal_IntrMasks_T. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of {`HALME_SUCCESS`, `HALME_DISABLED`, `HALME_FAIL`}. |

## 2.4.2.6 `halMe_SpawnIntrCallbackThd()`

This functon is similar to halMe_IntrPoll()except that it does not block waiting for the specified interrupts. Instead, it returns immediately and invokes the specified callback function for subsequent occurrences of the interrupt or interupts. The callback data are passed to the callback function on invocation.

A handle associated with this call is returned—this handle can be used to terminate the request and disregard the specified interrupts.

### C Syntax

```
int halMe_SpawnIntrCallbackThd(
    uint type_mask,
    HalMeIntrCallback_T callback_func,
    void* callback_data,
    int thd_priority,
    void **handle);
```

### Input

| | |
|---|---|
| `type_mask` | Specifies the mask type—must be one or more of the following values {`HALME_INTR_ATTN_BKPT_MASK`, `HALME_INTR_ATTN_PARITY_MASK`, `HALME_INTR_THD_A_MASK`, `HALME_INTR_THD_B_MASK`}. |
| `callback_func` | A pointer to the location of a callback function. See HalMeIntrCallback(). |
| `callback_data` | A pointer to the location of application-specific data. This pointer is passed to the callback function when it is invoked. |
| `thd_priority` | The thread priority. |

### Output/Returns

| | |
|---|---|
| `handle` | A handle associated with and returned by this call. This handle can be used to terminate the request and disregard the specified interrupts. |
| Return Value | Returns one of {`HALME_SUCCESS`, `HALME_FAIL`}. |

#### 2.4.2.6.1 HalMeIntrCallback()

The function prototype for interrupt-callback functions.

##### C Syntax

```
typedef void (*HalMeIntrCallback_T)(Hal_IntrMasks_T *masks, void* data);
```

##### Input

| | |
|---|---|
| `masks` | Specifies the microengines or threads which have an interrupt of the type or types specified when the callback was registered.<br>See Hal_IntrMasks_T. |
| `data` | A pointer to the location of application-specific data. |

### 2.4.2.7 halMe_TerminateCallbackThd()

Cancels a request initiated by the halMe_SpawnIntrCallbackThd()function.

##### C Syntax

```
int halMe_TerminateCallbackThd(void* handle);
```

##### Input

| | |
|---|---|
| `handle` | The handle returned by a call to halMe_SpawnIntrCallbackThd(). This handle specifies which request to terminate. |

##### Output/Returns

| | |
|---|---|
| Return Value | Returns `HALME_SUCCESS`. |

### 2.4.2.8 halMe_GetMeState()

Determines if the specified microengine is valid, whether it is initialized, or whether it is in or out of reset.

##### C Syntax

```
int halMe_GetMeState(uchar me);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_UNINIT`–a bad debug library is associated with the specified microengine—the library is not initialized<br>• `HALME_CLR_RST`–the microengine is *not* in a reset state<br>• `HALME_RST`–the microengine is in reset state |

## 2.4.2.9 `halMe_SetUstoreFreeMem()`

Defines a region of microstore that is unused. The range defined by the `begFreeAddr` and `size` parameters must be between zero and 4095.

### C Syntax

```
int halMe_SetUstoreFreeMem(uchar me, uint begFreeAddr, uint size);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `begFreeAddr` | Specifies the microstore address where the free region begins. This value must be between zero and 4095. |
| `size` | Indicates the number of free microstore words. This value must be between zero and 4095. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.10 `halMe_GetUstoreFreeMem()`

Returns the starting address and size of the specified microengine microstore free region.

### C Syntax

```
int halMe_GetUstoreFreeMem(uchar me, uint *begFreeAddr, uint *size);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified |
| `begFreeAddr` | A pointer to the location of the beginning of the free region for the specified microengine microstore. |
| `size` | A pointer to the location of the size of the free region for the specified microengine microstore. |

## 2.4.2.11 `halMe_IsMeEnabled()`

Determines if the specified microengine is enabled to run and if any of its contexts are either running or waiting to run.

### C Syntax

```
int halMe_IsMeEnabled(uchar me);
```

### Input

| | |
|---|---|
| `me` | Specifies a microengine. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns of of {`HALME_ENABLED`, `HALME_DISABLED`, `HALME_BADLIB`, `HALME_RESET`}. |

### 2.4.2.12 `halMe_Start()`

Sets the wake-event to *voluntary* and starts the microengine context specified by the `me` and `startCtx` parameters from that microengine's current program counter. The `ctxEnMask` specifies the contexts to be enabled. If one of these microengines is in reset it is taken out of reset.

#### C Syntax

```
int halMe_Start(uchar me, uint ctxEnMask);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `ctxEnMask` | Specifies the contexts to be enabled. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |

### 2.4.2.13 `halMe_Stop()`

Stops the microengine contexts that have a corresponding bit set in the `ctxMask` parameter at the next context arbitration instruction. The context may not stop because it never executes a context-arbitration instruction. A value of `HALME_RESET` is returned if the microengine is in reset.

#### C Syntax

```
int halMe_Stop(uchar me, uint ctxMask);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `ctxEnMask` | Specifies the contexts to stop. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_RESET`–the microengine is in the reset state<br>• `HALME_FAIL`–the operation failed |

### 2.4.2.14 `halMe_Reset()`

Resets the specified microengines with a corresponding bit set in `meMask`. If `clrReg` is set then the microengines are initialized to the states described in halMe_Init().

#### C Syntax

```
void halMe_Reset(uint meMask, int clrReg);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `clrReg` | `If this parameter is set, then following register initialization is performed:`<br>• `CTX_ENABLES` are set to zero<br>• `CTX_ARB_CNTL` is set to zero<br>• `CC_ENABLE` is set to x2000<br>• All context program counters are set to zero<br>• `WAKEUP_EVENTS` are set to one<br>• `SIG_EVENTS` are set to zero |

### 2.4.2.15 `halMe_ClrReset()`

Takes the specified microengines out of the reset state.

#### C Syntax

```
void halMe_ClrReset(uint meMask);
```

#### Input

| | |
|---|---|
| `meMask` | Specifies one or more microengines to take out of the reset state. |

### 2.4.2.16 `halMe_ResetTimestamp()`

Stops the timestamp clock and zeroes the timestamps of all specified microengines then restarts the timestamp clock.

#### C Syntax

```
int halMe_ResetTimestamp(unsigned int meMask);
```

#### Input

| | |
|---|---|
| `meMask` | Specifies one or more microengines. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of {`HALME_BADLIB`, `HALME_SUCCESS`}. |

### 2.4.2.17 `halMe_GetCtxArb()`

Reads the microengine context arbitration control register and returns the value in the `ctxArbCtl` argument.

#### C Syntax

```
int halMe_GetCtxArb(uchar me, uint*ctxArbCtl);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |
| `ctxArbCtl` | A pointer to the location of the specified context arbitration control register. |

### 2.4.2.18 `halMe_PutCtxArb()`

Writes a long-word `ctxArbCtl` value to the microengine context arbitration control register for the specified microengine.

#### C Syntax

```
int halMe_PutCtxArb(uchar me, uint ctxArbCtl);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `ctxArbCtl` | The new value for the context arbitration control register. |

#### Output/Returns

Return Value

Returns one of the following:

- `HALME_SUCCESS`–the operation was successful
- `HALME_BADARG`–an invalid argument was specified
- `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized

### 2.4.2.19 `halMe_PutCtxStatus()`

Writes the context-status CSR of the contexts specified by the `ctxMask` parameter.

The microengine must be stopped before calling this function.

#### C Syntax

```
int halMe_PutCtxStatus(uchar me, uint ctxMask, uint value);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine to write to. |
| `ctxMask` | Specifies the context or contexts to write to. |
| `value` | Specifies the longword value to write. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_MEACTIVE`–the microengine was active; the call failed |

## 2.4.2.20 `halMe_GetCtxStatus()`

Reads the context-status CSR for the context specified by the `ctx` parameter.

The microengine must be stopped before calling this function.

### C Syntax

```
int halMe_GetCtxStatus(uchar me, uchar ctx, uint &value);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine to read from. |
| `ctxMask` | Specifies the context to read from. |
| `value` | Specifies the location at which to return the longword value read. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_MEACTIVE`–the microengine was active; the call failed |

### 2.4.2.21 `halMe_GetMeCsr()`

Reads the microengine control status register indicated by `csr` and returns the value in the `value` parameter. The `csr` value must be a valid microengine CSR offset, as indicated in the *Intel® IXP2400 Network Processor Hardware Reference Manual* or the *Intel® IXP2800 Network Processor Hardware Reference Manual*, as appropriate.

Since the Local CSRs are single-ported, the microengine wins arbitration if the Intel XScale® core and a microengine attempts to access the CSR on a given cycle. Hence, this function checks the `Local_CSR_Status` register and retries until the operation is successful or until 500 attempts have been made.

#### C Syntax

```
int halMe_GetMeCsr(uchar me, uint csr, uint *value);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `csr` | A valid microengine CSR offset. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |
| `value` | A pointer to the location of the requested microengine control status register. |

### 2.4.2.22 `halMe_PutMeCsr()`

Writes the longword value to the microengine CSR indicated by csr parameter. The csr value must be a valid ME CSR offset, as indicated in the network processor’s Programmer’s Reference Manual.

Since the Local CSRs are single-ported, the microengine wins arbitration if the Intel® XScale™ core and microengine attempt to access the CSR on a given cycle. Hence, this function checks the `Local_CSR_Status` register and retries until the operation is successful or until 500 attempts have been made.

#### C Syntax

```
int halMe_PutMeCsr(uchar me, uint csr, uint value);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `csr` | A valid microengine CSR offset. |
| `value` | The new value for the specified microengine control status register. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |

### 2.4.2.23 `halMe_PutMeLmMode()`

Sets the local-memory mode to relative or global.

#### C Syntax

```
int halMe_PutMeLmMode(uchar me, ixp_RegType_T lmType, uchar mode);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `lmType` | Specifies the local memory bank. This value is one of:<br>• `IXP_LMEM0`<br>• `IXP_LMEM1` |
| `mode` | Specifies the local memory mode. This value is one of:<br>• zero–the memory mode is relative<br>• one–the memory mode is global |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_UNINIT`–a bad debug library is associated with the specified microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |

### 2.4.2.24 `halMe_PutMeCtxMode()`

Sets the microengine context mode to either four or eight.

#### C Syntax

```
int halMe_PutMeCtxMode(uchar me, uchar mode);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine of interest. |
| `mode` | Specifies the context mode. This value must be one of {`4`, `8`}. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_UNINIT`–a bad debug library is associated with the specified microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLID`–a bad debug library is associated with the microengine, the library not initialized |

### 2.4.2.25 `halMe_PutMeNnMode()`

Sets the microengine next-neighbor mode to either write to itself or to its neighbor.

#### C Syntax

```
int halMe_PutMeNnMode(uchar me, uchar mode);
```

#### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `mode` | The next-neighbor mode to set. This value is one of:<br>• zero–the next-neighbor mode is write to self<br>• one–the next-neighbor mode is write to neighbor |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_UNINIT`–a bad debug library is associated with the specified microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |

## 2.4.2.26 `halMe_GetCtxWakeupEvents()`

Reads the state of the microengine context wakeup-event signals.

### C Syntax

```
int halMe_GetCtxWakeupEvents(uchar me, uchar ctx, uint *events);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `ctx` | Specifies the microengine context of interest. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_BADARG`–an invalid argument was specified |
| `events` | A pointer to the location of the wakeup-event signals for the specified microengines and context. |

## 2.4.2.27 `halMe_PutCtxWakeupEvents()`

Writes the longword `events` to the microengine context wakeup-events CSR. Only the contexts with a corresponding bit set in `ctxMask` are written; unspecified contexts remains unchanged.

### C Syntax

```
int halMe_PutCtxWakeupEvents(uchar me, uint ctxMask, uint events);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `ctxMask` | Specifies the microengine context or contexts. |
| `events` | The longword specifying the new wakeup-event signals. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.28 `halMe_GetCtxSigEvents()`

Reads the signal events for the specified microengine context.

### C Syntax

```
int halMe_GetCtxSigEvents(uchar me, uchar ctx, uint *events);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `ctx` | Specifies the microengine context of interest. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_BADARG`–an invalid argument was specified |
| `events` | A pointer to the location of the current signal events for the specified microengine context. |

### 2.4.2.29 `halMe_PutCtxSigEvents()`

Writes the long-word value of the event parameter to the signal events of the specified microengine contexts.

#### C Syntax

```
int halMe_PutCtxSigEvents(uchar me, uint ctxMask, uint events);
```

#### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `ctxMask` | Specifies the microengine context or contexts. |
| `events` | The longword specifying the new signal events. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_BADARG`–an invalid argument was specified |

### 2.4.2.30 `halMe_PutPC()`

Sets the indicated microengine context program counters to the value specified by the `upc` parameter.

#### C Syntax

```
int halMe_PutPC(uchar me, uint ctxMask, uint upc);
```

#### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `ctxMask` | Specifies the microengine context or contexts. |
| `upc` | The new program counter value. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |

## 2.4.2.31 `halMe_GetPC()`

Returns the program counter for the specified microengine context.

### C Syntax

```
int halMe_GetPC(uchar me, uchar ctx, uint *upc);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `ctx` | Specifies the microengine context of interest. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized |
| `upc` | A pointer to the location of the requested program counter value. |

## 2.4.2.32 `halMe_PutUwords()`

Writes a number of microwords to the specified microengine. The microengine must be disabled and out of reset before writing to its microstore.

### C Syntax

```
int halMe_PutUwords(uchar me, uint uAddr, uint numWords, uword_T *uWord);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `uAddr` | The microstore address specifying the start of the write operation. |
| `numWords` | The number of microwords to write. |
| `uWord` | A pointer to the location of the microwords to write. |

### Output/Returns

Return Value — Returns one of the following:

- `HALME_SUCCESS`–the operation was successful
- `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized
- `HALME_ACTIVE`–the microengine is active
- `HALME_BADARG`–an invalid argument was specified
- `HALME_RESET`–the microengine is in the reset state

## 2.4.2.33 `halMe_GetUwords()`

Reads and returns the number of microwords from the specified microengine microstore. Microengines must be disabled before reading from the microstore.

### C Syntax

```
int halMe_GetUwords(uchar me, uint uAddr, uint numWords, uword_T *uWord);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `uAddr` | The microstore address specifying the start of the write operation. |
| `numWords` | The number of microwords to read. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine, the library not initialized<br>• `HALME_ACTIVE`–the microengine is active<br>• `HALME_BADARG`–an invalid argument was specified<br>• `HALME_RESET`–the microengine is in the reset state |
| `uWord` | A pointer to the location of the requested microwords. |

## 2.4.2.34 `halMe_PutRelDataReg()`

Writes a longword to the relative microengine general purpose register specified by the `me`, `ctx`, reg`Type`, and `regNum` parameters.

The microengine must be disabled before calling this function.

### C Syntax

```
int halMe_PutRelDataReg(
    uchar me,
    uchar ctx,
    ixp_RegType_T regType,
    ushort regNum,
    uint data);
```

### Input

| | |
|---|---|
| `me` | Specifies the microengine to write to. |
| `ctx` | Specifies the context. |
| `regType` | Specifies the type of register to write to—this must be one of {`IXP_GPA_REL`, `IXP_GPB_REL`, `IXP_DR_RD_REL`, `IXP_SR_RD_REL`, `IXP_DR_WR_REL`, `IXP_SR_WR_REL`, `IXP_NEIGH_REL`}. |
| `regNum` | Specifies the register number to write to. |
| `data` | The longword to write. |

### Output/Returns

| | |
|---|---|
| Return Value | If the function succeeds the function returns `HALME_SUCCESS`, otherwise returns one of {`HALME_BADLIB`, `HALME_BADARG`, `HALME_ENABLED`, `HALME_FAIL`,`HALME_RESET`}. |

### 2.4.2.35 `halMe_GetRelDataReg()`

Reads a longword from the relative microengine general purpose register specified by the `me`, `ctx`, `regType`, and `regNum` parameters.

The microengine must be disabled before calling this function.

#### C Syntax

```
int halMe_GetRelDataReg(
    uchar me,
    uchar ctx,
    ixp_RegType_T regType,
    ushort regNum,
    uint *data);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine to read from. |
| `ctx` | Specifies the context. |
| `regType` | Specifies the type of register to read from—this must be one of {`IXP_GPA_REL`, `IXP_GPB_REL`, `IXP_DR_RD_REL`, `IXP_SR_RD_REL`, `IXP_DR_WR_REL`, `IXP_SR_WR_REL`, `IXP_NEIGH_REL`}. |
| `regNum` | Specifies the register number to read from. The register number is relative to the context. Therefore, this number is in the range of 0 through max - 1, where max is the maximum number of context-relative registers for the type specfied by regType argument and the context mode. |

#### Input/Output

| | |
|---|---|
| `data` | Specifies the location at which to return the requested longword. |

#### Output/Returns

| | |
|---|---|
| Return Value | If the function succeeds the function returns `HALME_SUCCESS`, otherwise returns one of {`HALME_BADLIB`, `HALME_BADARG`, `HALME_ENABLED`, `HALME_FAIL`,`HALME_RESET`}. |

### 2.4.2.36 `halMe_PutAbsDataReg()`

Writes a longword to the microengine absolute general purpose register specified by the `me`, `ctx`, `regType`, and `regNum` parameters.

The microengine must be disabled before calling this function.

#### C Syntax

```
int halMe_PutAbsDataReg(
    uchar me,
    uchar ctx,
    ixp_RegType_T regType,
    ushort regNum,
    uint data);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine to write to. |
| `ctx` | Specifies the context. |
| `regType` | Specifies the type of register to write to—this must be one of {`IXP_GPA_ABS`, `IXP_GPB_ABS`, `IXP_DR_RD_ABS`, `IXP_SR_RD_ABS`, `IXP_DR_WR_ABS`, `IXP_SR_WR_ABS`, `IXP_NEIGH_ABS`}. |
| `regNum` | Specifies the register number to write to. |
| `data` | The longword to write. |

#### Output/Returns

| | |
|---|---|
| Return Value | If the function succeeds the function returns `HALME_SUCCESS`, otherwise returns one of {`HALME_BADLIB`, `HALME_BADARG`, `HALME_ENABLED`, `HALME_FAIL`,`HALME_RESET`}. |

### 2.4.2.37 halMe_GetAbsDataReg()

Reads a longword from the microengine absolute general purpose register specified by the `me`, `ctx`, `regType`, and `regNum` parameters.

The microengine must be disabled before calling this function.

#### C Syntax

```
int halMe_GetAbsDataReg(
   uchar me,
   ixp_RegType_T regType,
   ushort regNum,
   uint *data);
```

#### Input

| | |
|---|---|
| `me` | Specifies the microengine to read from. |
| `regType` | Specifies the type of register to read from—this must be one of {`IXP_GPA_ABS`, `IXP_GPB_ABS`, `IXP_DR_RD_ABS`, `IXP_SR_RD_ABS`, `IXP_DR_WR_ABS`, `IXP_SR_WR_ABS`, `IXP_NEIGH_ABS`}. |
| `regNum` | Specifies the register number to read from. |

#### Input/Output

| | |
|---|---|
| `data` | Specifies the location at which to return the requested longword. |

#### Output/Returns

| | |
|---|---|
| Return Value | If the function succeeds the function returns `HALME_SUCCESS`, otherwise returns one of {`HALME_BADLIB`, `HALME_BADARG`, `HALME_ENABLED`, `HALME_FAIL`,`HALME_RESET`}. |

### 2.4.2.38 `halMe_PutLM()`

Writes a longword value to the specified microengine's local memory location specified by the lmaddr word address.

#### C Syntax

```
int halMe_PutLM(uchar me, ushort lmAddr, uint value);
```

#### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `lmAddr` | Local memory word address. |
| `value` | The longword to write. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified |

### 2.4.2.39 `halMe_GetLM()`

Reads a longword value from the specified microengine local memory location specified by the lmAddr word address.

#### C Syntax

```
int halMe_GetLM(uchar me, ushort lmAddr, uint *value);
```

#### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |
| `lmAddr` | Local memory word address. This is in the range of 0 - 639 and can be obtained from using the halMe_PutTbuf() or the halMe_PutLM() API functions. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified |
| `value` | A pointer to the location of location of the longword read from the specified register. |

## 2.4.2.40 `halMe_VerifyMe()`

Checks that the value specified by the me parameter is a valid microengine number. The library must be initialized before calling this function.

### C Syntax

```
int halMe_VerifyMe(uchar me);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine or microengines that are the target of the operation. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.41 `halMe_VerifyMeMask()`

Checks that the value specified by the `meMask` argument specifies valid microengines. The library must be initialized before calling this function.

### C Syntax

```
int halMe_VerifyMeMask(uint meMask);
```

### Input

| | |
|---|---|
| `meMask` | The bits correspond to the microengine number and address of microengines to be initialized. For example, the first microengine in the second cluster corresponds to bit 16. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.42 `halMe_SysMemVirtSize()`

Returns the virtual memory region sizes:

### C Syntax

```
int halMe_sysMemVirtSize(uint *sys, uint *dram, uint *sram0, uint *sram1,
    uint *sram2, uint *sram3);
```

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified |
| `sys` | A pointer to the location of the location of the size of memory mapped at zero for the operating system. |
| `dram` | A pointer to the location of the location of the size of memory mapped and usable by hardware. |
| `sram0` | A pointer to the location of the location of the size of SRAM bank zero. |
| `sram1` | A pointer to the location of the location of the size of SRAM bank one. |
| `sram2` | A pointer to the location of the location of the size of SRAM bank two. |
| `sram3` | A pointer to the location of the location of the size of SRAM bank three. |

### 2.4.2.43 `halMe_GetVirAddr()`

Translates an Intel XScale® core physical address to a virtual address.

#### C Syntax

```
unsigned int halMe_GetVirAddr(uint phyAddr);
```

#### Input

| | |
|---|---|
| `phyAddr` | An Intel XScale® core physical address. |

#### Output/Returns

| | |
|---|---|
| Return Value | Returns the virtual address or `0xffffffff` if the address cannot be translated. |

### 2.4.2.44 halMe_GetVirXaddr()

Return the starting virtual address of the spcified physical region (*phyXaddr+physSize)* if it's mapped by the HAL, otherwise, an invalid address (0xffffffffffffffff) is returned.

#### C Syntax

```
uint64 halMe_GetVirXaddr(uint64 phyXaddr, unsigned int physSize);
```

#### Input

| | |
|---|---|
| `phyAddr` | *phyXaddr+physSize* |

#### Output/Returns

| | |
|---|---|
| Return Value | The virtual address, or 0xffffffffffffffff if the address is not mapped by the HAL library. |

### 2.4.2.45 halMe_GetPhyXaddr

Return the starting physical address of the spcified virtual region (*virtXaddr+virtSize)* if it was mapped by the HAL, otherwise, an invalid address (0xffffffffffffffff) is returned.

#### C Syntax

```
uint64 halMe_GetPhyXaddr(uint64 virtXaddr, unsigned int virtSize);
```

#### Input

| | |
|---|---|
| `phyAddr` | *phyXaddr+physSize* |

#### Output/Returns

| | |
|---|---|
| Return Value | The virtual address, or 0xffffffffffffffff if the address is not mapped by the HAL library. |

### 2.4.2.46 halMe_PutCAM()

Writes the longword data to the specified microengine CAM entry and marks it as the MRU—that is, as the most recently used. The entry and state must be in the range of zero through fifteen.

#### C Syntax

```
int dbgMe_PutCAM(uchar me, uchar *entry, uint data, uchar state);
```

#### Input

| | |
|---|---|
| `me` | A mask specifying the microengine that is the target of the operation. |
| `entry` | A pointer to the location of the CAM entry ID. |
| `data` | The CAM data to write. |
| `state` | The value of the state for the specified CAM entry. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.47 halMe_GetCAMState()

Reads the entire contents of the microengine CAM—that is, sixteen CAM entries—and stores the tag value and status in the appropriate parameters.

### C Syntax

```
int halMe_GetCAMState(uchar me, uint *tags, uchar *state, uint *lru);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine that is the target of the operation. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified |
| `tags` | An array of 16 unsigned integers to receive the CAM tag-data of each CAM entry. |
| `state` | An array of 16 chars to receive the CAM state-bits of each CAM entry. |
| `lru` | A pointer to the location of the array describing the order of use of the CAM entries with values of zero through fifteen. The entry with a zero value is the Least Recently Used—that is LRU—entry. The entry with the value of fifteen is the Most Recently Used—that is, MRU—entry. |

## 2.4.2.48 halMe_IsInpStateSet()

Determines if the microengine is in the state specified by `inpState` and returns a value indicating whether it is set, not set, or an error occurred. There are 16 values for `inpState`—zero through fifteen—and their meaning is specific to the chip implementation. See `BR_INP_STATE` instruction in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* or *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for a description of and values for `inpState`.

### C Syntax

```
int halMe_IsInpStateSet (uchar me, uchar inpState);
```

### Input

| | |
|---|---|
| `me` | A mask specifying the microengine that is the target of the operation. |
| `inpState` | A chip-specific value denoting a microengine state. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_ISSET`–the value is set<br>• `HALME_NOTSET`–the value is not set<br>• `HALME_BADLIB`–a bad debug library is associated with the microengine—the library is not initialized<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.49 halMe_PutTbuf()

Writes a longword from `value` to the TBUF starting at the address location specified by `tbufByte`—this address must be aligned on a four-byte boundary.

### C Syntax

```
int halMe_PutTbuf(uint tbufByte, uint value);
```

### Input

| | |
|---|---|
| `tbufByte` | Specifies the starting address of a TBUF—this address must be aligned on a four-byte boundary. |
| `value` | The longword to write to the TBUF. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified. |

## 2.4.2.50 halMe_GetRbuf()

Reads a longword from the `RBUF` location `rbufByte`—this address must be aligned on a four-byte boundary—and stores the longword at the location specified by the `value` pointer.

### C Syntax

```
int halMe_GetRbuf(uint rbufByte, uint *value);
```

### Input

| | |
|---|---|
| `rbufByte` | Specifies the starting address of an RBUF. This address must be aligned on a four-byte boundary and is in the range of 0 through 0x1fff of the buffer. |
| `value` | On input a pointer to a longword used to return the longword from the RBUF specified. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.51 halMe_PutTbufEleCntl()

Writes the `eleCtlA`, and `eleCtlB` long-words to the TBUF element control specified by `eleNum` whose value must be in the range of zero to 127. See *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* or *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the format and information about `RBUF` control.

### C Syntax

```
int halMe_PutTbufEleCntl(uint eleNum, uint eleCtlA, uint eleCtlB);
```

### Input

| | |
|---|---|
| `eleNum` | Specifies the element control of interest. |
| `eleCtlA`<br>`eleCtlB` | The first and second longword to write to the TBUF element control specified. |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• `HALME_SUCCESS`–the operation was successful<br>• `HALME_BADARG`–an invalid argument was specified |

## 2.4.2.52 halMe_PutCtxIndrCsr()

Write a 32-bit value to the micro-engine context(s) indirect CSR. It's unsafe to call this function while the ME is enabled.

### C Syntax

```
int halMe_PutCtxIndrCsr(unsigned char me, unsigned int ctxMask, unsigned int
meCsr, unsigned int csrVal);
```

### Input

| | |
|---|---|
| me | Specifies the microengine |
| ctxMask | Specifies the contexts CSR to be written |
| meCsr | The indirect CSR to be written. Must be one of the following:<br>• CTX_FUTURE_COUNT_INDIRECT<br>• CTX_WAKEUP_EVENTS_INDIRECT<br>• CTX_STS_INDIRECT<br>• CTX_SIG_EVENTS_INDIRECT<br>• LM_ADDR_0_INDIRECT<br>• LM_ADDR_1_INDIRECT |
| csrVal | Specifies the long-word value to write |

### Output/Returns

| | |
|---|---|
| Return Value | Returns one of the following:<br>• HALME_SUCCESS -- the operation was successfull<br>• HALME_BADLIB -- bad or uninitialized debug library<br>• HALME_BADARG -- an invalid argument was specified<br>• HALME_MEACTIVE -- the ME is active |

## 2.4.2.53 halMe_GetCtxIndrCsr()

Read a micro-engine context indirect CSR. It's unsafe to call this function while the ME is enabled.

### C Syntax

```
int halMe_GetCtxIndrCsr(unsigned char me, unsigned int ctx, unsigned int
meCsr, unsigned int *csrVal);
```

### Input

| | |
|---|---|
| me | Specifies the microengine |
| ctxMask | Specifies the contexts CSR to be written |
| meCsr | The indirect CSR to be written. Must be one of the following:<br>• CTX_FUTURE_COUNT_INDIRECT<br>• CTX_WAKEUP_EVENTS_INDIRECT<br>• CTX_STS_INDIRECT<br>• CTX_SIG_EVENTS_INDIRECT<br>• LM_ADDR_0_INDIRECT<br>• LM_ADDR_1_INDIRECT |

### Output/Returns

| | |
|---|---|
| csrVal | Specifies the long-word value to write |
| Return Value | Returns one of the following:<br>• HALME_SUCCESS -- the operation was successfull<br>• HALME_FAIL -- the call failed<br>• HALME_BADLIB -- bad or uninitialized debug library<br>• HALME_BADARG -- an invalid argument was specified<br>• HALME_MEACTIVE -- the ME is active |

## 2.5 OSSL Services

The Operating System Services Layer (OSSL) defines a portable interface for operating system services required by all levels. Modules in the IXA software infrastructure should not use directly services provided by an operating system; instead, these services need to be accessed via a library (OSSL) that provides portable interfaces to the services normally provided by an operating system and used by one or more IXA software infrastructure modules. The OSSL services may also be used by an application based on IXA SDK.

The definition of the OSSL needs to include the definition of the API that exports the OS services. The OSSL provides an abstraction layer for the following types of OS services:

- Thread management
- Synchronization
- Mutual Exclusion
- Timers
- Dynamic Memory Management
- Message Logging

### 2.5.1 OSSL Abstraction Layer

The OSSL abstraction layer is a set of functions that provides OS independent APIs and data types. In IXA SDK 3.0 Release 4, OSSL has support for: threads, mutexes, semaphores, timers, dynamic memory management, and message logging. These primitives are implemented as different sets of APIs. These APIs invoke corresponding OS functions.

#### 2.5.1.1 Threads

It is often essential to organize applications into independent, though cooperating, tasks. Each of these tasks, while executing, is called a thread. Threads have immediate, shared access to most resources; also maintain enough separate contexts to maintain individual control. 'ix_ossl_thread_create(..)' function call is used to create an OSSL thread. A user provided thread entry point function is passed as an argument to the thread creation function.

#### 2.5.1.2 Semaphores

Semaphores provide interthread synchronization mechanisms. They coordinate a thread's execution with other threads. The OSSL semaphores are binary and have 'available' or 'unavailable' states. A thread signals the occurrence of an event by setting the semaphore to an 'available' state. Another thread waits for the semaphore to become 'available'. The waiting thread will block until the semaphore becomes 'available' or for a specified timeout period. Currently, counting semaphores are not supported.

#### 2.5.1.3 Mutual Exclusion

Mutual Exclusion (Mutex) is used to guard a critical section. Critical section is a part of the program that needs exclusive access to shared resources such as a buffer, an I/O device etc. A critical section should be protected by a mutex for the correct behavior of the code executing in the critical section. Each critical section should be protected by a mutex. Critical sections are identified during development and a mutex should be allocated using 'ix_ossl_mutex_init'. When a thread wants to access a critical section, it must first lock the mutex for that critical section. When the thread is finished with the critical section, it unlocks the mutex, allowing another thread to use the critical section.

#### 2.5.1.4 Semaphores versus Mutexes

Semaphores are used for thread coordination and synchronization. For example, a receiving thread can set a semaphore available as soon it receives packets from the port. A processing thread will wait on this semaphore and when a semaphore becomes available, it will process the packet buffer. This kind of thread coordination will be used to avoid busy wait in a program. A busy wait is a waste of CPU resource; instead, the thread should wait on a semaphore, which puts the thread in a pending queue and is a more efficient way of using the system resources.

Mutexes are used to provide mutual exclusive access to critical sections of the program code. For example, a thread that wants to access a shared packet buffer should lock the mutex for the buffer before it starts using it. After it completes the processing, it unlocks the mutex. Each mutex lock must be followed by a subsequent mutex unlock by the same thread. Otherwise, deadlocks will occur and program will enter in to an undesirable state.

A mutex is owned by a thread that locked it, a semaphore is not. That means that a semaphore can be unlocked by another thread unlike a mutex. A mutex has to be unlocked by the same thread that locked it.

#### 2.5.1.5 Timers

OSSL timers provide primitives to get system time and cause thread delays. System time is provided in nanoseconds resolution.

#### 2.5.1.6 Memory Management

Memory management functions provide dynamic memory allocation, deallocation, memory copy and memory set operations. There is no garbage collection done in OSSL and modules in IXA SDK should deallocate the dynamic memory using 'ix_ossl_free'.

#### 2.5.1.7 Message Logging

Message logging is used to log and/or display the error messages. The Message logging is initialized using 'ix_ossl_message_log_init' function. This function should be called before any call to ix_ossl_message_log(). For each OS the messages will be logged into an implementation dependent stream. Further details will be provided on where the messages will be logged! 'ix_ossl_message_log' is used to log error messages.

### 2.5.2 OSSL Functions

Each OSSL API function uses the corresponding OS provided service. Basic OSSL services provided are:

- Create Thread
- Delete Thread
- Get Thread id

- Kill Thread
- Create Semaphore
- Delete Semaphore
- Take Semaphore
- Give Semaphore
- Create Mutex
- Delete Mutex
- Lock Mutex
- Unlock Mutex
- Sleep
- Sleep Ticks
- Get time
- Dynamic Memory Allocation
- Memory Deallocation
- Memory Copy
- Memory Set
- Message Logging

## 2.5.3 OSSL Data Structures

The data structures exported by OSSL are presented in this section.

### 2.5.3.1 ix_error

All the OSSL functions will return an error token of the type ix_error. This type is defined as an unsigned 32-bit integer and several pieces of information will be packed in this token as follows:

**Table 2-4. ix_error Fields**

| Bits | Size (in # of bits) | Field | Description |
|---|---|---|---|
| 0:15 | 16 | Error code | This field represents the error code of ix_error token. |
| 16:23 | 8 | Error group | This field represents the error group of ix_error token. |
| 24:31 | 8 | Error level | This field represents the error level of ix_error token. |

IXA SDK errors belong to different error groups. This feature permits the reuse of same error code numbers for different modules (error groups). A module can define the error codes independently from other modules, as long as it uses different error group. However, some synchronization should be done in choosing the error groups. The error level describes the severity of an error. Based on this level, the programmer can take different actions to handle the error. Macros are provided to get and set the different bit fields of an ix_error token.

```
/**
 * TYPENAME: ix_error
 *
 * DESCRIPTION: This type defines an IXA SDK error token.
 *
 */
typedef ix_uint32     ix_error;
```

### 2.5.3.2 IX_SUCCESS

This symbol defines an error token corresponding to successful completion of an operation.

```
#define IX_SUCCESS    ((ix_error)0)
```

### 2.5.3.3 ix_ossl_error_code

This enumeration type defines error codes returned by OSSL calls.

```
typedef enum ix_e_ossl_error_code
{
    IX_OSSL_ERROR_SUCCESS = IX_ERROR_MAKE_GROUP(OSSL),
    /* general error codes */
    IX_OSSL_ERROR_INVALID_ARGUMENTS,
    IX_OSSL_ERROR_INVALID_OPERATION,
    IX_OSSL_ERROR_TIMEOUT,

    /* Message error codes */
    IX_OSSL_ERROR_MESSAGE_LOG_TOO_MANY_ARGS,
    IX_OSSL_ERROR_MESSAGE_LOG_FORMAT_STRING,
    IX_OSSL_ERROR_MESSAGE_UNSUPPORTED_DATA_TYPE,

    /* sleep error codes */
    IX_OSSL_ERROR_SLEEP_FAILED,
    IX_OSSL_ERROR_SLEEP_CALLED_FROM_ISR,
    IX_OSSL_ERROR_SIGNALED_IN_SLEEP_PERIOD,
    IX_OSSL_ERROR_SLEEP_CLOCK_RES_FAILED,
    IX_OSSL_ERROR_SLEEP_UNSLEPT_MSEC,
    IX_OSSL_ERROR_SLEEP_TIMER_RESOURCE,

    /* mutex specific error codes */
    IX_OSSL_ERROR_MUTEX_INIT_FAILED,
    IX_OSSL_ERROR_MUTEX_LOCK_BAD_ID,
    IX_OSSL_ERROR_MUTEX_LOCK_DEADLOCK,
    IX_OSSL_ERROR_MUTEX_LOCK_INSUF_RES,
    IX_OSSL_ERROR_MUTEX_LOCK_FAILED,
    IX_OSSL_ERROR_MUTEX_LOCK_TIMEOUT,
    IX_OSSL_ERROR_MUTEX_UNLOCK_FAILED,
    IX_OSSL_ERROR_MUTEX_UNLOCK_BAD_ID,
    IX_OSSL_ERROR_MUTEX_UNLOCK_CALLED_FROM_ISR,
    IX_OSSL_ERROR_MUTEX_UNLOCK_INVALID_OP,
    IX_OSSL_ERROR_MUTEX_FINI_BAD_ID,
    IX_OSSL_ERROR_MUTEX_FINI_CALLED_FROM_ISR,
    IX_OSSL_ERROR_MUTEX_FINI_NO_OBJECT_TO_DESTROY,
    IX_OSSL_ERROR_MUTEX_FINI_LOCKED,
    IX_OSSL_ERROR_MUTEX_FINI_FAILED,

    /* semaphore specific error code */
    IX_OSSL_ERROR_SEMAPHORE_INIT_FAILED,
    IX_OSSL_ERROR_SEMAPHORE_TAKE_FAILED,
    IX_OSSL_ERROR_SEMAPHORE_TAKE_BAD_ID,
    IX_OSSL_ERROR_SEMAPHORE_TAKE_DEADLOCK,
    IX_OSSL_ERROR_SEMAPHORE_TAKE_SIG_INTR,
    IX_OSSL_ERROR_SEMAPHORE_GIVE_FAILED,
    IX_OSSL_ERROR_SEMAPHORE_GIVE_BAD_ID,
    IX_OSSL_ERROR_SEMAPHORE_GIVE_CALLED_FROM_ISR,
    IX_OSSL_ERROR_SEMAPHORE_GIVE_INVALID_OP,
    IX_OSSL_ERROR_SEMAPHORE_FINI_BAD_ID,
    IX_OSSL_ERROR_SEMAPHORE_FINI_CALLED_FROM_ISR,
    IX_OSSL_ERROR_SEMAPHORE_FINI_NO_OBJECT_TO_DESTROY,
    IX_OSSL_ERROR_SEMAPHORE_FINI_FAILED,
    IX_OSSL_ERROR_SEMAPHORE_FLUSH_FAILED,

    /* thread specific error codes */
    IX_OSSL_ERROR_THREAD_CREATE_FAILED,
    IX_OSSL_ERROR_THREAD_CREATE_ID,
    IX_OSSL_ERROR_THREAD_CREATE_CALLED_FROM_ISR,
    IX_OSSL_ERROR_THREAD_CREATE_OUT_OF_MEMORY,
    IX_OSSL_ERROR_THREAD_CREATE_MEMORY_BLOCK_ERR,
    IX_OSSL_ERROR_THREAD_CREATE_MEMORY_UNINITIALIZED_OBJLIB,
    IX_OSSL_ERROR_THREAD_CREATE_NULL_ENTRY_POINT,
    IX_OSSL_ERROR_THREAD_CREATE_INVALID_ATTR,
    IX_OSSL_ERROR_THREAD_SET_PRIORITY_FAILED,
    IX_OSSL_ERROR_THREAD_SET_PRIORITY_ON_BAD_THREAD_ID,
    IX_OSSL_ERROR_THREAD_SET_PRIORITY_ILLEGAL_PRIORITY,
    IX_OSSL_ERROR_THREAD_SET_PRIORITY_INVAL_POLICY_OR_SCHED_PARAM,
    IX_OSSL_ERROR_THREAD_SET_PRIORITY_UNSUPRTD_POLICY_OR_SCHED_PARAM,
    IX_OSSL_ERROR_THREAD_SET_PRIORITY_INSUFFICIENT_PRIVS,
    IX_OSSL_ERROR_THREAD_GET_ID_FAILED,
    IX_OSSL_ERROR_THREAD_KILL_CALLED_FROM_ISR,
    IX_OSSL_ERROR_THREAD_KILL_TASK_ALREADY_DEAD,
    IX_OSSL_ERROR_THREAD_KILL_INVALID_THREAD_ID,
    IX_OSSL_ERROR_THREAD_KILL_THREAD_UNAVAILABLE,
    IX_OSSL_ERROR_THREAD_KILL_FAILED,

    /* time specific error codes */
    IX_OSSL_ERROR_GET_TIME_FAILED,
    IX_OSSL_ERROR_GET_TIME_OF_DAY_FAILED

} ix_ossl_error_code;
```

### 2.5.3.4 ix_error_group

This enumeration describes the existing error groups in the system. New error groups should be added all the times at the end. Developers can use IX_ERROR_GROUP_LAST value to start assigning new error group numbers.

```
typedef enum ix_e_error_group
{
    IX_ERROR_GROUP_FIRST = 0,
    IX_ERROR_GROUP_RESOURCE_MANAGER = IX_ERROR_GROUP_FIRST,
        /* Resource Manager error group */
    IX_ERROR_GROUP_OSSL,
        /* OSSL API error group */
    IX_ERROR_GROUP_CC_INFRASTRUCTURE,
        /* Core component infrastructure API error group */
    IX_ERROR_GROUP_LAST
} ix_error_group;
```

### 2.5.3.5 ix_error_level

This enumeration designates the severity level of an error, zero representing no error at all, and higher numbers representing increasingly severe errors. Severity roughly means how much of the system has become corrupted and is likely to be no longer functional after the error occurs.

```
typedef enum ix_e_error_level
{
    IX_ERROR_LEVEL_FIRST = 0,

    /**
     * No error reported.
     *
     * This error level corresponds to returns from functions that
     * successfully accomplished their given tasks. This error level
     * will mean that no error occurred. It should not be used in any
     * valid  error token because it corresponds to IX_SUCCESS.
     */
    IX_ERROR_LEVEL_NONE = IX_ERROR_LEVEL_FIRST,

    /**
     * Warning level: Recoverable condition.
     *
     * This error level indicates that the requested task could not be
     * performed, but there are no lasting effects within the system
     * from that failure. The caller can recover from this condition
     * by simply continuing to execute as if the call had not been
     * made. Functions returning WARNING level messages should clearly
     * document the procedures for recovery.
     *
     * During debugging, WARNING messages should be logged for later
```

```
 * analysis as they may indicate the presence of a bug. This step
 * can probably be skipped in Production mode systems.
 */
IX_ERROR_LEVEL_WARNING,

/**
 * Error level: Unrecoverable Local Condition.
 *
 * This error level indicates that the requested task could not be
 * performed, and that one or more of the specific data items
 * involved in the call has been permanently altered, so that in
 * general recovery may require "leaking" a corrupted data item
 * or shutting down and restarting a service. Functions returning
 * ERROR level messages should clearly document recovery methods
 * for keeping the system running, and what storage or
 * functionality will be lost.
 *
 * During debugging, ERROR messages should be immediately
 * analyzed , along with any prior WARNING messages. Production
 * mode systems should log all ERROR messages, even when we
 * expect to recover from them.
 */
IX_ERROR_LEVEL_LOCAL,

/**
 * Error level: Unrecoverable Remote Condition.
 *
 * This error level indicates that there is an error in the module
 * we are trying to communicate with. That will have no impact on the
 * local module but the functionality of the system as a whole will be
 * most likely affected.
 * This error level might be used for the case the libraries are used
 * in conjunction with a Foreign Model running on Transactor.
 */
IX_ERROR_LEVEL_REMOTE,

/**
 * Error level: Unrecoverable Global Condition.
 *
 * This error level indicates that a function has detected a
 * condition that indicates that the system as a whole has become
 * compromised, and that multiple processing elements or data
 * modules are likely to experience a cascade failure.
 *
 * During debugging, this level of error message should cause the
 * system to freeze and drop into a debugger where the problem
 * can be analyzed by an engineer. In Production deployment,
 * huge effort should be made to log these errors and as much
 * supporting information as possible where the logs can be
 * analyzed by a responsible human.
 */
IX_ERROR_LEVEL_GLOBAL,

/**
 * Error level: Totally Impossible Condition.
```

```
         *
         * This error level indicates that a function has detected a
         * condition that indicates that a primary assumption in a module
         * design has been proven impossible, and that in general the
         * system would be doing arbitrarily bad things.
         *
         * During debugging, PANIC messages should cause the system to
         * freeze and drop into a debugger state where the problem can be
         * analyzed by an engineer.
         *
         * In a Production deployed system, PANIC messages should cause
         * an immediate attention.
         *
         * PANIC messages should be used very, very sparingly.
         */
        IX_ERROR_LEVEL_PANIC,

        IX_ERROR_LEVEL_LAST
    } ix_error_level;
```

### 2.5.3.6 IX_ERROR_GET_CODE()

| | |
|---|---|
| SYNTAX | IX_ERROR_GET_CODE(arg_Error) |
| DESCRIPTION | This macro retrieves the error code field from an error token. |
| PARAMETERS: | IN: arg_Error - error token of type ix_error. |
| RETURNS | Returns an ix_uint32 value representing the error code for the error token. |

### 2.5.3.7 IX_ERROR_SET_CODE()

| | |
|---|---|
| SYNTAX | IX_ERROR_SET_CODE(arg_Error, arg_ErrorCode) |
| DESCRIPTION | This macro sets the error code field for an error token. |
| PARAMETERS: | IN arg_Error - error token of type ix_error.<br>IN arg_ErrorCode - this is the new error code for the error token |
| RETURNS | Returns the new value of the error token. |

### 2.5.3.8 IX_ERROR_GET_GROUP()

| | |
|---|---|
| SYNTAX | IX_ERROR_GET_GROUP(arg_Error) |
| DESCRIPTION | This macro retrieves the error group field from an error token. |
| PARAMETERS: | IN arg_Error - error token of type ix_error. |
| RETURNS | Returns an ix_uint32 value representing the error group for the error token. |

### 2.5.3.9 IX_ERROR_SET_GROUP()

| | |
|---|---|
| SYNTAX | IX_ERROR_SET_GROUP(arg_Error, arg_ErrorGroup) |
| DESCRIPTION | This macro sets the error group field for an error token. |
| PARAMETERS: | IN arg_Error - error token of type ix_error.<br>IN arg_ErrorGroup - this is the new error group for the error token. |
| RETURNS | Returns the new value of the error token. |

### 2.5.3.10 IX_ERROR_GET_LEVEL()

| | |
|---|---|
| SYNTAX | IX_ERROR_GET_LEVEL(arg_Error) |
| DESCRIPTION | This macro retrieves the error level field from an error token. |
| PARAMETERS: | IN arg_Error - error token of type ix_error. |
| RETURNS | Returns a ix_uint32 value representing the error level for this error token. |

### 2.5.3.11 IX_ERROR_SET_LEVEL()

| | |
|---|---|
| SYNTAX | IX_ ERROR_SET_LEVEL(arg_Error, arg_ErrorLevel) |
| DESCRIPTION | This macro sets the error level field for an error token. |
| PARAMETERS: | IN arg_Error - error token of type ix_error.<br>IN arg_ErrorLevel - this is the new error level for the error token. |
| RETURNS | Returns the new value of the error token. |

### 2.5.3.12 IX_ERROR_NEW()

| | |
|---|---|
| SYNTAX | IX_ERROR_NEW(arg_ErrorCode, arg_ErrorGroup, arg_ErrorLevel) |
| DESCRIPTION | This macro generates a new error token based on the error code, group and level that are passed as arguments. |
| PARAMETERS: | IN arg_ErrorCode - the error code of the new error token to be generated. The range is 0..65535<br><br>IN arg_ErrorGroup - the error group of the new error token to be generated. The range is 0..255<br><br>IN arg_ErrorLevel - the error level of the new error token to be generated. The range is 0..255. |
| RETURNS | Returns a new ix_error token. |

## 2.5.4 ix_ossl_thread_t

The ix_osssl_thread_t data structure exports thread data type. OSSL will use this data type for handling threads.

```
typedef  os_specific_thread_type ix_ossl_thread_t;
```

## 2.5.5 ix_ossl_sem_t

This type defines OSSL semaphore type.

```
typedef os_specific_semaphore_typeix_ossl_sem_t;
```

## 2.5.6 ix_ossl_mutex_t

This type defines OSSL mutex type.

```
typedef os_specific_mutex_typeix_ossl_mutex_t;
```

## 2.5.7 ix_ossl_time_t

This type defines OSSL time. The ix_ossl_time_t struct has two fields. 'sec' and 'nsec'. Time value is computed as: sec * 10^9 + nsec.

```
typedef struct ix_s_time_t
{
    unsigned long sec; /* seconds */
    unsigned long nsec; /* nanoseconds [1, 999,999,999] */
} ix_ossl_time_t;
```

## 2.5.8 ix_ossl_thread_entry_point_t

This function pointer type defines an OSSL thread entry point function. The arg pointer is to a custom thread argument.

```
typedef ix_error (*ix_ossl_thread_entry_point_t)(void* arg);
```

## 2.5.9 ix_ossl_sem_state

This type defines OSSL binary semaphore states.

```
typedef enum ix_e_ossl_sem_state
{
    IX_OSSL_SEM_UNAVAILABLE = OS_SEM_UNVAILABLE,
    IX_OSSL_SEM_AVAILABLE = OS_SEM_AVAILABLE
} ix_ossl_sem_state;
```

### 2.5.10 ix_ossl_mutex_state

This type defines OSSL mutex states.

```
typedef enum ix_e_ossl_mutex_state
{
    IX_OSSL_MUTEX_UNLOCK = OS_MUTEX_UNLOCK,
    IX_OSSL_MUTEX_LOCK = OS_MUTEX_LOCK
} ix_ossl_mutex_state;
```

### 2.5.11 ix_ossl_thread_priority

This type define OSSL thread priority levels.

```
typedef enum ix_e_ossl_thread_priority
{
    IX_OSSL_THREAD_PRI_HIGH = OS_THREAD_PRI_HIGH,
    IX_OSSL_THREAD_PRI_MEDIUM_HIGH = OS_THREAD_PRI_MEDIUM_HIGH,
    IX_OSSL_THREAD_PRI_MEDIUM = OS_THREAD_PRI_MEDIUM,
    IX_OSSL_THREAD_PRI_MEDIUM_LOW = OS_THREAD_PRI_MEDIUM_LOW,
    IX_OSSL_THREAD_PRI_LOW = OS_THREAD_PRI_LOW
} ix_ossl_thread_priority;
```

### 2.5.12 ix_ossl_size_t

This type describes a generic size type.

```
typedef unsigned int ix_ossl_size_t;
```

### 2.5.13 Pre-processor Definitions

The following symbols are useful for specifying timeout values in ix_ossl_sem_take and ix_ossl_mutex_lock function calls:

```
#define IX_OSSL_WAIT_FOREVER    OS_WAIT_FOREVER /* (-1)UL */
#define IX_OSSL_WAIT_NONE       OS_WAIT_NONE     /* 0 */
```

## 2.6 OSSL APIs

The OSSL thread, semaphore, mutex, and timer APIs are presented in these sections. Each of these APIs is a set of OSSL functions built around an OS service. For example, thread API comprises thread related functions.

## 2.6.1 OSSL Thread API

The OSSL Thread API consists of create thread, delete thread, kill thread and get thread id functions.

### 2.6.1.1 ix_ossl_thread_create()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_thread_create(<br>ix_ossl_thread_entry_point_t entryPoint,<br>void* arg,<br>const char* name,<br>ix_ossl_thread_t* ptrTid); |
| DESCRIPTION | This function creates a cancelable thread that will execute the user-provided entry point function. Custom arguments can be passed to this function using the "arg" argument. On return, '*ptrTid' will contain the newly created thread id. The threads are created with IX_OSSL_THREAD_PRI_MEDIUM priority level. The priority can be changed later. |
| PARAMETERS: | IN entryPoint: Thread's entry point function.<br><br>IN arg: Pointer to custom arguments that will be passed to an entry point function as the first argument.<br><br>IN name: name to be assigned to the newly created thread<br><br>OUT ptrTid: Address at which the id of the calling thread will be returned. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_THREAD_CREATE_FAILED<br>IX_OSSL_ERROR_THREAD_CREATE_ID<br>IX_OSSL_ERROR_THREAD_CREATE_CALLED_FROM_ISR<br>IX_OSSL_ERROR_THREAD_CREATE_OUT_OF_MEMORY<br>IX_OSSL_ERROR_THREAD_CREATE_MEMORY_BLOCK_ERR<br>IX_OSSL_ERROR_THREAD_CREATE_MEMORY_UNINITIALIZED_OBJLIB<br>IX_OSSL_ERROR_THREAD_CREATE_NULL_ENTRY_POINT<br>IX_OSSL_ERROR_THREAD_CREATE_INVALID_ATTR |

### 2.6.1.2 ix_ossl_thread_get_id()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_thread_get_id(ix_ossl_thread_t* ptrTid); |
| DESCRIPTION | This function returns id of the calling thread. 'ptrTid' is the address at which the id of the calling thread will be returned. |
| PARAMETERS: | OUT ptrTid: Address at which the id of the calling thread will be returned. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_THREAD_GET_ID_FAILED |

### 2.6.1.3 ix_ossl_thread_exit()

| | |
|---|---|
| SYNTAX | void ix_ossl_thread_exit(void); |
| DESCRIPTION | This function causes the calling thread to exit. |
| PARAMETERS: | None. |
| RETURNS | None. |

### 2.6.1.4 ix_ossl_thread_kill()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_thread_kill(ix_ossl_thread_t tid); |
| DESCRIPTION | Kills the running thread specified by its thread id 'tid'. Because the thread will be killed instantly, the caller must be extremely careful when using this function, as the thread will not have time to release any of the resources it currently owns.<br><br>***Note:*** ix_ossl_exit_thread should be used to terminate a thread and release its resources instead! |
| PARAMETERS: | IN tid: id of the thread to be killed. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_THREAD_KILL_CALLED_FROM_ISR<br>IX_OSSL_ERROR_THREAD_KILL_TASK_ALREADY_DEAD<br>IX_OSSL_ERROR_THREAD_KILL_INVALID_THREAD_ID<br>IX_OSSL_ERROR_THREAD_KILL_THREAD_UNAVAILABLE<br>IX_OSSL_ERROR_THREAD_KILL_FAILED |

### 2.6.1.5 ix_ossl_thread_set_priority()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_thread_set_priority(<br>    ix_ossl_thread_t tid,<br>    ix_ossl_thread_priority priority); |
| DESCRIPTION | This function sets the priority of the thread indicated by 'tid'. Possible values for 'priority' are:<br><br>    IX_OSSL_THREAD_P_HIGH<br>    IX_OSSL_THREAD_PRI_MEDIUM_HIGH<br>    IX_OSSL_THREAD_PRI_MED<br>    IX_OSSL_THREAD_PRI_MEDIUM_LOW<br>    IX_OSSL_THREAD_PRI_LOW<br><br>The priority can be changed at any time after the thread has been started. |
| PARAMETERS: | IN tid: ID of the thread.<br>IN priority: New thread priority level. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_THREAD_SET_PRIORITY_FAILED<br>IX_OSSL_ERROR_THREAD_SET_PRIORITY_ON_BAD_THREA<br>D_ID<br>IX_OSSL_ERROR_THREAD_SET_PRIORITY_ILLEGAL_PRIORI |

| | |
|---|---|
| | TY<br>IX_OSSL_ERROR_THREAD_SET_PRIORITY_INVAL_POLICY_O R_SCHED_PARAM<br>IX_OSSL_ERROR_THREAD_SET_PRIORITY_UNSUPRTD_POLI CY_OR_SCHED_PARAM<br>IX_OSSL_ERROR_THREAD_SET_PRIORITY_INSUFFICIENT_P RIVS |

## 2.6.2 OSSL Semaphore API

This section describes OSSL semaphore API. The implemented semaphores are binary and have ’available’ or ’unavailable’ states. Semaphore init, semaphore take, semaphore give, semaphore flush and semaphore delete functions are supported in this category of OSSL API.

### 2.6.2.1 ix_ossl_sem_init()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_sem_init(<br>ix_ossl_sem_state start_state,<br>ix_ossl_sem_t* sid); |
| DESCRIPTION | This function initializes a new semaphore. ’sid’ is a pointer to an ix_ossl_sem_t. Upon success, ’*sid’ will be the semaphore id used in all other ix_ossl_sem functions. The newly created semaphore state will be initialized to the value of ’start_state’ (IX_OSSL_SEM_UNAVAILABLE or IX_OSSL_SEM_AVAILABLE). |
| PARAMETERS: | IN start_value: Initial value of the semaphore.<br>OUT sid: Address where the newly created semaphore id will be returned. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_SEMAPHORE_INIT_FAILED. |

### 2.6.2.2 ix_ossl_sem_take()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_sem_take(<br>ix_ossl_sem_t sid,<br>ix_uint32 timeout); |
| DESCRIPTION | If the semaphore is ’unavailable’, the calling thread is blocked. If the semaphore is ’available’, it is taken and control is returned to the caller. If the time indicated in ’timeout’ (expressed in milliseconds) is reached, the calling thread will unblock and return an error indication. If the timeout is set to ’IX_OSSL_WAIT_NONE’, the calling thread will never block; if it is set to ’IX_OSSL_WAIT_FOREVER’, the calling thread will block until the semaphore is available. |
| PARAMETERS: | IN sid: Semaphore id.<br>IN timeout: Timeout value of type ix_uint32 expressed in milliseconds |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_SEMAPHORE_TAKE_FAILED<br>IX_OSSL_ERROR_TIMEOUT |

### 2.6.2.3 ix_ossl_sem_give()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_sem_give(ix_ossl_sem_t sid); |
| DESCRIPTION | This function causes the next available thread in the pending queue to be unblocked. If no thread is pending on this semaphore, the semaphore state becomes ’available’. |
| PARAMETERS: | IN sid: Semaphore id. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_SEMAPHORE_GIVE_FAILED<br>IX_OSSL_ERROR_SEMAPHORE_GIVE_BAD_ID<br>IX_OSSL_ERROR_SEMAPHORE_GIVE_CALLED_FROM_ISR<br>IX_OSSL_ERROR_SEMAPHORE_GIVE_INVALID_OP |

### 2.6.2.4 ix_ossl_sem_flush()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_sem_flush(ix_ossl_sem_t sid); |
| DESCRIPTION | This function unblocks all pending threads without altering the semaphore count. ’sid’ is the id for the semaphore. |
| PARAMETERS: | IN sid: Semaphore id |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_SEMAPHORE_FLUSH_FAILED |

### 2.6.2.5 ix_ossl_sem_fini()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_sem_fini(ix_ossl_sem_t sid); |
| DESCRIPTION | This function frees a semaphore. ’sid’ is the semaphore id. The semaphore is terminated and all resources are freed. The threads pending on this semaphore will be released and return an error. |
| PARAMETERS: | IN sid: Semaphore id |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_SEMAPHORE_FINI_BAD_ID<br>IX_OSSL_ERROR_SEMAPHORE_FINI_NO_OBJECT_TO_DESTR<br>OY<br>IX_OSSL_ERROR_SEMAPHORE_FINI_FAILED |

## 2.6.3 OSSL Mutex API

The OSSL mutex API provides a mutual exclusion primitive that is used to guard the critical sections used by threads. The functions around this primitive are: mutex init, mutex lock, mutex unlock and mutex delete.

### 2.6.3.1 ix_ossl_mutex_init()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_mutex_init(<br>ix_ossl_mutex_state start_state,<br>ix_ossl_mutex_t* mid); |
| DESCRIPTION | This function initializes a new mutex. 'mid' is a pointer to an ix_ossl_mutex_t. Upon success, '*mid' will contain the mutex id. 'start_state' is the initial mutex state. |
| PARAMETERS: | IN start_state: 'start_state' is initial locking state. Valid values are:<br>IX_OSSL_MUTEX_LOCK will lock the mutex.<br>IX_OSSL_MUTEX_UNLOCK will leave the mutex unlocked.<br>OUT mid: Pointer where the id of the mutex created will be returned |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_MUTEX_INIT_FAILED |

### 2.6.3.2 ix_ossl_mutex_lock()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_mutex_lock(<br>ix_ossl_mutex_t mid,<br>ix_uint32 timeout); |
| DESCRIPTION | This function locks the mutex. If the mutex is already locked, the thread will block. If the time indicated in 'timeout' (expressed in milliseconds) is reached, the thread will unblock and return an error indication. If the timeout is set to 'IX_OSSL_WAIT_NONE', the thread will never block (this is a trylock). If the timeout is set to IX_OSSL_WAIT_FOREVER', the thread will block until the mutex is unlocked. |
| PARAMETERS: | IN mid: Mutex id.<br>IN timeout: Timeout value of type ix_uint32, expressed in milliseconds. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_MUTEX_LOCK_BAD_ID<br>IX_OSSL_ERROR_MUTEX_LOCK_DEADLOCK<br>IX_OSSL_ERROR_MUTEX_LOCK_INSUF_RES<br>IX_OSSL_ERROR_MUTEX_LOCK_FAILED<br>IX_OSSL_ERROR_MUTEX_LOCK_TIMEOUT |

### 2.6.3.3 ix_ossl_mutex_unlock()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_mutex_unlock(ix_ossl_mutex_t mid); |
| DESCRIPTION | This function unlocks the mutex. 'mid' is the mutex id. If there are threads pending on the mutex, the next one is given the lock. If there are no pending threads, then the mutex is unlocked. |
| PARAMETERS: | IN mid: Mutex id |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_MUTEX_UNLOCK_FAILED<br>IX_OSSL_ERROR_MUTEX_UNLOCK_BAD_ID |

IX_OSSL_ERROR_MUTEX_UNLOCK_CALLED_FROM_ISR
IX_OSSL_ERROR_MUTEX_UNLOCK_INVALID_OP

### 2.6.3.4 ix_ossl_mutex_fini()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_mutex_fini(ix_ossl_mutex_t mid); |
| DESCRIPTION | This function frees a mutex. 'mid' is the mutex id. The mutex is deleted and its resources are freed. Any threads pending on this mutex will be unblocked and return an error. |
| PARAMETERS: | IN mid: Mutex id |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_MUTEX_FINI_BAD_ID<br>IX_OSSL_ERROR_MUTEX_FINI_CALLED_FROM_ISR<br>IX_OSSL_ERROR_MUTEX_FINI_NO_OBJECT_TO_DESTROY<br>IX_OSSL_ERROR_MUTEX_LOCKED<br>IX_OSSL_ERROR_MUTEX_FINI_FAILED |

## 2.6.4 OSSL Timer API

The timer API provides OSSL calls for thread sleep and get time functions. Sleep granularity is milliseconds or OS ticks. The get time function returns the time with nanoseconds resolution (please refer to ix_ossl_time_t data type defined in previous sections for more details). A slew of macros around ix_ossl_time_t are also provided (ix_ossl_time_t Macros). Applications that need addition, subtraction and other operations on ix_ossl_time_t will use these macros.

### 2.6.4.1 ix_ossl_sleep()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_sleep(ix_uint32 sleeptime_ms); |
| DESCRIPTION | This function causes the calling thread to sleep for the specified time expressed in milliseconds. |
| PARAMETERS: | IN sleeptime_ms: sleep time specified in milliseconds. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_SLEEP_CALLED_FROM_ISR<br>IX_OSSL_ERROR_SIGNALED_IN_SLEEP_PERIOD |

### 2.6.4.2 ix_ossl_sleep_tick ()

| | |
|---|---|
| SYNTAX | ix_error   ix_ossl_sleep_tick(ix_uint32 sleeptime_ticks); |
| DESCRIPTION | This function causes the calling thread to sleep for the time specified in clock ticks. |
| PARAMETERS: | IN sleeptime_ticks: sleep time specified in clock ticks. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_SLEEP_CALLED_FROM_ISR<br>IX_OSSL_ERROR_SIGNALED_IN_SLEEP_PERIOD |

### 2.6.4.3 ix_ossl_time_get ()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_time_get(ix_ossl_time_t* ptime); |
| DESCRIPTION | This function places the current value of a timer, in seconds and nano-seconds, into an ix_ossl_time_t structure pointed by 'ptime'. This function does not provide a time-of-day. The intention is to provide a nano-second resolution time where supported. |
| PARAMETERS: | OUT ptime: Pointer to 'ix_ossl_time_t' structure where data will be returned. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | None. |

### 2.6.4.4 ix_ossl_time_of_day_get ()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_time_of_day_get(ix_uint32* arg_pTime); |
| DESCRIPTION | This function returns the UTC time of day. The time will be returned at the location addressed by arg_pTime. The returned time is expressed in milliseconds. Theoretically the resolution should be milliseconds, but on some OSs the best resolution would be seconds. |
| PARAMETERS: | OUT arg_pTime: Address where the time of day expressed in milliseconds will be stored upon return. |
| RETURNS | 0 if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_GET_TIME_OF_DAY_FAILED. |

### 2.6.4.5 ix_ossl_time_t Macros

For the following macros, 'true' is a non-zero integer value, and 'false' is a zero integer value. The result has type 'int'.

#### 2.6.4.5.1 IX_OSSL_TIME_EQ()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_EQ(a,b) |
| DESCRIPTION | Compares 'a' operand with 'b' operand. Returns true if they are equal and false otherwise. |
| PARAMETERS: | Anna - type ix_ossl_time_t<br>IN: b - type ix_ossl_time_t |
| RETURNS | True if a == b and false otherwise. |

#### 2.6.4.5.2 IX_OSSL_TIME_GT()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_GT(a,b) |
| DESCRIPTION | Compares 'a' operand with 'b' operand. Returns true if a > b, and false otherwise. |
| PARAMETERS: | IN: a - type ix_ossl_time_t<br>IN: b - type ix_ossl_time_t |

| | |
|---|---|
| RETURNS | True if a > b and false otherwise. |

### 2.6.4.5.3 IX_OSSL_TIME_LT()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_LT(a, b) |
| DESCRIPTION | Compares ’a’ operand with ’b’ operand. Returns true if a < b, and false otherwise. |
| PARAMETERS: | IN: a - type ix_ossl_time_t<br>IN: b - type ix_ossl_time_t |
| RETURNS | True if a < b and false otherwise. |

### 2.6.4.5.4 IX_OSSL_TIME_ISZERO()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_ISZERO(a) |
| DESCRIPTION | This macro checks if the operand ’a’ is zero. Returns true if ’a’ is zero (both sec and nsec fields must be zero) and false otherwise. |
| PARAMETERS: | IN: a - type ix_ossl_time_t |
| RETURNS | True if a is zero and false otherwise. |

### 2.6.4.5.5 IX_OSSL_TIME_SET()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_SET(a, b) |
| DESCRIPTION | This macro sets operand ’a’ to the value of operand ’b’. |
| PARAMETERS: | OUT: a - type ix_ossl_time_t<br>IN: b - type ix_ossl_time_t |

### 2.6.4.5.6 IX_OSSL_TIME_ADD()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_ADD(a, b) |
| DESCRIPTION | This macro performs a += b operation (i.e., it increments a by the value of b). |
| PARAMETERS: | INOUT: a - type ix_ossl_time_t<br>IN: b - type ix_ossl_time_t |

### 2.6.4.5.7 IX_OSSL_TIME_SUB()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_SUB(a, b) |
| DESCRIPTION | This macro performs a -= b operation (i.e., it decrements a by the value of b). |
| PARAMETERS: | INOUT: a - type ix_ossl_time_t<br>IN: b - type ix_ossl_time_t |

### 2.6.4.5.8 IX_OSSL_TIME_NORMALIZE()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_NORMALIZE(a) |
| DESCRIPTION | This macro normalizes the value of ’a’. If ’a.nsec’ > 10^9, it is decremented by 10^9 and ’a.sec’ is incremented by 1. |

| | |
|---|---|
| PARAMETERS: | INOUT: a - type ix_ossl_time_t |

#### 2.6.4.5.9 IX_OSSL_TIME_VALID()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_VALID(a) |
| DESCRIPTION | This macro checks whether ’a’ is a valid ix_ossl_time_t i.e. 0 =< a.nsec < 10^9. Returns true if ’a’ is valid and false otherwise. |
| PARAMETERS: | IN: a - type ix_ossl_time_t |
| RETURNS | True if ’a’ is valid ix_ossl_time_t and false otherwise. |

#### 2.6.4.5.10 IX_OSSL_TIME_ZERO()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_ZERO(a) |
| DESCRIPTION | This macro sets the value of a to zero. |
| PARAMETERS: | INOUT: a - type ix_ossl_time_t |

#### 2.6.4.5.11 IX_OSSL_TIME_CONVERT_TO_TICKS()

| | |
|---|---|
| SYNTAX | IX_OSSL_TIME_CONVERT_TO_TICK(a, b) |
| DESCRIPTION | This macro converts b value in ix_ossl_time_t to a value in OS ticks. |
| PARAMETERS: | IN: b - type ix_ossl_time_t<br>OUT: a - type unsigned int |

## 2.6.5 Memory Management API

Memory management API is used for to allocate and deallocate dynamically memory. Memory set and copy functions are provided.

### 2.6.5.1 ix_ossl_malloc()

| | |
|---|---|
| SYNTAX | void* ix_ossl_malloc(ix_ossl_size_t arg_Size); |
| DESCRIPTION | This function will allocate a memory block. The function returns a void pointer to the allocated space, or NULL if there is insufficient memory available. To return a pointer to a type other than void, use a type cast on the return value. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. If size is 0, ix_ossl_malloc allocates a zero-length item in the heap and returns a valid pointer to that item. Always check the return from ix_ossl_malloc, even if the amount of memory requested is small. |
| PARAMETERS: | IN arg_Size - the size of the memory block requested. |
| RETURNS | Returns a valid address if successful or a NULL for failure. |

### 2.6.5.2 ix_ossl_free ()

| | |
|---|---|
| SYNTAX | void ix_ossl_free (void* arg_pMemory); |

| | |
|---|---|
| DESCRIPTION | This function will free a memory block specified by the passed address. The ix_ossl_free function deallocates a memory block (arg_pMemory) that was previously allocated by a call to ix_ossl_malloc. If arg_pMemory is NULL, the pointer is ignored and ix_ossl_free immediately returns. Attempting to free an invalid pointer (a pointer to a memory block that was not allocated by ix_ossl_malloc) may affect subsequent allocation requests and cause errors. |
| PARAMETERS: | IN arg_pMemory - address of the memory block to be deallocated. |
| RETURNS | None. |

### 2.6.5.3 ix_ossl_memcpy()

SYNTAX

```
void* ix_ossl_memcpy(
        void* arg_pDest,
        const void* arg_pSrc,
        ix_ossl_size_t arg_Count);
```

| | |
|---|---|
| DESCRIPTION | This function will copy memory bytes between buffers. The ix_ossl_memcpy function copies count bytes of arg_pSrc to arg_pDest. If the source and destination overlap, this function does not ensure that the original source bytes in the overlapping region are copied before being overwritten. |
| PARAMETERS: | INOUT arg_pDest - destination buffer address.<br>IN arg_pSrc - source buffer address.<br>IN arg_Count - number of bytes to copy. |
| RETURNS | Returns the address of the destination buffer. |

### 2.6.5.4 ix_ossl_memset()

SYNTAX

```
void* ix_ossl_memset(
        void* arg_pDest,
        int arg_Char,
        x_ossl_size_t arg_Count);
```

| | |
|---|---|
| DESCRIPTION | This function sets buffers to a specified character. The ix_ossl_memset function sets the first arg_Count bytes of arg_pDest to the character arg_Char. |
| PARAMETERS: | INOUT arg_pDest - destination buffer address.<br>IN arg_pChar - character to set.<br>IN arg_Count - number of characters to set. |
| RETURNS | Returns the address of the destination buffer. |

## 2.6.6 Message Logging API

The Message Logging API provides functions to log error messages.

### 2.6.6.1 ix_ossl_message_log_init()

SYNTAX

```
ix_error ix_ossl_message_log_init(void);
```

| | |
|---|---|
| DESCRIPTION | This function is used to initialize the error message logging. For each OS, the messages are logged into an implementation dependent stream. This function should be called before any call to ix_ossl_message_log().<br><br>***Note:*** VXWORKS:IN ORDER TO USE MESSAGE LOGGING, THE PREPROCESSOR CONSTANT "INCLUDE_LOGGING" MUST BE SET DURING COMPILATION. THIS WILL CAUSE THE ROOT TASK usrRoot() IN usrConfig.c TO CALL logInit(). |
| PARAMETERS: | None. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | None. |

### 2.6.6.2 ix_ossl_message_log()

| | |
|---|---|
| SYNTAX | ix_error ix_ossl_message_log(const char * fmt_string, …); |
| DESCRIPTION | This routine is used to log a specified message. This routine's syntax is similar to printf() - a format string is followed by a variable number of arguments, which will be interpreted and formatted according to the fmt_string passed as the first argument. |
| PARAMETERS: | IN fmt_string - format string for the log message.<br><br>WIN32: The logs will be directed to stderr.<br><br>VXWORKS: This function encapsulates logMsg in the VxWorks logLib library. This function allows messages to be output even from ISRs. The logs are directed to stderr. |
| RETURNS | IX_SUCCESS if successful or a valid ix_error token for failure. |
| ERRORS | IX_OSSL_ERROR_MESSAGE_LOG_TOO_MANY_ARGS<br>IX_OSSL_ERROR_MESSAGE_LOG_FORMAT_STRING<br>IX_OSSL_ERROR_MESSAGE_UNSUPPORTED_DATA_TYPE |

## 2.7 Workbench/Transactor OSAPI

The Workbench/Transactor OSAPI has limited functions and is used for workbench and Transactor back-end. IXA SDK 3.0 will continue to provide this additional API for the backward compatibility with IXA SDK 2.0 OSAPI calls used by Transactor. We recommend non-workbench applications to use the OSSL API presented above.

**Table 2-5. Workbench/Transaction API Summary**

| API Function | Description |
|---|---|
| Thread_Create | Creates a thread. |
| Thread_Cancel | Deletes a thread. |
| Thread_SetPriority | Changes a thread priority. |
| Sem_Init | Creates a semaphore. |

**Table 2-5. Workbench/Transaction API Summary**

| API Function | Description |
|---|---|
| Sem_Destroy | Deletes a semaphore. |
| Sem_Wait | Waits on a semaphore to be available. |
| Sem_Post | Releases a semaphore. |
| delayMs | Provides milliseconds delay. |
| delayTick | Provides system tick delay. |

## 2.7.1 Transactor Thread API

### 2.7.1.1 Thread_Create()

SYNTAX

```
thread_T    Thread_Create(
             int     priority,
             void   (*startFunc)(threadArg_T *),
             threadArg_T *arg);
```

DESCRIPTION Creates a thread and execute the function pointed to by startFunc.

PARAMETERS: IN priority - The priority must be one of the following: HIGH_PRIORITY, MID_PRIORITY, or LOW_PRIORITY

IN - startFunc - Pointer to a start thread function.

IN arg - Argument to be passed to the start function.

RETURNS Returns the status of the operation.

### 2.7.1.2 Thread_Cancel()

SYNTAX int Thread_Cancel(thread_T thread);

DESCRIPTION Deletes a thread and return its resources back to the system.

PARAMETERS: IN thread - Thread to delete.

RETURNS Returns 0 always.

### 2.7.1.3 Thread_SetPriority()

SYNTAX int Thread_SetPriority(thread_T thread, int priority);

DESCRIPTION Changes the priority of a thread.

PARAMETERS: IN priority - The priority must be one of the following: HIGH_PRIORITY, MID_PRIORITY, or LOW_PRIORITY.

IN thread - The thread whose priority is to be changed.

| | |
|---|---|
| RETURNS | Returns 0 always. |

## 2.7.2 Transactor Semaphore API

## 2.7.3 Sem_Init()

| | |
|---|---|
| SYNTAX | sem_T Sem_Init(unsigned int initState); |
| DESCRIPTION | Creates and initializes a semaphore. Returns a value of type sem_T. |
| PARAMETERS: | IN initState - The initState parameter indicates the initial state of a counting semaphore. On OSs that do not support native counting semaphores they are implemented using mutexes. InitState should not be more than MAX_SEM_STATE, defined in sem.h. |
| RETURNS | Sem_T the semaphore value. |

### 2.7.3.1 Sem_Destroy()

| | |
|---|---|
| SYNTAX | int Sem_Destroy(sem_T semaphore); |
| DESCRIPTION | Deletes a semaphore that was created by a call to Sem_Init. |
| PARAMETERS: | IN semaphore - The semaphore to delete. |
| RETURNS | Returns 0 always. |

### 2.7.3.2 Sem_Wait()

| | |
|---|---|
| SYNTAX | int   Sem_Wait(sem_T semaphore, unsigned int timeout); |
| DESCRIPTION | Waits for the specified semaphore to be available and takes exclusive control of it. If the semaphore has not been released, then this call blocks for the number of milli-second specified by the timeout parameter. If the semaphore is not released within the specified timeout period, then the call fails and returns SEM_TIMEDOUT. |
| PARAMETERS: | IN semaphore - The semaphore to take control of after it is released.<br>IN timeout - The number of milliseconds to wait for the semaphore to be released. |
| RETURNS | SEM_SUCCESS: Successfully completed.<br>SEM_TIMEOUT: Completed with timeout.<br>SEM_FAILED: Completed with failure. |

### 2.7.3.3 Sem_Post()

| | |
|---|---|
| SYNTAX | int Sem_Post(sem_T semaphore); |
| DESCRIPTION | Releases exclusive control of semaphore and make it available to a waiting thread process. |
| PARAMETERS: | IN semaphore - The semaphore to release. |
| RETURNS | Returns the status of the operation. |

## 2.7.4 Transactor Delay API

### 2.7.4.1 delayMs()

| | |
|---|---|
| SYNTAX | void delayMs(unsigned int msec); |
| DESCRIPTION | Delays the current process by the number of millisecond specified by the msec parameter. |
| PARAMETERS: | IN msec - Number of milliseconds to delay. |
| RETURNS | None. |

### 2.7.4.2 delayTick()

| | |
|---|---|
| SYNTAX | void delayTick(unsigned int tick); |
| DESCRIPTION | Delays the current process by the number of system clock ticks specified by the tick parameter. |
| PARAMETERS: | IN tick - Number of system clock ticks to delay. |
| RETURNS | None. |

# Development Tools API 3

## 3.1 Hardware Debug Library

The network processor Debug Library allows you to debug Intel XScale® core code. The Developer Workbench application uses this library to provide a complete development environment on the network processor hardware and the Transactor.

The microengine debug component is a library of APIs that runs on the Intel XScale® core target and is primarily used by the Workbench to control and examine microengine states. This library is independent of any communication method/transport between itself and the Workbench and could be used for any debug-agent. Also, the debug library relies on the microengine driver to provide the necessary interfaces to access hardware resources.

The Debug library provides the ability to:

- Start, pause, resume, and stop microengines
- Insert breakpoints
- Single-step instructions
- Examine or modify microengine states

The parameters of the Debug library functions are checked for the appropriate values based on the version and the mode of the microengine. If any of the parameter values are inappropriate, or if the function fails, then an error value is returned.

Parameter range for MEv2:

Table 3-1. Valid Parameter Ranges

| Function Parameter Name | Valid Range |
|---|---|
| me | IXP2800 - (0x00:0x07, 0x16:0x23)<br>IXP2400 - (0x00:0x03, 0x10:0x13 |
| MAX_ME_NUM | 0x17 |
| ctx | 0:7 (8-context mode)<br>0,2,4,6 (4-context mode) |
| numCtx | 8 |
| upc/uAddr | 0:4095 |
| numUwords | 1:4096 |
| nnRegNum | 0:127 |
| nnNum | 1:128 |
| lmAddr | 0:639 |
| lmNum | 1:640 |

**Table 3-1. Valid Parameter Ranges (Continued)**

| Function Parameter Name | Valid Range |
|---|---|
| camId | 0:15 |
| inpState | 0:14 |
| gprRegNum | 0:15 |

## 3.1.1 Compiler Directives

Compiler directives support multiple chips and steppings, and operating systems. These directives can be specified on the compile command line (using -D) or within the source code (using #define) and should be used to isolate sections of code that pertain only to a particular IXP chip/rev and operating system.

### 3.1.1.1 `IOSTYLE`

The `IOSTYLE` directive indicates that the source to be compiled for use with the transactor or on the hardware. If this directive is used, then one of the following values must be assigned.

#### Values

| | |
|---|---|
| `XACTIO` | IO through transactor APIs |
| `CMBIO` | IO through transactor Core Memory Bus |
| `HARDWARE` | IO directly to hardware memory |

#### Example Usage

```
DIOSTYLE=XACTIO
```

### 3.1.1.2 `OS`

The `OS` directive specifies the Operating System on which the code is being generated.

#### Values

| | |
|---|---|
| `VXWORKS` | VxWorks* (hardware environment) |
| `WIN_32` | Microsoft* Windows (transactor environment) |
| `UNIX` | Unix (generic, for transactor or hardware) |
| `LINUX` | Linux (transactor/hardware) |

#### Example Usage

`DOS=VXWORKS`

### 3.1.1.3 `IXP_FAMILY`

The `IXP_FAMILY` directive indicates the IXP chip type.

#### Values

| | |
|---|---|
| IXP2000 | This value is used with the IXP2400 and IXP2800 chips |

### 3.1.1.4 `IXP_PROD_TYPE`

The `IXP_PROD_TYPE` directive indicates the specific product type.

#### Values

| | |
|---|---|
| IXP2400 | This value is used with the IXP2400 chip. |
| IXP2800 | This value is used with the IXP2800 chip. |

### 3.1.1.5 `IXP_MAJ_REV, IXP_MIN_REV`

The `IXP_MAJ_REV` and `IXP_MIN_REV` defines the stepping of the specific product.

#### Values

`REV_0`

#### Example Usage

`DIXP_MAJ_REV=A3`

## 3.1.2 Breakpoint Restrictions

- Breakpoints occur on a per -microengine basis and not per thread. You cannot debug one thread without affecting the others.
- Breakpoints are not allowed on statements that immediately precede a branch statement.
- Breakpoints are not allowed in a branch defer shadow.
- A microengine must be out of reset and paused before you instal breakpoints or access microstore locations.
- Fourteen microstore locations must be available per breakpoint.

### 3.1.3 MicroDebug Control Interface

Table 3-2 shows the micro debug control API.

**Table 3-2. Debug Control API**

| Function | Description |
|---|---|
| dbgMe_ AttachLib() | Initialize the debug library. |
| dbgMe_ DeleLib() | Delete the debug library object. |
| dbgMe_DefineFreeUstore() | Define the microstore free region for the specified microengines. |
| dbgMe_UpdateFreeUstore() | Update the debug-library free-ustore-region values. |
| dbgMe_Reset() | Reset microengine(s). |
| dbgMe_ClrReset() | Take the microengine(s) out of reset. |
| dbgMe_Start() | Start microengines threads from uPC 0. |
| dbgMe_Pause() | Stop microengines at earliest context arbitration. |
| dbgMe_PauseMeCtx() | Stop the microengines contexts, indicated by the `meMask`, and `ctxMask` parameters |
| dbgMe_ResumeMeCtx() | Resume the specified microengines contexts, indicated by the `meMask`, and `ctxMask` parameters |
| dbgMe_Resume() | Start microengines from current uPC. |
| dbgMe_PutMeCsr() | Write a long-word value to the microengine CSR specified by the me and csr parameters respectively. |
| dbgMe_GetMeCsr() | Read a long-word from the microengine CSR specified by the me and csr parameters respectively and store it in value. |
| dbgMe_PutCsr() | Write the long-word data to the CSR specified by the csrAddr parameter. |
| dbgMe_GetCsr() | Read the long-word from the CSR specified by the csrAddr parameter and return in data. |
| dbgMe_WriteMem32() | Write a longword to DRAM and SCSRATCH or SRAM. |
| dbgMe_ReadMem32() | Read a longword from DRAM and SCRATCH or SRAM. |
| dbgMe_GetLM() | Obtain count longwords from a microengine Local Memory starting at lmAddr and store them in lmData. |
| dbgMe_PutLM() | Write count longwords from lmData to microengine Local Memory starting at lmAddr. |
| dbgMe_GetCAM() | Read the contents of the microengine CAM and store the tag value, lock bits, and LRU information in camStat. |
| dbgMe_PutCAM() | Write content of camStat to the microengine CAM. In order to maintain the LRU information being written, the camStat->lru[x] with the lowest value is written first, then the next lowest, and so on. |
| dbgMe_IsInpStateSet() | This routine determines if the microengine is in the specified inpState, and return a value indicating whether it's set, not set, or an error code. |
| dbgMe_IsMeEnabled() | Determines whether the microengines are active. |
| dbgMe_GetMeRunState | Determines which of the MEs that are specified in meMask are enabled or active. |
| dbgMe_GetMEsEvents | Get an array of dbgMe_MeEvents_T for specified Microengines. |

**Table 3-2. Debug Control API (Continued)**

| Function | Description |
|---|---|
| dbgMe_GetMEsStatus | Get an array of dbgMe_MeStatus_T for specified microengines. |
| dbgMe_GetMEsPCs | Get an array of dbgMe_MePCs_T for specified microengines. |
| dbgMe_PutMeCtxPCs() | Set the context-program-counters of the specified micro-engines to the same ustore location indicated by the upc parameter. |
| dbgMe_GetMeCtxPCs() | Obtain the program-counter of the indicated microengine context and returned the value in pc. |
| dbgMe_PutMePCs() | Sets the program counters of the specified microengine to the values in the array `upc` argument |
| dbgMe_GetMePCs() | Obtains the program-counters of the specified ME and stores them in the array `upc` argument. |
| dbgMe_PutUwords() | Write longwords to microengine microstore. |
| dbgMe_GetUwords() | Read longwords from microengine microstore. |
| dbgMe_PutRelDataReg() | Write longwords to relative Data Registers. |
| dbgMe_GetRelDataReg() | Read longwords from relative Data Registers. |
| dbgMe_PutAbsDataReg() | Writes num longwords from `regData` to the microengine data register starting at `regNum`. |
| dbgMe_GetAbsDataReg() | Obtains num longwords from a microengine data register starting at `regNum` |
| dbgMe_PutTbuf() | Writes num longwords from data to `TBUF` starting at the address location specified |
| dbgMe_GetRbuf() | Reads num longwords from RBUF byte location specified |
| dbgMe_PutTbufEleCntl() | Write a quadword from bufEleCtl to the RBUF element-control specified by eleNum. |
| dbgMe_GetWakeupEvents() | Reads the wakeup-events of the specified microengine context. |
| dbgMe_PutWakeupEvents() | Writes the wakeup-events of the specified microengine context. |
| dbgMe_GetSigEvents() | Reads the signal-events of the specified microengine context. |
| dbgMe_PutSigEvents() | Writes the signal-events of the specified microengine context. |
| dbgMe_GetSysMemSize() | Obtain the system memory sizes. |
| dbgMe_GetMeStatus() | Obtain information on microengine. |
| dbgMe_SetBreakpoint() | Set a breakpoint. |
| dbgMe_ShowBreakpoints() | List the current breakpoints. |
| dbgMe_ClearBreakpoint() | Clear a breakpoint |
| dbgMe_ClearAllBreakpoints() | Clear all breakpoints. |
| dbgMe_Step() | This function is implemented using the “Inline break, Inline continue” method described above for stepping the microengines. |

## 3.1.4 Defined Types, Enumerations, and Data Structures

### 3.1.4.1 `DBGME_MAX_BREAKPOINTS`

The maximum number of outstanding breakpoints that can be handled by the library.

### C Syntax

```
#define DBGME_MAX_BREAKPOINTS 256
```

### 3.1.4.2 dbgMe_uStoreFree_T

The `dbgMe_uStoreFree_T` structure is used for expressing the available microstore. The start and size of each microengine's free-ustore region is specified by `freeUpc` and `freeSize` respectively.

### C Syntax

```
typedef struct{
   unsigned short freeUpc[MAX_NUM_ME];
   unsigned short freeSize[MAX_NUM_ME];
}dbgMe_uStoreFree_T;
```

### 3.1.4.3 uword_T

The `uword_T` type describes the ustore value and is eight bytes in length.

### C Syntax

```
typedef uint64 uofUword_T;
```

### 3.1.4.4 ixp_RegType_T

The `ixp_RegType_T` type describes the general purpose register, transfer register, signal, neighbor, and local memory registers.

### C Syntax

```
typedef enum{
    IXP_NO_DEST,
    IXP_GPA_REL,
    IXP_GPA_ABS,
    IXP_GPB_REL,
    IXP_GPB_ABS,
    IXP_SR_REL,
    IXP_SR_RD_REL,
    IXP_SR_WR_REL,
    IXP_SR_ABS,
    IXP_SR_RD_ABS,
    IXP_SR_WR_ABS,
    IXP_SR0_SPILL,
    IXP_SR1_SPILL,
    IXP_SR2_SPILL,
    IXP_SR3_SPILL,
    IXP_SR0_MEM_ADDR,
    IXP_SR1_MEM_ADDR,
    IXP_SR2_MEM_ADDR,
    IXP_SR3_MEM_ADDR,
    IXP_DR_REL,
    IXP_DR_RD_REL,
    IXP_DR_WR_REL,
    IXP_DR_ABS,
    IXP_DR_RD_ABS,
    IXP_DR_WR_ABS,
    IXP_DR_MEM_ADDR,
    IXP_LMEM,
    IXP_LMEM0,
    IXP_LMEM1,
    IXP_LMEM_SPILL,
    IXP_LMEM_ADDR,
    IXP_NEIGH_REL,
    IXP_NEIGH_INDX,
    IXP_SIG_REL,
    IXP_SIG_INDX,
    IXP_SIG_DOUBLE,
    IXP_SIG_SINGLE,
    IXP_SCRATCH_MEM_ADDR,
    IXP_ANY_REG = 0xffff
}ixp_RegType_T;
```

### 3.1.4.5 dbgMe_Bkpt_T

This structure is used for setting and retrieving breakpoint information.

#### C Syntax

```
typedef struct dbgMe_Bkpt{
void (*callback)(ushort bkptId, uchar me, uchar ctx, ushort uPc, void
usrData);
uchar ctx;                // out: ctx at breakpoint
ushort id;                // out: the assigned breakpoint ID
ushort uAddr;             // microstore address
void *usrData;            // pointer to user specific data
uint brkIfCtxMask;        // break if equal to ctx
uint stpOnBrkMeMask;      // uengines to stop on break
uint me;                  // microengine at breakpoint
} dbgMe_Bkpt_T;
```

### 3.1.4.6 dbgMe_BkptArray_T

This structure is used to return a list of breakpoints.

#### C Syntax

```
typedef struct dbgMe_BkptArray{
    ushort len;// number of elements in array
    dbgMe_Bkpt_T bkpts[DBGME_MAX_BREAKPOINTS];
} dbgMe_BkptArray_T;
```

### 3.1.4.7 dbgMeInfo_SrcLine_T, dbgMeInfo_SrcImage_T, dbgMeInfo_Image_T

The dbgMeInfo_SrcLine_T, dbgMeInfo_SrcImage_T, and dbgMeInfo_Image_T structures contain information regarding whether the source line is valid for setting a breakpoint, the number of defer statements, and the branch-target address.

#### C Syntax

```
typedef struct{
char brkPtAllowed;                   // 0=not allowed, 1=allowed
uchar deferCount;                    // num defer instructions; 0 = normal
short brAddr;                        // branch address; -1 = none
int regAddr;                         // microengine reg type & address; -1 =
none
}dbgMeInfo_SrcLine_T;

typedef struct {
    uint meAssigned;                 // bit-flags of the assigned microengines
    ushort reserved1;                // reserved for future use
    ushort numSrcLine;               // the number of line pointed to by
srcLine
    dbgMeInfo_SrcLine_T *srcLine;
}dbgMeInfo_SrcImage_T;

typedef struct{
    uint numImage;                   // num ptr in srcImage array
    dbgMeInfo_SrcImage_T  *srcImage[MAX_ME_NUM];
}dbgMeInfo_Image_T;
```

### 3.1.4.8 dbgMe_MeStatus

The dbgMe_MeStatus structure is used to indicate the state of an microengine.

#### C Syntax

```
typedef struct{
    char state;               // active, inactive, break
    uchar ctx;                // active context
    uchar enables;            // cc<5>, parity<4>, ctxMode<3>, nnMode<2>,
                              // lmAddr1<1>, lmAddr0<0>
    ushort breakId;           // breakpoint ID
    ushort ctxEnabled;        // contexts that are enabled
    ushort activePC;          // active PC
    uint profileCnt;          // profile count
    uint timeStampLow;        // low 32 bits of the timestamp
    uint timeStampHigh;       // high 32 bits of the timestamp
}dbgMe_MeStatus_T;
```

### 3.1.4.9 dbgMe_MeCamStatus

The `dbgMe_MeCamStatus` structure is used to read and write the microengine CAM.

#### C Syntax

```
typedef struct{
    uint tag[16];
    uchar status[16];
    uchar lru[16];
}dbgMe_MeCamStatus_T;
```

### 3.1.4.10 dbgMe_MemConfig_T

This structure is used to provide OS memory configuration information to the user.

#### C Syntax

```
typedef struct{
uint sys;             // Size of memory usable by the OS
uint dram;            // Size of memory mapped at DRAM and usable by HW
uint sram0;           // Size of SRAM Bank0
uint sram1;           // Size of SRAM Bank1
uint sram2;           // Size of SRAM Bank2
uint sram3;           // Size of SRAM Bank3
}dbgMe_MemConfig_T;
```

### 3.1.4.11 Hal_Me_CSR_T

These values represent the microengine CSRs addresses and are used to describe the CSR being acted on by the `dbgMe_GetMeCsr` and `dbgMe_PutMeCsr` functions.

#### C Syntax

```
typedef enum{
    USTORE_ADDRESS                  = 0x000,
    USTORE_DATA_LOWER               = 0x004,
    USTORE_DATA_UPPER               = 0x008,
    USTORE_ERROR_STATUS             = 0x00c,
    ALU_OUT                         = 0x010,
    CTX_ARB_CNTL                    = 0x014,
    CTX_ENABLES                     = 0x018,
    CC_ENABLE                       = 0x01c,
    CSR_CTX_POINTER                 = 0x020,
    CTX_STS_INDIRECT                = 0x040,
    ACTIVE_CTX_STATUS               = 0x044,
    CTX_SIG_EVENTS_INDIRECT         = 0x048,
    CTX_SIG_EVENTS_ACTIVE           = 0x04c,
    CTX_WAKEUP_EVENTS_INDIRECT      = 0x050,
    CTX_WAKEUP_EVENTS_ACTIVE        = 0x054,
    CTX_FUTURE_NUM_INDIRECT         = 0x058,
    CTX_FUTURE_NUM_ACTIVE           = 0x05c,
        LM_ADDR_0_INDIRECT            = 0x060,
        LM_ADDR_0_ACTIVE              = 0x064,
        LM_ADDR_1_INDIRECT            = 0x068,
        LM_ADDR_1_ACTIVE              = 0x06c,
        BYTE_INDEX                    = 0x070,
        ACTIVE_LM_ADDR_0_BYTE_INDEX   = 0x0e0,
        INDIRECT_LM_ADDR_0_BYTE_INDEX= 0x0e4,
        INDIRECT_LM_ADDR_1_BYTE_INDEX= 0x0e8,
        ACTIVE_LM_ADDR_1_BYTE_INDEX   = 0x0ec,
        T_INDEX_BYTE_INDEX            = 0x0f4,
        T_INDEX                       = 0x074,
        FUTURE_NUM_SIGNAL_INDIRECT    = 0x078,
        FUTURE_NUM_SIGNAL_ACTIVE      = 0x07c,
        NN_PUT                        = 0x080,
        NN_GET                        = 0x084,
        TIMESTAMP_LOW                 = 0x0c0,
        TIMESTAMP_HIGH                = 0x0c4,
        INSTRUCTION_SIGNATURE         = 0x0c8,
        NEXT_NEIGHBOR_SIGNAL          = 0x100,
        PREV_NEIGHBOR_SIGNAL          = 0x104,
        SAME_ME_SIGNAL                = 0x108,
        CRC_REMAINDER                 = 0x140,
        PROFILE_NUM                   = 0x144,
        PSEUDO_RANDOM_NUMBER          = 0x148,
        LOCAL_CSR_STATUS              = 0x180,
        NULL_CSR                      = 0x3fc
}Hal_Me_CSR_T;
```

### 3.1.5 API Functions

This section describes all of the debug library's C functions. The debug library must be initialized prior to accessing any of the microengine debug routines by invoking the dbgMe_ AttachLib()routine. Also, the microengine driver must be initialized prior to initializing this library.

#### 3.1.5.1 `dbgMe_ AttachLib()`

This routine initializes the microengine debug library and instantiates a debug object. Only one instance of the library is created. Therefore, if a library object exists, then it is deleted and its breakpoints copied to a new object—a new object is created in order to provide a unique handle. This routine is an enhancement of the `dbgMe_InitLib()` to provide re-connection, and some control over how the microengines are initialized.

The unused ustore for each microengine can be defined during initialization by specifying a `uStoreFree` parameter which is an array describing the free-ustore region for each microengine. If this parameter is NULL, then the microengine driver shall be interrogated to determine the free-microstore regions. To avoid library initialization-order dependency—whether the microcode was loaded prior to initializing the udebug library—the dbgMe_DefineFreeUstore(), and dbgMe_UpdateFreeUstore()routine are also provided.

The `dbgInfo` parameter points to debug information describing whether a breakpoint is allowed on a particular line, the number of defer instructions, and other debugging information. If this parameter is `NULL`, then breakpoint and *step* operations are not allowed.

The meMask parameter specifies the microengines that have their states restored/reset during initialization. This parameter is useful for attaching to an existing udebug instance without disturbing the state of the selected microengines.

#### C Syntax

```
int dbgMe_AttachLib(
    dbgMe_uStoreFree_T *uStoreFree,
    dbgMeInfo_Image_T *dbgInfo,
    uint meMask);
```

#### Returns

- DBGME_SUCCESS—the operation succeeded.
- DBGME_FAIL—the operation failed.

### 3.1.5.2 dbgMe_ DeleLib()

Removes the debug object and releases all associated resources. In order to prevent stranded breakpoints in the ustore, all breakpoints are deleted. If a breakpoint could not be deleted due to an active microengine, then the microengine is reset and the breakpoint deleted.

#### C Syntax

```
void dbgMe_DeleLib(void);
```

#### Returns

None

### 3.1.5.3 dbgMe_DefineFreeUstore()

Define the unused ustore region for the specified microengines. The targeted microengines are selected by setting the corresponding bits in meMask.

#### C Syntax

```
int dbgMe_DefineFreeUstore(
    uint meMask,
    uint freeUaddr,
    uint freeSize);
```

#### Returns

- DBGME_SUCCESS—the operation succeeded.
- DBG_BADARG—an invalid argument.

#### 3.1.5.4 `dbgMe_UpdateFreeUstore()`

Updates the free-ustore values with values obtained from the microengine driver. This function should be called if the ustore was loaded after the udebug library has been initialized. The appropriate microengines are selected by setting the corresponding bits in `meMask`.

##### C Syntax

```
void dbgMe_UpdateFreeUstore(uint meMask);
```

##### Returns

None

#### 3.1.5.5 `dbgMe_Reset()`

Reset the microengines. Microengines are selected by setting the corresponding bits in `uEngMask`. A zero value in the `clrReg` argument maintains the state of the registers; otherwise, the registers are restored to their power-on states.

##### C Syntax

```
int dbgMe_Reset(uint meMask, int clrReg);
```

##### Returns

- `DBGME_SUCCESS`—the operation succeeded.
- `DBGME_BADARG`—an invalid argument

#### 3.1.5.6 `dbgMe_ClrReset()`

This routine takes the specified microengines out of reset. Certain operations, for example reading from ustore, cannot be performed when the microengine is running or is in reset. The microengines are selected by setting the corresponding bits in `meMask`.

##### C Syntax

```
int dbgMe_ClrReset(uint meMask)
```

##### Returns

- `DBGME_SUCCESS`—the operation succeeded.
- `DBGME_BADARG`—an invalid argument.

#### 3.1.5.7 `dbgMe_Start()`

Enable all contexts of the specified microengines to run starting with context-0 and from ustore address zero. Because the uPC must be written, the microengines should be inactive. If any of the specified microengines are active, then the function returns a value indicating the failure. Setting the corresponding bits in `meMask` specifies the microengines.

#### C Syntax

```
int dbgMe_Start(uint meMask);
```

#### Returns

- DBGME_SUCCESS—the operation succeeded.
- DBGME_ENABLED—one or more of the selected microengine are active.
- DBGME_BADARG—bad function argument.
- DBGME_BADLIB—bad debug library—not initialized.

### 3.1.5.8 `dbgMe_Pause()`

Stop all threads of the selected microengines at the next context arbitration, and save relevant information to allow them to resume using the dbgMe_Resume()function. The microengines are placed in an inactive state allowing the reading and writing of ustore, CSRs, and general purpose registers. The microengines are selected by setting the corresponding bits in `meMask`. The delay parameter indicates the maximum number of milliseconds to wait for the microengines to be paused and a fail status is returned if any microengine fails to stopped within the desired time.

#### C Syntax

```
int dbgMe_Pause(uint meMask, uint delay);
```

#### Returns

- `DBGME_SUCCESS`—the operation succeeded.
- `DBGME_FAIL`—the operation failed.
- `DBGME_BADARG`—an invalid argument passed to function.

### 3.1.5.9 `dbgMe_PauseMeCtx()`

Stops the microengines contexts indicated by the `meMask`, and `ctxMask` parameters, at the next context arbitration. The microengines and contexts are selected by setting the corresponding bits in `meMask`, and `ctxMask`.

#### C Syntax

```
int dbgMe_PauseMeCtx(uint meMask, uint ctxMask);
```

#### Returns

- `DBGME_SUCCESS`—the operation succeeded.
- `DBGME_BADARG`—an invalid argument

### 3.1.5.10 `dbgMe_ResumeMeCtx()`

Resumes the specified microengines conexts, indicated by the `meMask`, and `ctxMask` parameters, from the current PCs. The microengines and contexts are selected by setting the corresponding bits in `meMask`, and `ctxMask`.

#### C Syntax

```
int dbgMe_ResumeMeCtx(uint meMask, uint ctxMask);
```

#### Returns

- `DBGME_SUCCESS`—the operation succeeded.
- `DBGME_BADARG`—an invalid argument

### 3.1.5.11 `dbgMe_Resume()`

Resume normal execution of all threads of the selected microengines; starting from the current program-counter. Threads that have been disabled via `ctx_arb[kill]` instruction are not resumed by this function. Microengines are selected by setting the corresponding bits in `meMask`.

#### C Syntax

```
int dbgMe_Resume(uint meMask);
```

#### Returns

- `DBGME_SUCCESS`—the operation succeeded.
- `DBGME_BADARG`—an invalid argument

### 3.1.5.12 `dbgMe_PutMeCsr()`

Write a long-word value to the microengine CSR specified by the `me` and CSR parameters respectively. Because the Local CSRs are single-ported, the microengine wins arbitration if the XScale™ and microengine attempt to access the CSR on a given cycle. Therefore, this function checks the `Local_CSR_Status` register and retry until the operation is successful. If the function succeeds, a zero value is returned, otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_PutMeCsr(uchar me, Hal_Me_CSR_T csr, uint value);
```

#### Returns

- Zero—the operation succeeded.
- Non-zero—the operation failed.

### 3.1.5.13 `dbgMe_GetMeCsr()`

Read a long-word from the microengine CSR specified by the `me` and `csr` parameters respectively and store it in value. Because the Local CSRs are single-ported, the microengine wins arbitration if the XScale™ and microengine attempts to access the CSR on a given cycle. Therefore, this function checks the `Local_CSR_Status` register and retries until the operation is successful. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetMeCsr(uchar me, Hal_Me_CSR_T csr, uint * value);
```

#### Returns

- 0—the operation succeeded.
- Non-zero—the operation failed.

### 3.1.5.14 `dbgMe_PutCsr()`

Write the long-word data to the CSR specified by the XAddr_Hi and XAddr_Lo parameters. The function does not verify the validity of the CSR address or whether is correctly maps to a valid CSR; it is provided as a convenient mechanism for the WorkBench, or other remote services, to access ISP CSRs. Therefore, it is the responsibility of the called to ensure that the address is valid.

#### C Syntax

```
void dbgME_PutCsr(uint Xaddr_Hi, uint XAddr_Lo, uint data);
```

#### Returns

None

### 3.1.5.15 `dbgMe_GetCsr()`

Read the long-word fro the CSR specified by he XAddr_Hi and XAddr_Lo parameters and return in data. The function does not verify the validity of the CSR address or whether is correctly maps to a valid CSR; it is provided as a convenient mechanism for the WorkBench, or other remote services, to access ISP CSRs. Therefore, it is the responsibility of the called to ensure that the address is valid. If the function succeeds, a zero value is returned, otherwise, a nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgME_GetCsr(uint Xaddr_Hi, uint XAddr_Lo, uint *data);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.16 dbgMe_WriteMem32()

Write count longwords from data to the physical memory region starting at the byte-addressable memory location specified by `addr`. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgME_WriteMem32(
    dbgMe_MemRegion_T memRegion,
    uint addr, uint *data,
    uint count);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.17 dbgMe_ReadMem32()

Reads count longwords from the byte-addressable physical memory region specified by `addr` and store in data. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgME_ReadMem32(
    dbgMe_MemRegion_T memRegion,
    uint addr,
    uint *data,
    uint count);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.18 dbgMe_GetLM()

Obtains num longwords from `me` Local Memory starting at `lmAddr` and store them in `lmData`. If the function succeeds, a zero value is returned; otherwise, a nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetLM(uchar me, uint lmAddr, uint *lmData, uint num);
```

#### Returns

- Zero–the operation was successful.

- Non-zero–the operation failed.

## 3.1.5.19 `dbgMe_PutLM()`

Write `num` longwords from `lmData` to `me` Local Memory starting at `lmAddr`. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

### C Syntax

```
int dbgMe_PutLM(uchar me, uint lmAdr, uint *lmData, uint num);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

## 3.1.5.20 `dbgMe_GetCAM()`

Read the contents of the microengine CAM and store the tag value, status bits in `camStat`. The `camStat->lru` is undefined. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

### C Syntax

```
int dbgMe_GetCAM(uchar me, dbgMe_MeCamStatus_T *camStat);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

## 3.1.5.21 `dbgMe_PutCAM()`

Writes content of `camStat` to the microengine CAM. The LRU—that is, the Least Recently Used list—maintains a time-ordered list of CAM usage. When an entry is loaded—or matches on a lookup—it is marked as MRU—that is, as Most Recently Used. Therefore, in order to maintain the LRU information being written, the `camStat->lru[x]` with the lowest value is written first, then the next lowest, and so on. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

### C Syntax

```
int dbgMe_PutCAM(uchar me, dbgMe_MeCamStatus_T *camStat);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.22 dbgMe_IsInpStateSet()

This routine determines if the microengine specified by `me` is in the state specified by `inpState`, returning a value indicating whether it is set, not set, or an error code. There are 15 values of `inpState`—that is, zero through fourteen—and their meaning is specific to the chip implementation. Please see `BR_INP_STATE` instruction in the network processor's Programmer's Reference Manual. for a description and values of states.

#### C Syntax

```
int dbgMe_IsInpStateSet (uchar me, uchar inpState);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.23 dbgMe_IsMeEnabled()

This routine determines if all of the microengines that are specified in the `meMask` parameter are active. A value is returned indicating whether one or all of the selected microengines were active, whether all the microengines were inactive, or a value of minus one if the routine failed.

#### C Syntax

```
int dbgMe_IsMeEnabled(uint meMask);
```

#### Returns

- `DBGME_ENABLED`–all the specified microengines are active.
- `DBGME_DISABLED`–one or all of the specified microengines are inactive or the debug library/class is not initialized.
- `DBGME_RESET`–one or all microengines are in reset.

### 3.1.5.24 dbgMe_GetMeRunState

Determines which of the MEs that are specified in meMask are enabled or active.

The MEs that are either enabled, or active will be returned in the enabledMask and the activeMask parameter respectively.

#### C Syntax

```
int dbgMe_GetMeRunState(uint meMask, uint *enabledMask, uint *activeMeMask,);
```

#### Returns

If the function succeeds, DBGME_SUCCESS will be returned, otherwise, nonzero value indicating the failure will be returned.

- DBGME_SUCCESS–the operation succeeded.
- DBGME_BADARG–bad function argument.
- DBGME_BADLIB–bad debug library—not initialized.

### 3.1.5.25 dbgMe_GetMEsEvents

Get an array of dbg_MeEvents_T for the MEs defined by meMask. A db_MeEvents_T structure is returned in the MeEvents array for each microengine found in mesMask, with the first structure returned for the first microengine found, the second structure for the second microengine found, and so on. The asize parameter must be set to the size of the dbg_MeEvents[] array.

#### C Syntax

```
int dbgMe_GetMEsEvents(uint mesMask, dbg_MeEvents_T *MeEvents, uint asize);
```

#### Returns

If the function succeeds, DBGME_SUCCESS is returned; otherwise, a nonzero value indicating the failure is returned.

- DBGME_SUCCESS–the operation succeeded.
- DBGME_BADARG–bad function argument.
- DBGME_BADLIB–bad debug library—not initialized.

### 3.1.5.26 dbgMe_GetMEsStatus

Get an array of dbgMe_MeStatus_T for the microengines defined by mesMask. A dbgMe_MeStatus_T structure is returned in the MeStatus array for each microengine found in mesMask, with the first structure returned for the first mircoengine found, the second structure for the second microengine found, and so on. The asize parameter must be set to the size of the dbgMe_MeStatus_T array.

#### C Syntax

```
int dbgMe_GetMEsStatus(uint mesMask, dbg_MeStatus_T *MeStatus, uint asize);
```

#### Returns

If the function succeeds, DBGME_SUCCESS is returned; otherwise, a nonzero value indicating the failure is returned.

- DBGME_SUCCESS–the operation succeeded.
- DBGME_BADARG–bad function argument.
- DBGME_BADLIB–bad debug library—not initialized.

### 3.1.5.27 dbgMe_GetMEsPCs

Get an array of dbgMe_MePCs_T for the microengines defined by mesMask. A dbgMe_MePCs_T structure is returned in the MePCs array for each microengine found in mesMask, with the first structure returned for the first mircoengine found, the second structure for the second microengine found, and so on. The asize parameter must be set to the size of the dbgMe_MeStatus_T array.

#### C Syntax

```
int dbgMe_GetMEsPCs(uint mesMask, dbg_MePC_T *MePCs, uint asize);
```

#### Returns

If the function succeeds, DBGME_SUCCESS is returned; otherwise, a nonzero value indicating the failure is returned.

- `DBGME_SUCCESS`–the operation succeeded.
- `DBGME_BADARG`–bad function argument.
- `DBGME_BADLIB`–bad debug library—not initialized.

### 3.1.5.28 dbgMe_PutMeCtxPCs()

Set the context-program-counters of the specified micro-engines to the same microstore location indicated by the `upc` parameter. The microengines must be out of reset and inactive before setting the program counters. Setting the corresponding bits in `meMask` and `ctxMask` respectively specifies the microengines and contexts. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_PutMeCtxPCs(uint meMask, uint ctxMask, uint upc);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.29 dbgMe_GetMeCtxPCs()

Obtains the program-counter of the microengine context `me` and returns the value in `pc`. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetMeCtxPC(uchar me, uchar ctx, uint *upc);
```

#### Returns

- Zero–the operation was successful.

- Non-zero–the operation failed.

### 3.1.5.30 `dbgMe_PutMePCs()`

Sets the program counters of the specified microengine to the values in the array `upc` argument. The microengines must be out of reset and inactive before setting the program counters. The `upc` array must be large enough to accommodate numCtx—eight for MEv2—elements. This function writes to all the PCs regardless of whether the microengine is in the eight-context or four-context mode. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_PutMePCs(uchar me, uint *upc);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.31 `dbgMe_GetMePCs()`

Obtains the program counters of the specified microengine and stores them in the array `upc` argument. The `upc` array must be large enough to accommodate `numCtx`—eight for MEv2—elements. This function retrieves all the PCs regardless of whether the microengine is in the eight-context or four-context mode. If the function succeeds, a zero value is returned; otherwise, a nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetMePCs(uchar me, uint *upc);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.32 `dbgMe_PutUwords()`

Write numUwords of longwords from the `uWord` value to the ustore location, starting at `uAddr`, of the microengines with a corresponding bit set in `meMask`. This function assumes that each `uword` occupies one element of the `uWord` parameter and that only the lower five bytes are utilized. The microengines must be out of reset and inactive before writing to the ustore. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_PutUwords
    uint meMask,
    uint uAddr,
    ushort numUwords,
    uword_T *uWord);
```

**Returns**

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.33 dbgMe_GetUwords()

Read numUwords longwords from the microengine ustore location, specified by me and uAddr respectively, to uWord. This function assumes that each uword occupies one element of the uWord parameter and that only the lower five bytes are utilized. The microengine must be out of reset and inactive before reading the ustore. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

**C Syntax**

```
int dbgMe_GetUwords(
    uchar me,
    uint uAddr,
    ushort numUwords,
    uword_T *uWord);
```

**Returns**

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.34 dbgMe_PutRelDataReg()

Write num longwords from regData to the microengine data register starting at regNum. The register number is context relative and must be within the range of that's appropriate for the microengine context mode. The regType parameter must either be one of the following: IXP_GPA_REL, IXP_GPB_REL, IXP_DR_RD_REL, IXP_SR_RD_REL, IXP_DR_WR_REL, IXP_SR_WR_REL, IXP_NEIGH_REL.

If the regType is IXP_NEIGH_REL, then the specified microengine‘s NN register is written and not its neighbor's. The microengine must be disabled before calling this function. If the function succeeds, DBGME_SUCCESS is returned, otherwise, nonzero value indicating the failure is returned.

**C Syntax**

```
int dbgMe_PutRelDataReg(
    uchar me,
    uchar ctx,
    ixp_RegType_T regType,
    ushort RegNum,
    uint *regData, uint num);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

#### 3.1.5.35 `dbgMe_GetRelDataReg()`

Obtains `num` longwords from a microengine data register starting at `regNum`. The register number is context relative and must be within the range suited for the microengine type and context mode. The `regType` parameter must be one of the following: `IXP_GPA_REL, IXP_GPB_REL, IXP_DR_RD_REL, IXP_SR_RD_REL, IXP_DR_WR_REL, IXP_SR_WR_REL, IXP_NEIGH_REL`.

The microengine must be disabled before calling this function. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

### C Syntax

```
int dbgMe_GetRelDataReg(
    uchar me,
    uchar ctx,
    ixp_RegType_T regType,
    ushort RegNum,
    uint *regData, uint num);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

#### 3.1.5.36 `dbgMe_PutAbsDataReg()`

Write num longwords from `regData` to the microengine data register starting at `regNum`.The register number is absolute and must be within the range suited for the microengine type. The `regType` parameter must be one of the following: `IXP_GPA_ABS, IXP_GPB_ABS, IXP_DR_RD_ABS, IXP_SR_RD_ABS, IXP_DR_WR_ABS, IXP_SR_WR_ABS, IXP_NEIGH_ABS`.

If the `regType` is `IXP_NEIGH_ABS`, then the specified microengine's NN register is written and not its neighbor's. The microengine must be disabled before calling this function. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

### C Syntax

```
    int dbgMe_PutAbsDataReg(
    uchar me,
    uchar ctx,
    ixp_RegType_T regType,
    ushort regNum,
    uint *regData, uint num);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

## 3.1.5.37 `dbgMe_GetAbsDataReg()`

Obtains `num` longwords from a microengine data register starting at `regNum`. The register number is absolute and must be within the range suited for the type of microengine. The `regType` parameter must be one of the following types: `IXP_GPA_ABS, IXP_GPB_ABS, IXP_DR_RD_ABS, IXP_SR_RD_ABS, IXP_DR_WR_ABS, IXP_SR_WR_ABS, IXP_NEIGH_ABS`. The microengine must be disabled before calling this function. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

### C Syntax

```
int dbgMe_GetAbsDataReg(
   uchar me,
   uchar ctx,
   ixp_RegType_T regType,
   ushort regNum,
   uint *regData, uint num);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

## 3.1.5.38 `dbgMe_PutTbuf()`

Writes `num` longwords from data to `TBUF` starting at the address location specified by `tbufByte`, (which must be aligned to 4 bytes). If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

### C Syntax

```
int dbgMe_PutTbuf(
   uint tbufByte,
   uint *data,
   uint num);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.39 `dbgMe_GetRbuf()`

Reads `num` longwords from `RBUF` byte location specified by `rbufByte` (which must be aligned to 4 bytes) to data. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetRbuf(
    uint rbufByte,
    uint *data,
    uint num);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.40 `dbgMe_PutTbufEleCntl()`

Write the `eleCtlA`, and `eleCtlB` long-words to the `TBUF` element-control specified by `eleNum`, (which must be 0 to 127). Refer to *IXP2000 Network Processor Programmer's Reference Manual* for the format and information about the `RBUF` control. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_PutTbufEleCntl(
    uint eleNum,
    uint eleCtlA,
    uint eleCtlB);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.41 `dbgMe_GetWakeupEvents()`

Reads the wakeup-events of the specified microengine context. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetWakeupEvents(
    uchar me,
    uchar ctx,
    uint *events);
```

### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.42 dbgMe_PutWakeupEvents()

Writes the wakeup-events of the specified microengine context. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_PutWakeupEvents(uchar me,
    uchar ctx,
    uint events);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.43 dbgMe_GetSigEvents()

Read the signal-events of the specified microengine context. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetSigEvents(uchar me,
    uchar ctx,
    uint *events);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.44 dbgMe_PutSigEvents()

Writes the signal-events of the specified microengine context. If the function succeeds, `DBGME_SUCCESS` is returned, otherwise, nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_PutSigEvents(uchar me, uchar ctx, uint events);
```

#### Returns

- Zero–the operation was successful.

- Non-zero–the operation failed.

### 3.1.5.45 `dbgMe_GetSysMemSize()`

Obtains the system memory configuration and return them in the `memConfig` parameter. If the function succeeds, a zero value is returned; otherwise, a nonzero value indicating the failure is returned.

#### C Syntax

```
int dbgMe_GetSysMemSize(dbgMe_MemConfig_T *memConfig);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.46 `dbgMe_GetMeStatus()`

Retrieves the state of the microengine and returns it in the `meStatus` parameter. If the function succeeds, a zero value is returned; otherwise, a non-zero value indicating the failure is returned.

#### C Syntax

```
int GetMeStatus(uchar me, dbgMe_MeStatus_T *meStatus);
```

#### Returns

- Zero–the operation was successful.
- Non-zero–the operation failed.

### 3.1.5.47 `dbgMe_SetBreakpoint()`

Sets a microengine breakpoint as defined by the breakpoint parameter. The routine implements the `Inline break, Non-inline continue` method, as describe above, for MEv2. This method was chosen because it uses the least amount of micro-store, and most of the limitations—*conditional breakpoints*, breaking on a context swap instruction, breaking on defer instruction—can easily be done.

#### C Syntax

```
int dbgMe_SetBreakpoint(dbgMe_Bkpt_T *breakpoint);
```

#### Returns

- `DBGME_SUCCESS`–the operation succeeded.
- `DBGME_BADARG`–bad function argument.
- `DBGME_BADLIB`–bad debug library—not initialized.

- DBGME_ENABLED–microengine active.
- DBGME_NODBG_INFO–no debug information.
- DBGME_RESET–microengine in *reset* state.

### 3.1.5.48 dbgMe_ShowBreakpoints()

Show all breakpoints for the specified microengines. The maximum number of breakpoints is defined by the compile-time value DBGME_MAX_BREAKPOINTS.

#### C Syntax

```
int dbgMe_ShowBreakpoints(
    uint meMask,
    dbgMe_BkptArray_T *breakpoints);
```

#### Returns

- DBGME_SUCCESS–the operation succeeded.
- DBGME_BADARG–bad function argument.

### 3.1.5.49 dbgMe_ClearBreakpoint()

Remove the breakpoint associated with breakpointId.

#### C Syntax

```
int dbgMe_ClearBreakpoint(ushort breakpointId);
```

#### Returns

- DBGME_SUCCESS–the operation succeeded.
- DBGME_BADARG–bad function argument.
- DBGME_BADLIB–bad debug library—not initialized.
- DBGME_ENABLED–microengine active.
- DBGME_RESET–microengine in *reset* state.

### 3.1.5.50 dbgMe_ClearAllBreakpoints()

Remove all breakpoints associated with the specified microengines.

#### C Syntax

```
int dbgMe_ClearAllBreakpoints(uint meMask);
```

#### Returns

- DBGME_SUCCESS–the operation succeeded.

- `DBGME_BADARG`–bad function argument.
- `DBGME_BADLIB`–bad debug library—not initialized.
- `DBGME_ENABLED`–microengine active.

#### 3.1.5.51 `dbgMe_Step()`

This function allows the specified microengine context to proceed to the next instruction that allows breakpoint. The microengine starts in an active context and not necessarily the specified context. Therefore, it is possible that the specified context may not “step” if it never becomes active due to never being signaled, or interrupted by another context reaching a breakpoint.

**C Syntax**

```
int dbgMe_Step(uchar me, uchar ctx);
```

**Returns**

- `DBGME_SUCCESS`—the operation succeeded.
- `DBGME_BADARG`—bad function argument.
- `DBGME_ENABLED`—microengine active.
- `DBGME_FAIL`—operation failed.
- `DBGME_NODBG_INFO`—no debug information.
- `DBGME_EOUSTORE`–end of ustore—the program is out of space.
- `DBGME_CTXKILLED`–no context—killed by a `ctx arb[kill]` instruction.
- `DBGME_TIMEOUT`—operation timed out.
- `DBGME_UENGATBREAK`—operation interrupted by a breakpoint.

## 3.2 Remote Services and Control Layer

The Remote Services component consists of a target-side library—`rs_udebug.a`—for the VxWorks, QNX, and Linux target platforms, and a host Win32 DLL—`rs_udbg_client_dll.dll`. The purpose of the target libraries and the host DLL is to export the Debug and Control APIs, presented in Section 3.2.2, “CNTL API—RS_CNTL_doitXX,” to host applications, in particular, the IXP2000 Work Bench.

Most of the core functionality for the following APIs resides on the Target, and is exported by the Target Library `dbgme.a`. That functionality is presented here, as seen by the Host software, through Remote Services, and maps directly to information presented in Section 3.1, “Hardware Debug Library,” The mapping is shown in Table 3-3.

**Table 3-3. Mapping of Target to Host Syntax for Remote Services**

| Target | Host |
|---|---|
| `dbgMe_doitXX(arg1, ..);` | `RS_uDbg_doitXX(hRS_Conn_T, arg1, ..);` |
| `CNTL_doitXX(arg1, ..);` | `RS_CNTL_doitXX(hRS_Conn_T, arg1, ..);` |

The host-side functionality is supplemented by the APIs, presented in the following section, which setup the Host/Target connection and return the `hRS_Conn_T` value, which is passed in the host-side APIs as the first argument.

## 3.2.1 Remote Services Connection API

Remote Services allows a number of connection options from the Windows Host to the remote target. Current options for MEv2 are RPC over TCP/IP and VxWorks WTX. RPC over TCP/IP is supported on the Linux, QNX, and VxWorks platforms. The following APIs are available to provide Connect and Disconnect functionality for the above connection types, and are defined in include\rs_connection.h.

### 3.2.1.1 API Functions

#### 3.2.1.1.1 Open_EtherNet_Connection

Sets up an RPC over TCP/IP or UDP CLIENT Connection, as defined by `TargetName`, where `TargetName` may be either an IP address X.X.X.X or the `HostName` of the Target. Return the Remote Connection Handle value at `*hTarget`.

**C Syntax**

```
RS_API Open_EtherNet_Connection(char *TargetName, hRS_Conn_T *hTarget);
Close_EtherNet_Connection(hRS_Conn_T hTarget);
```

#### 3.2.1.1.2 Open_VxWorks_Connection

Sets up a VxWorks WTX connection, as defined by `TargetName`, which refers to the name of the Tornado target server. Return the remote connection handle value at `*hTarget`.

**C Syntax**

```
RS_API Open_VxWorks_Connection(char *TargetName, hRS_Conn_T *hTarget);
```

#### 3.2.1.1.3 Close_RS_Connection

Closes a connection, opened by `Open_Ethernet_Connection` or `Open_VxWorks_Connection`.

**C Syntax**

```
RS_API Close_RS_Connection(hRS_Conn_T hTarget);
```

## 3.2.2 CNTL API—RS_CNTL_doitXX

The following APIs result in the execution of an `CNTL_doitXX` API being executed on the remote target, with the arguments presented by the host application, and the results returned to the host application. The goal of the control library is to decouple the application from the library and still provide a mechanism to synchronize the applications handling of the network ports with the Remote Services library's handling of the microengines.

### C Syntax

```
int RS_CNTL_InitUof(hRS_Conn_T hTarget, char* buf, int buflen);
int RS_CNTL_Reset(hRS_Conn_T hTarget, uint engineMask, uint portMask,int
clrReg);
int RS_CNTL_Pause(hRS_Conn_T hTarget, uint engineMask, uint portMask,uint
delay);
int RS_CNTL_Resume(hRS_Conn_T hTarget, uint engineMask, uint portMask);
int RS_CNTL_Start(hRS_Conn_T hTarget, uint engineMask, uint portMask);
int RS_CNTL_Step(hRS_Conn_T hTarget, uchar ME, uchar ctx, uint portMask);
```

The target Control Layer APIs, denoted by `CNTL_doitXX` calls are described briefly below:

- `CNTL_Reset`–reset the microengine's and stop network ports
- `CNTL_Start`–start the microengine's and start network ports
- `CNTL_Pause`–pause the microengine's and stop network ports
- `CNTL_Resume`–resume the microengine's and start network ports
- `CNTL_Step`–step the microengine's and start/stop network ports
- `CNTL_InitUof`–parse a UOF image and down load microcode images to the microengine's

***Note:*** The following APIs are not exported by Remote Services and are intended to allow the target network application to be notified of `CNTL_doitxx` actions by Remote Services, and provide a mechanism for the network application to provide the appropriate control over the network ports.

- `CNTL_GetInfo`–get Remote Services connection information
- `CNTL_RegisterCallback`–register a CNTL Callback routine
- `CNTL_UnregisterCallback`–unregister a CNTL Callback routine

## 3.2.3 Debug API—RS_uDbg_doitXX

The following APIs are a result of executing a `dbgMe_doitXX` API on a remote target, with arguments presented by the Host Application, and the results returned to the Host Application. Refer to Section 3.1.5, "API Functions" for details of the Target APIs.

Table 3-4 shows the remote services debug API.

**Table 3-4. Remote Services Debug API**

| Function | Description |
|---|---|
| RS_uDbg_AttachLib() | Initialize the debug library. |
| RS_uDbg_DeleLib() | Delete the debug library object. |

## Table 3-4. Remote Services Debug API (Continued)

| Function | Description |
|---|---|
| RS_uDbg_DefineFreeUstore() | Define the microstore free region for the specified microengines. |
| RS_uDbg_UpdateFreeUstore() | Update the debug-library free-ustore-region values. |
| RS_uDbg_Reset() | Reset microengine(s). |
| RS_uDbg_ClrReset() | Take the microengine(s) out of reset. |
| RS_uDbg_Start() | Start microengines threads from uPC 0. |
| RS_uDbg_Pause() | Stop microengines at earliest context arbitration. |
| `RS_uDbg_Resume()` | Start microengines from current uPC. |
| `RS_uDbg_PauseMeCtx()` | Pause the specified microengines conexts, indicated by the `meMask`, and `ctxMask` parameters |
| `RS_uDbg_ResumeMeCtx()` | Resume the specified microengines conexts, indicated by the `meMask`, and `ctxMask` parameters |
| `RS_uDbg_GetWakeupEvents()` | Reads the wakeup-events of the specified microengine context. |
| `RS_uDbg_PutWakeupEvents()` | Writes the wakeup-events of the specified microengine context. |
| `RS_uDbg_GetSigEvents()` | Reads the signal-events of the specified microengine context. |
| `RS_uDbg_PutSigEvents()` | Writes the signal-events of the specified microengine context. |
| RS_uDbg_IsMeEnabled() | Determine if all the microengines specified in the `meMask` parameter are enabled. |
| RS_uDbg_PutMeCtxPCs() | Set the context-program-counters of the specified micro-engines to the same ustore location indicated by the upc parameter. |
| RS_uDbg_GetMeCtxPC() | Obtain the program-counter of the indicated microengine context and returned the value in pc**.** |
| `RS_uDbg_PutMePCs()` | Sets the program counters of the specified microengine to the values in the array `upc` argument |
| `RS_uDbg_GetMePCs()` | Obtains the program-counters of the specified ME and stores them in the array `upc` argument. |
| `RS_uDbg_PutUwords()` | Write longwords to microengine microstore. |
| RS_uDbg_GetUwords() | Read longwords from microengine microstore. |
| RS_uDbg_SetBreakpoint() | Set a breakpoint. |
| RS_uDbg_ShowBreakpoints() | Lists the current breakpoints. |
| RS_uDbg_ClearBreakpoint() | Clears a breakpoint |
| RS_uDbg_ClearAllBreakpoints() | Clears all breakpoints. |
| `RS_uDbg_PutRelDataReg()` | Writes num `longwordsregData` to the microengine GPR (General Purpose Register) starting at `gprRegNum` |
| `RS_uDbg_GetRelDataReg()` | Obtains num longwords from a microengine GPR starting at `gprRegNum`. |
| `RS_uDbg_PutAbsDataReg()` | Writes num longwords from `regData` to the microengine data register starting at `regNum`. |
| `RS_uDbg_GetAbsDataReg()` | Obtains num longwords from a microengine data register starting at `regNum` |
| RS_uDbg_GetMeStatus() | Obtains information on microengine. |
| RS_uDbg_GetMesStatus() | Obtains information on one or more microengines. |
| RS_uDbg_GetMesEvents() | Obtains event information on or more microengines. |

**Table 3-4. Remote Services Debug API (Continued)**

| Function | Description |
|---|---|
| RS_uDbg_GetMesPCs() | Obtains the program counters of a specified ME and stores them in an array . |
| RS_uDbg_GetMeRunState() | Determines which of the specified MEs are enabled or active. |
| RS_uDbg_PutMeCsr()r | Writes a long-word value to the microengine CSR specified by the me and `csr` parameters respectively. |
| RS_uDbg_GetMeCsr() | Reads a long-word from the microengine CSR specified by the me and `csr` parameters respectively and store it in value. |
| RS_uDbg_PutCsr() | Writes the long-word data to the CSR specified by the `csrAddr` parameter. |
| RS_uDbg_GetCsr() | Reads the long-word from the CSR specified by the `csrAddr` parameter and return in data. |
| RS_uDbg_GetLM() | Obtains count longwords from a microengine Local Memory starting at lmAddr and store them in lmData. |
| RS_uDbg_PutLM() | Writes count longwords from lmData to microengine Local Memory starting at lmAddr. |
| RS_uDbg_GetCAM() | Reads the contents of the microengine CAM and store the tag value, lock bits, and LRU information in camStat. |
| RS_uDbg_PutCAM() | Writes content of camStat to the microengine CAM. |
| RS_uDbg_IsInpStateSet() | Determined if the microengine is in the specified inpState. |
| RS_uDbg_WriteMem32() | Writes a longword to DRAM and SCRATCH or SRAM. |
| RS_uDbg_ReadMem32() | Reads a longword from DRAM and SCRATCH or SRAM. |
| RS_uDbg_PutTbuf() | Writes num longwords from data to `TBUF` starting at the address location specified |
| RS_uDbg_GetRbuf()f | Reads num longwords from RBUF byte location specified |
| RS_uDbg_PutRbufEleCntl() | Writes a quadword from `bufEleCtl` to the `RBUF` element-control specified by `eleNum`. |
| RS_uDbg_GetRbufEleCntl() | Reads a quadword from `bufEleCtl` to the `RBUF` element-control specified by eleNum. |
| RS_uDbg_Step() | Implemented using the “Inline break, Inline continue” method for stepping the microengines. |
| RS_uDbg_GetSysMemSize() | Obtains the system memory sizes. |

## 3.2.3.1 API Functionality

### 3.2.3.1.1 `RS_uDbg_AttachLib()`

Initializes the debug library. Refer to dbgMe_ AttachLib() for a description.

#### C Syntax

```
Int RS_uDbg_AttachLib(hRS_Conn_T hTarget, dbgMe_UstoreFree_T
                      uStoreFree, dbgMeInfo_Image_T *dbgInfo,
                      uint MeMask);
```

#### 3.2.3.1.2 RS_uDbg_DeleLib()

Deletes the debug library object. Refer to dbgMe_ DeleLib() for a description.

**C Syntax**

```
Int RS_uDbg_DeleLib(hRS_Conn_T hTarget);
```

#### 3.2.3.1.3 RS_uDbg_DefineFreeUstore()

Defines the microstore free region for the specified microengines. Refer to dbgMe_DefineFreeUstore() for a description.

**C Syntax**

```
Int RS_uDbg_DefineFreeUstore(hRS_Conn_T hTarget, uint MeMask,
    uint freeUaddr, uint freeSize);
```

#### 3.2.3.1.4 RS_uDbg_UpdateFreeUstore()

Updates the debug-library free-ustore-region values. Refer to dbgMe_UpdateFreeUstore() for a description.

**C Syntax**

```
Int RS_uDbg_UpdateFreeUstore(hRS_Conn_T hTarget, uint MeMask);
```

#### 3.2.3.1.5 RS_uDbg_Reset()

Resets the remote services. Refer to dbgMe_Reset() for a description.

**C Syntax**

```
Int RS_uDbg_Reset(hRS_Conn_T hTarget, uint MeMask, int clrReg);
```

#### 3.2.3.1.6 RS_uDbg_ClrReset()

Takes the microengines out of reset. Refer to dbgMe_ClrReset() for a description.

**C Syntax**

```
Int RS_uDbg_ClrReset(hRS_Conn_T hTarget, uint MeMask);
```

#### 3.2.3.1.7 RS_uDbg_Start()

Starts the microengines threads from uPC 0. Refer to dbgMe_Start() for a description.

**C Syntax**

```
Int RS_uDbg_Start(hRS_Conn_T hTarget, uint MeMask);
```

#### 3.2.3.1.8 `RS_uDbg_Pause()`

Stops the microengines at earliest context arbitration. Refer to dbgMe_Pause() for a description.

**C Syntax**

```
Int RS_uDbg_Pause(hRS_Conn_T hTarget, uint MeMask, uint delay);
```

#### 3.2.3.1.9 `RS_uDbg_Resume()`

Starts the microengines from current uPC. Refer to dbgMe_Resume() for a description.

**C Syntax**

```
Int RS_uDbg_Resume(hRS_Conn_T hTarget, uint MeMask);
```

#### 3.2.3.1.10 `RS_uDbg_PauseMeCtx()`

Stops the microengines contexts, indicated by the `meMask`, and `ctxMask` parameters. Refer to dbgMe_PauseMeCtx() for a description.

**C Syntax**

```
Int RS_uDbg_PauseMeCtx(hRS_Conn_T hTarget, uint meMask, uint ctxMask);
```

#### 3.2.3.1.11 `RS_uDbg_ResumeMeCtx()`

Resumes the specified microengine contexts, indicated by the `meMask`, and `ctxMask` parameters. Refer to dbgMe_ResumeMeCtx() for a description.

**C Syntax**

```
Int RS_uDbg_ResumeMeCtx(hRS_Conn_T hTarget, uint meMask, uint ctxMask);
```

#### 3.2.3.1.12 `RS_uDbg_GetWakeupEvents()`

Reads the wakeup-events of the specified microengine context. Refer to dbgMe_GetWakeupEvents() for a description.

**C Syntax**

```
Int RS_uDbg_GetWakeupEvents(hRS_Conn_T hTarget, uchar me, uchar ctx,
    uint *events);
```

#### 3.2.3.1.13 `RS_uDbg_PutWakeupEvents()`

Writes the wakeup-events of the specified microengine context. Refer to RS_uDbg_PutWakeupEvents() for a description.

**C Syntax**

```
Int RS_uDbg_PutWakeupEvents(hRS_Conn_T hTarget, unsigned char me,
    unsigned int ctxMask, unsigned int events);
```

#### 3.2.3.1.14 `RS_uDbg_GetSigEvents()`

Reads the signal-events of the specified microengine context. Refer to dbgMe_GetSigEvents() for a description.

##### C Syntax

```
Int RS_uDbg_GetSigEvents(hRS_Conn_T hTarget, uchar me, uchar ctx,
    uint *events);
```

#### 3.2.3.1.15 `RS_uDbg_PutSigEvents()`

Reads the signal-events of the specified microengine context. Refer to RS_uDbg_PutSigEvents() for a description.

##### C Syntax

```
Int RS_uDbg_PutSigEvents(hRS_Conn_T hTarget, unsigned char me, unsigned int
ctxMask, unsigned int events);
```

#### 3.2.3.1.16 `RS_uDbg_IsMeEnabled()`

Determine if all the microengines specified in the `meMask` parameter are enabled. Refer to dbgMe_IsMeEnabled() for a description.

##### C Syntax

```
Int RS_uDbg_IsMeEnabled(hRS_Conn_T hTarget, uint MeMask);
```

#### 3.2.3.1.17 `RS_uDbg_PutMeCtxPCs()`

Sets the context-program-counters of the specified micro-engines to the same ustore location indicated by the `upc` parameter. Refer to dbgMe_PutMeCtxPCs() for a description.

##### C Syntax

```
Int RS_uDbg_PutMeCtxPCs(hRS_Conn_T hTarget, uint MeMask,
    uint ctxMask, uint upc);
```

#### 3.2.3.1.18 `RS_uDbg_GetMeCtxPC()`

Obtains the program-counter of the indicated microengine context and returned the value in `pc`. Refer to dbgMe_GetMeCtxPCs() for a description.

##### C Syntax

```
Int RS_uDbg_GetMeCtxPC(hRS_Conn_T hTarget, uchar Me,
       uchar ctx, uint *upc);
```

#### 3.2.3.1.19 `RS_uDbg_PutMePCs()`

Sets the program-counters of the specified microengine to the values in the array `upc` argument. Refer to dbgMe_PutMePCs() for a description.

#### C Syntax

```
Int RS_uDbg_PutMePCs(hRS_Conn_T hTarget,uchar Me, uint *upc);
```

### 3.2.3.1.20 RS_uDbg_GetMePCs()

Obtains the program-counters of the specified microengine and store them in the array `upc` argument.Refer to dbgMe_GetMePCs() for a description.

#### C Syntax

```
Int RS_uDbg_GetMePCs(hRS_Conn_T hTarget, uchar Me, uint *upc);
```

### 3.2.3.1.21 RS_uDbg_PutUwords()

Writes `numUwords` of longwords from the `uWord` value to the ustore location. Refer to dbgMe_PutUwords() for a description.

#### C Syntax

```
Int RS_uDbg_PutUwords (hRS_Conn_T hTarget, uint MeMask,
    uint uWordAddr, unsigned short numWords, uword_T *uWord);
```

### 3.2.3.1.22 RS_uDbg_GetUwords()

Reads `numUwords` longwords from the microengine `ustore` location, specified by `me` and `uAddr` to `uWord`. Refer to dbgMe_GetUwords() for a description.

#### C Syntax

```
Int RS_uDbg_GetUwords (hRS_Conn_T hTarget, uchar me,
    uint uWordAddr, unsigned short numWords, uword_T *uWord);
```

### 3.2.3.1.23 RS_uDbg_SetBreakpoint()

Sets a microengine breakpoint as defined by the `breakpoint` parameter. Refer to dbgMe_SetBreakpoint() for a description.

#### C Syntax

```
Int RS_uDbg_SetBreakpoint (hRS_Conn_T hTarget, dbgMe_Bkpt_T *breakpoint);
```

### 3.2.3.1.24 RS_uDbg_ShowBreakpoints()

Shows all breakpoints for the specified microengine. Refer to dbgMe_ShowBreakpoints() for a description.

#### C Syntax

```
Int RS_uDbg_ShowBreakpoints(hRS_Conn_T hTarget, uint MeMask,
           dbgMe_BkptArray_T *breakpoints);
```

### 3.2.3.1.25 RS_uDbg_ClearBreakpoint()

Removes the breakpoint associated with `breakpointId`. Refer to dbgMe_ClearBreakpoint() for a description.

#### C Syntax

```
Int RS_uDbg_ClearBreakpoint(hRS_Conn_T hTarget, short breakpointId);
```

### 3.2.3.1.26 RS_uDbg_ClearAllBreakpoints()

Removes all breakpoints associated with the specified microengine. Refer to dbgMe_ClearAllBreakpoints() for a description.

#### C Syntax

```
Int RS_uDbg_ClearAllBreakpoints(hRS_Conn_T hTarget, uint MeMask);
```

### 3.2.3.1.27 RS_uDbg_PutRelDataReg()

Write num longwords from regData to the ME data register starting at regNum. Refer to dbgMe_PutRelDataReg() for a description.

#### C Syntax

```
Int RS_uDbg_PutRelDataReg(hRS_Conn_T hTarget, uchar Me, uchar ctx,
    ixp_RegType_T regType, unsigned short regNum, uint *regData, uint count);
```

### 3.2.3.1.28 RS_uDbg_GetRelDataReg()

Obtain num longwords from a ME data register starting at regNum. Refer to dbgMe_GetRelDataReg() for a description.

#### C Syntax

```
Int RS_uDbg_GetRelData(hRS_Conn_T hTarget, uchar Me, uchar ctx,
    ixp_RegType_T regType, unsigned short regNum,
    uint *regData, uint count);
```

### 3.2.3.1.29 RS_uDbg_PutAbsDataReg()

Writes `num` longwords from regData to the microengine data register starting at `regNum`. Refer to dbgMe_PutAbsDataReg() for a description.

#### C Syntax

```
Int RS_uDbg_PutAbsDataReg(hRS_Conn_T hTarget, uchar Me,
    ixp_RegType_T regType, unsigned short regNum,
    uint *regData, uint count);
```

### 3.2.3.1.30 RS_uDbg_GetAbsDataReg()

Obtains `num` longwords from a microengine data register starting at `regNum`. Refer to dbgMe_GetAbsDataReg() for a description.

#### C Syntax

```
Int RS_uDbg_GetAbsDataReg(hRS_Conn_T hTarget, uchar Me,
    ixp_RegType_T regType, unsigned short regNum,
    uint *regData, uint count);
```

##### 3.2.3.1.31 RS_uDbg_GetMeStatus()

Obtains information about the microengine. Refer to dbgMe_GetMeStatus() for a description.

#### C Syntax

```
Int RS_uDbg_GetMeStatus(hRS_Conn_T hTarget, uchar Me,
    dbgMe_MeStatus_T *MeStatus);
```

##### 3.2.3.1.32 RS_uDbg_GetMesStatus()

Obtains information on one or more microengines.

#### C Syntax

```
Int RS_uDbg_GetMesStatus(hRS_Conn_T hTarget, uint MeMask,
    dbgMe_MeStatus_T *MeStatus, uint Buffersize);
```

##### 3.2.3.1.33 RS_uDbg_GetMesEvents()

Obtains event information for one or more microengines.

#### C Syntax

```
Int RS_uDbg_GetMesEvents(hRS_Conn_T hTarget, uint MeMask,
    dbgMe_MeEvents_T *MeEvents, uint Buffersize);
```

##### 3.2.3.1.34 RS_uDbg_GetMesPCs()

Obtains the program-counters of the specified ME and store them in the array upc argument.

#### C Syntax

```
int dbgMe_GetMesPCs(uchar me, uint *upc);
```

##### 3.2.3.1.35 RS_uDbg_GetMeRunState()

Determines which of the MEs that are specified in meMask are enabled or active. The MEs that are either enabled or active are returned in the enabledMask and the activeMask parameter respectively.

#### C Syntax

```
int dbgMe_GetMeRunState(uint meMask, uint *enabledMask, uint *activeMeMask,);
```

##### 3.2.3.1.36 RS_uDbg_PutMeCsr()

Writes a long-word value to the microengine CSR specified by the `me` and `csr` parameters. Refer to dbgMe_PutMeCsr() for a description.

### C Syntax

```
Int RS_uDbg_PutMeCsr(hRS_Conn_T hTarget, uchar Me,
    Hal_Me_CSR_T csr, uint value);
```

## 3.2.3.1.37 RS_uDbg_GetMeCsr()

Reads a long-word from the microengine CSR specified by the `me` and `csr` parameters respectively and stores it in value. Refer to dbgMe_GetMeCsr() for a description.

### C Syntax

```
Int RS_uDbg_GetMeCsr (hRS_Conn_T hTarget, uchar Me, Hal_Me_CSR_T csr,
    uint *value);
```

## 3.2.3.1.38 RS_uDbg_PutCsr()

Writes the long-word data to the CSR specified by the `address` parameters. Refer to dbgMe_PutCsr() for a description.

### C Syntax

```
Int RS_uDbg_PutCsr(hRS_Conn_T hTarget, uint XAddrH, unit XAddrL, uint value);
```

## 3.2.3.1.39 RS_uDbg_GetCsr()

Reads the long-word from the CSR specified by the address parameters and returns in data. Refer to dbgMe_GetCsr() for a description.

### C Syntax

```
Int RS_uDbg_GetCsr(hRS_Conn_T hTarget, uint XAddrH, unit XAddrL, uint *value);
```

## 3.2.3.1.40 RS_uDbg_GetLM()

Obtains count longwords from a microengine Local Memory starting at `lmAddr` and stores them in `lmData`. Refer to dbgMe_GetLM() for a description.

### C Syntax

```
Int RS_uDbg_GetLM (hRS_Conn_T hTarget, uchar Me,
    uint lmAddr, uint *lmData, uint lmCount);
```

## 3.2.3.1.41 RS_uDbg_PutLM()

Writes count longwords from `lmData` to microengine Local Memory starting at `lmAddr`. Refer to dbgMe_PutLM() for a description.

### C Syntax

```
Int RS_uDbg_PutLM (hRS_Conn_T hTarget, uchar Me,
    uint lmAddr, uint *lmData, uint lmCount);
```

#### 3.2.3.1.42 `RS_uDbg_GetCAM()`

Reads the contents of the microengine CAM and store the tag value, lock bits, and LRU information in `camStat`. Refer to dbgMe_GetCAM() for a description.

##### C Syntax

```
Int RS_uDbg_GetCAM(hRS_Conn_T hTarget, uchar Me,
    dbgMe_MeCamStatus_T *camStat);
```

#### 3.2.3.1.43 `RS_uDbg_PutCAM()`

Writes content of `camStat` to the microengine CAM. In order to maintain the LRU information being written, the `camStat->lru[x]` with the lowest value is written first, then the next lowest, and so on. Refer to dbgMe_PutCAM() for a description.

##### C Syntax

```
Int RS_uDbg_PutCAM(hRS_Conn_T hTarget, uchar Me,
    dbgMe_MeCamStatus_T *camStat);
```

#### 3.2.3.1.44 `RS_uDbg_IsInpStateSet()`

This routine determines if the microengine is in the specified `inpState`, and return a value indicating whether it’s set, not set, or an error code. Refer to dbgMe_IsInpStateSet() for a description.

##### C Syntax

```
Int RS_uDbg_IsInpStateSet (hRS_Conn_T hTarget, uchar Me, uchar inpState);
```

#### 3.2.3.1.45 `RS_uDbg_WriteMem32()`

Writes a longword to DRAM and SCRATCH or SRAM. Refer to dbgMe_WriteMem32() for a description.

##### C Syntax

```
Int RS_uDbg_WriteMem32(hRS_Conn_T hTarget, dbgMe_MemRegion_T memRegion,
    uint addr, uint *data, uint numWords);
```

#### 3.2.3.1.46 `RS_uDbg_ReadMem32()`

Reads a longword from DRAM and SCRATCH or SRAM. Refer to dbgMe_ReadMem32() for a description.

##### C Syntax

```
Int RS_uDbg_ReadMem32(hRS_Conn_T hTarget, dbgMe_MemRegion_T memRegion,
    uint addr, uint *data, uint numWords);
```

### 3.2.3.1.47 RS_uDbg_PutTbuf()

Writes num longwords from data to `TBUF` starting at the address location specified. Refer to dbgMe_PutTbuf() for a description.

#### C Syntax

```
Int RS_uDbg_PutTbuf(hRS_Conn_T hTarget, uchar Me, uint xbufByte, uint *data,
    uint count);
```

### 3.2.3.1.48 RS_uDbg_GetRbuf()

Reads num longwords from RBUF byte location specified. Refer to dbgMe_GetRbuf() for a description.

#### C Syntax

```
Int RS_uDbg_GetRbuf(hRS_Conn_T hTarget, uchar Me, uint xbufByte, uint *data,
    uint count);
```

### 3.2.3.1.49 RS_uDbg_PutRbufEleCntl()

#### C Syntax

```
Int RS_uDbg_PutRbufEleCntl(hRS_Conn_T hTarget, uint eleNum, uint64 bufEleCntl,
    uchar valid);
```

### 3.2.3.1.50 RS_uDbg_GetRbufEleCntl()

#### C Syntax

```
Int RS_uDbg_GetRbufEleCntl(hRS_Conn_T hTarget, uint eleNum,
    uint64 *bufEleCntl);
```

### 3.2.3.1.51 RS_uDbg_Step()

This function is implemented using the “Inline break, Inline continue” method described above for stepping the microengines. Refer to dbgMe_Step() for a description.

#### C Syntax

```
Int RS_uDbg_Step(hRS_Conn_T hTarget, uchar Me, uchar ctx);
```

### 3.2.3.1.52 RS_uDbg_GetSysMemSize()

Obtains the system memory sizes. Refer to dbgMe_GetSysMemSize() for a description.

#### C Syntax

```
Int RS_uDbg_GetSysMemSize(hRS_Conn_T hTarget, dbgMe_MemConfig_T * memConfig);
```

## 3.3 Control Layer Design

### 3.3.1 CNTL Callback Registration API

This section describes the CNTL Callback scheme and the APIs to register and de-register the callback handling.

#### 3.3.1.1 Defined Types, Enumerations, and Data Structures

##### 3.3.1.1.1 CNTL Callback Structure

The following packet is a required argument to the CNTL_RegisterCallback API:

**C Syntax**

```
typedef struct CNTL_Register_S{
    void    *cb_link;         /* internal use only */
    int     (*callback)(uint Event_Code, uint Param1, uint Param2,
                              uint UserData);
    uint    NotifyMask;       /* see Event Notify Masks */
    uint    UserData;         /* passed to callback as shown above */
    uint    Priority;         /* used to determine callback order */
                              /*  0 highest priority */
} CNTL_Register_T;
```

**C Syntax**

```
typedef uint CHandle;
```

##### 3.3.1.1.2 Callback Event Definitions

The following event codes are defined to indicate the processing required by a network application:

- `#define CNTL_RESET 1`
  printing as #define `CNTL_RESET1 (multiple)`
  Generated in `CNTL_Reset`
  `Param1 = MeMask` and `Param2 = Port Mask`
- `#define CNTL_START`
  Generated in `CNTL_Start`, `CNTL_Resume`, `CNTL_Step`,
  `Param1 = MeMask` and `Param2 = Port Mask`
- `#define CNTL_STOP 3`
  Generated in `CNTL_Pause`, `CNTL_Step`
  `Param1 = MeMask` and `Param2 = Port Mask`
- `#define CNTL_UOF_LOADED 4`
  Generated after a UOF image has been received and parsed by the target `CNTL_InitUof` handler. The images have not yet been downloaded to the microengines. This allows the callback perform `BindSymbols` or perform other modifications to the UOF image.
  `Param1 = ucloHandle`, that is to be passed to any `Uclo_doitXX` APIs, that the callback wishes to use. `Param2` is unused.

The following Event Notify Masks are defined to indicate the callback processing required on the part of the Network Application:

- `#define CNTL_RESET_NOTIFY1`
- `#define CNTL_START _NOTIFY2`
- `#define CNTL_STOP _NOTIFY4`
- `#define CNTL_UOF_LOADED_NOTIFY8`
- `#define CNTL_NOTIFY_ALL0xf`

## 3.3.1.2 API Functions

The following API is defined to allow an Intel XScale® core network application to register and deregister a callback routine. These APIs are only valid on the target Intel XScale® core processor. See Section 3.3.1.1.1, "CNTL Callback Structure," for definition of `CNTL_Register_T *callback_info`.

### 3.3.1.2.1 `CNTL_RegisterCallback()`

Register a callback routine, as specified by `callback_info.callback`, for specified events, as specified in `callback_info.NotifyMask`. All callbacks registered for a particular event is called by `cntl_ExecuteCallback`, in the order as determined by the priority values provided in the `CNTL_Register_T` packets, where zero represents the highest priority.

#### C Syntax

```
int
CNTL_RegisterCallback(CNTL_Register_T *callback_info, CHandle *handle);
```

### 3.3.1.2.2 `CNTL_UnregisterCallback()`

Unregister a callback routine. The `CHandle` value is that passed back from `CNTL_RegisterCallback *handle` pointer.

See Section 3.5, "Control Layer Callback Usage" for an example of callback usage.

#### C Syntax

```
CNTL_UnregisterCallback(CHandle handle);
```

## 3.3.2 CNTL APIs

This section describes the routines exported by the Control Layer, and which provide system and board level initialization and control.

The most notable feature of the new API handlers is that the Dev_1200 NetApp functionality in the form of `SysConfig::<StartPorts, StopPorts, ResetPorts>` calls, is now replaced with `cntl_ExecuteCallback` calls. The placement of these calls is roughly that of the Dev_1200 `SysConfig::<StartPorts, StopPorts, ResetPorts>` calls.

All of the Target routines are single threaded. These APIs are accessible from both network applications and Remote Services. The following API descriptions describe the target actions.

### 3.3.2.1 API Functions

#### 3.3.2.1.1 `CNTL_InitUof()`

Pass a buffer and buffer length, containing an image of a UOF file, and make appropriate calls to parse the UOF file and make any necessary modifications to the embedded microcode images. The latter is accomplished by executing any registered callback routine for the `UOF_LOADED` event. After the above operations are complete, the embedded microcode images are downloaded to the proper MEv2 microengines and thread contexts. After the UOF file was loaded, a handle to an object, representing the in memory image of the UOF file, is retrieved and saved for future actions on the UOF image, including those executed by the callback routine.

**C Syntax**

```
int CNTL_InitUof(char * buffer, int buflen);
```

Basic code for handler is provided in the following section.

**C Example Code**

```
if (uclo_handle)
        UcLo_DeleObj(uclo_handle);

    if ((uof_buffer) && (uof_buflen < buflen)) {
        free(uof_buffer);
       uof_buffer = NULL;
    }

  if (uof_buffer == NULL) {
       uof_buffer = malloc(buflen);
       uof_buflen = buflen;
    }
    memcpy(uof_buffer, buf, buflen);

   if (retval = UcLo_MapObjAddr(&uclo_handle, buf, buflen, 0)){
        uclo_handle = NULL;
       return(retval);// return uclo error

cntl_ExecuteCallback(CNTL_UOF_LOADED, uclo_handle, 0, UserData);
// call any/all registered callback for CNTL_UOF_LOADED

return(UcLo_WriteUimageAll(uclo_handle));
}
```

#### 3.3.2.1.2 `CNTL_GetInfo()`

**C Syntax**

```
int CNTL_GetInfo(CNTL_Info_T *cntlInfo);
```

This functionality allows a network application to query the status of the target/host Remote Services connection. The definition for `CNTL_Info_T` follows.

#### 3.3.2.1.3 CNTL_Info_S

The data structure specifies control information.

##### C Syntax

```
typedef struct CNTL_Info_S {
    uint    StatusMask;
} CNTL_Info_T;
```

#### 3.3.2.1.4 Mask Status Flags

These symbolic constants specify valid mask status flag values.

##### C Syntax

```
#define RS_RPC_CONNECTED     1
#define RS_WTX_CONNECTED     2
#define RS_UOF_LOADED        4
```

##### Defined Values

| | |
|---|---|
| RS_RPC_CONNECTED | RPC connected. |
| RS_WTX_CONNECTED | WTX connected. |
| RS_UOF_LOADED | Microcode object file is loaded. |

#### 3.3.2.1.5 CNTL_Reset

##### C Syntax

```
int CNTL_Reset(uint MeMask, uint portMask, int clrReg);
```

`CNTL_Reset` performs the following operations:

##### C Example Code

```
              // Note - MEv1 DEV_1200 reset uEngines here, in
              // addition to after the Port Resets

cntl_ExecuteCallback(CNTL_RESET, MeMask, portMask, UserData);
              // call any/all registered callback for CNTL_RESET

dbgMe_Reset(MeMask, clrReg);
              // Reset the microengine's
              // if (clrReg) clear the microengine Reset states
```

#### 3.3.2.1.6 `CNTL_Start`

##### C Syntax

```
int CNTL_Start(uint engineMask, uint portMask);
```

`CNTL_Start` performs the following operations:

##### C Example Code

```
dbgMe_Start(MeMask);     // Start the microengine's

cntl_ExecuteCallback(CNTL_START, MeMask, portMask, UserData);
// call any/all registered callback for CNTL_START
```

#### 3.3.2.1.7 `CNTL_Pause`

##### C Syntax

```
int CNTL_Pause(uint engineMask, uint portMask, uint delayMs);
```

`CNTL_Pause` performs the following operations:

##### C Example Code

```
cntl_ExecuteCallback(CNTL_STOP, MeMask, portMask, UserData);
// call any/all registered callback for CNTL_STOP

dbgMe_Pause(MeMask, delayMs);// Pause the microengine's
```

#### 3.3.2.1.8 `CNTL_Resume`

##### C Syntax

```
int CNTL_Resume(uint engineMask, uint portMask);
```

`CNTL_Resume` performs the following operations:

##### C Example Code

```
cntl_ExecuteCallback(CNTL_START, MeMask, portMask, UserData);
// call any/all registered callback for CNTL_START

dbgMe_Resume(MeMask);       // Start the microengine's
```

#### 3.3.2.1.9 `CNTL_Step`

##### C Syntax

```
int CNTL_Step(uchar Me, uchar ctx, uint portMask);
```

`CNTL_Step` performs the following operations:

### C Example Code

```
MeMask = 1 << Me;

cntl_ExecuteCallback(CNTL_START, MeMask, portMask, UserData);
// call any/all registered callback for CNTL_START

dbgMe_Step(Me, ctx);// Step the microengine

cntl_ExecuteCallback(CNTL_STOP, MeMask, portMask, UserData);
// call any/all registered callback for CNTL_STOP
```

#### 3.3.2.1.10 `cntl_ExecuteCallback`

### C Syntax

```
int cntl_ExecuteCallback(unsigned int Event, unsigned int Param1, unsigned int
Param2, unsigned int UserData);
```

`cntl_ExecuteCallback` is an internal routine which does the following operations:

### C Example Code

```
For each CNTL_Register_T structure (CReg) in the callbacks linked list:
    If (EventsMasks[Event] & CReg-> NotifyMask;
        Call CReg->callback(Event, Param1, Param2, CReg->UserData);
    CReg = CReg->cb_link;
```

`Param1` and `Param2` are as defined in Section 3.3.2.1.2, “CNTL_GetInfo() The calling order is controlled by the list order, which was determined by the `CReg->Priority` value where zero is highest priority.

## 3.4 Remote Services Design

The Remote Services leaf (`rs_udebug`) supplies a Host side Win32 DLL (`Rs_udbg_client_dll.dll`) and either a VxWorks, QNX, or Linux library (`rs_udebug.a`) for the Target side of choice. Two transports are supported for the MEv2 products, RPC over TCP/IP, and VxWorks WTX.

As the basic transport frameworks remain similar to the MEv1 transport framework, the MEv2 support provides handlers for the new or modified APIs, provide support for multiple Targets (see Section 3.4.2, “Host RPC CLIENT Handling Support and Section 3.4.5, “Breakpoint Handling) and provide ENDIAN support for the WTX interface (see Section 3.4.4, “Host API WTX ENDIAN Support).

```
RS_uDbg_<Get/Put>LM, RS_uDbg_<Get/Put>RelNN
RS_uDbg_<Get/Put>CAM, RS_uDbg_<Get/Put>MeCsr
RS_uDbg_IsInpState
RS_uDbg_PutTbuf, RS_uDbg_GetRbuf
RS_uDbg_<Get/Put>WakeupEvents, RS_uDbg_<Get/Put>SigEvents
RS_uDbg_<Get/Put>RbufEleCntl
RS_uDbg_Step, RS_uDbg_ResumeMeCtx, RS_uDbg_PauseMeCtx
RS_GetSysMemSize, RS_AttachLib, RS_uDbg_<Get/Put>Uwords
```

```
RS_uDbg_<Put/Get>Rel<Rd/Wr>Xfer, RS_uDbg_<Put/Get>RelGPR,
RS_uDbg_<Put/Get>RelReg, RS_uDbg_<Put/Get>AbsReg
RS_uDbg_<Write/Read>Mem32
```

## 3.4.1 `Rs_udbg_client_dll.dll`

`Rs_udbg_client_dll.dll` is presented by a C interface (`Include\rs_udbg_lib.h`), but is designed around a number of C++ classes. Each of the `Open_TypeXX_Connection` APIs, presented in section Section 3.2.1, "Remote Services Connection API, creates a C++ class `(RS_Target(rs_target.h))`.

### 3.4.1.1 Remote Services Connections

The Connection type determines the proper constructor, which opens the underlying transport as either an RPC CLIENT or VxWorks WTX connection, and creates either an `RS_uDbg_RPC` or `RS_uDbg_WTX` C++ class to service the connection. The `RS_uDbg_RPC` and `RS_uDbg_WTX` classes rely on a C++ virtual class `RS_Debug(rs_udbg.h)` to define the handler prototypes.

When `RS_uDbg_AttachLib` is called, the handler uses the Connection Type, which was determined above, to create one of two C++ classes defined below:

- `RS_uDbg_RPC` initializes as an RPC SERVER for the BreakPoint Callback handler. The create then calls the Target `uDbg_AttachLib` via the RPC CLIENT connection.
- `RS_uDbg_WTX` registers for BreakPoint Callback WTX event, and calls the Target `uDbg_AttachLib` via the WTX connection.

Each of the above classes provide appropriate data formatting and driver support for the defined Connection Type. These classes use a common Base Class (`RS_Debug`) and define a common set of handlers, which correspond to the C interface APIs (`RS_uDbg_doitXX`), that are presented to the user. The classes are presented as `targetObj->debug`. The `RS_uDbg_RPC` class constructor is passed to the `RPC CLIENT` handle, gotten via `create_hClient`, while the `RS_uDbg_WTX` class constructor is passed the WTX connection handle, gotten via `wtxInitialize`.

The proper sequence of user APIs to use the interface is shown below:

1. `Status = Open_TypeXX_Connection("Name", *hTarget);`
   `// defines the Connection`
2. `Status = RS_uDbg_AttachLib(hTarget, pUStoreFree, pdbgInfo, uEngineMask);`
   `// pUStoreFree may be NULL`
   `// completes the connection and initializes the remaining handlers`
3. `Status = RS_uDbg_doitXX(hTarget, arg1, arg2,..)// All APIs available`

## 3.4.2 Host RPC CLIENT Handling Support

For each new or modified handler, the following design items are supplied:

- Define an RS_uDbg_doitXX prototype in rs_udbg_lib.h and a doitXX handler in the RS_Debug C++ class (rs_udbg.h).
- Implement the RS_uDbg_doitXX handler in rs_udbg_clib.cpp. The handler basically does a `return targetObj->debug->doitXX(arg1, arg2,..);`

- Define an `rs_uDbg_doitxx_T` packet and define `RS_UDBG_DOITXX`, as the request index. The `RS_UDBG_DOITXX` definition is made in the program `SA1200TARGET` structure in `rs_udbug.x`. The rs_udbug.x definitions are made using the RPC language. The `rs_uDbg_doitxx_T` packet presents the arguments to `RS_uDbg_doitXX` in a format that is acceptable to the `RPC XDR` handling.
- Generate `rs_udebug.h` from `rs_udebug.x` using `rpcgen -h rs_udebug.x > rs_udebug.h`. `rs_udebug.h` may be built manually, skipping the `rs_udebug.x step`. However, it seems simpler to build `rs_udebug.x` and convert to `rs_udebug.h`.
- Define and implement a `doitXX` handler in the C++ class `RS_uDbg_RPC (rs_udbg_rpc.h/cpp)`. This routine sets up the `rs_uDbg_doitxx_T arg`, allocates storage for the return data `rs_uDbg_retXX_T res`, and calls `rs_udbg_doitxx_c(&arg, &res, client) (rs_udebug_clnt.c)`. The client argument is the RPC CLIENT connection passed to the `RS_uDbg_RPC` class constructor.
- Define and implement the handler `rs_uDbg_doitXX_c(argp, resp, client) (rs_udebug_clnt.c)`. This handler basically does the following:

#### C Syntax

```
clnt_call(client, RS_UDBG_DOITXX, xdr_rs_uDbg_doitxx_T , argp,
xdr_rs_uDbg_status_T, resp, TIMEOUT);
```

The xdr_xxx_T arguments are the addresses of XDR routines (see below†).

- Define and implement the `xdr_rs_uDbg_doitxx_T handler (rs_udebug_xdr.c)`, which converts data from the `rs_uDbg_doitxx_T` packets to/from the XDR stream. If existing handlers can be used, this step may be omitted.

### 3.4.2.1 General Changes

The interface to the RPC Client layer `rs_udbg_doitXX_c` APIs are thread safe by allowing the `RS_uDbg_RPC` layer allocate the XDR storage for the return data, and pass a pointer, to the storage, to the `rs_udbg_doitXX_c` APIs. The new interface for the `rs_udbg_doitXX_c` APIs is shown below.

#### C Syntax

```
rs_uDbg_ReturnXX_T *
rs_udbg_doitXX_c (rs_uDbg_doitXX_T *argp,
                  rs_uDbg_ReturnXX_T *res,
                  CLIENT *clnt);
```

## 3.4.3 Host API WTX Handling Support

For each new or modified handler, the following design items are supplied:

- Modify rs_target.cpp to make a `RemoteEntryInsert(RS_UDBG_DOITXX`, “`uDbg_doitXX`”) call. This call retrieves and saves in an indexed array, the Target address for the Target API, which handles the RS_uDbg_doitXX request. The RS_UDBG_DOITXX value is defined in rs_udebug.h, and is the same value the RPC handlers use. This step assumes steps 1 and 3 in 3.4.1.2 have been completed.
- Define and implement a handler in the C++ class RS_uDbg_WTX (rs_udbg_wtx.h/cpp) to do the remote call, using basically the following VxWorks WTX APIs:

— `wtxMemAlloc`—allocate a Target Buffer

— `wtxMemFree`—free a Target Buffer

— `wtxMemWrite`—write to a Target Buffer

— `wtxMemRead`—read from a Target Buffer

— `wtxDirectCall`—execute a Target routine. The target address was determined in step 1, using the `wtxSymFind` API.

The above API addresses were obtained from the wtxapi.dll, using the Win32 API `GetProcAddress`, and stored in pointers named `Fn_wtxMemAlloc` etc.

## 3.4.4 Host API WTX ENDIAN Support

The MEv2 WTX supports both VxWorks LE and VxWorks BE Target platforms. The `wtxDirectCall` API is used to execute the remote subroutine, and is called as shown below.

### C Syntax

```
STATUS wtxDirectCall(HWTX handle, TGT_ADDR_T entry, void *retValue,
                     TGT_ARG_T arg0, TGT_ARG_T arg1, ...TGT_ARG_T arg9 );
```

The `*retValue` and `arg0-arg9` arguments are treated as 32bit integers and are correctly passed by WTX for both LE and BE Targets. If a Target Buffer is used to send or receive data, the address of the buffer is passed correctly, if presented as one of the `arg0 - arg9` values. However, the contents of the buffer are incorrect for the BE Targets. We can determine the Target ENDIAN using the following call and making the appropriate conversions:

### C Syntax

```
TgtEndian = wtxTargetEndianGet();// Get the Target Endian value
```

As we have no control over the Target side WTX processing, the following translation handlers are implemented in the Host Side `RS_uDbg_WTX` C++ class:

### C Syntax

```
uDbg_uStoreFree_T *Endian_uDbg_uStoreFree (
    uDbg_uStoreFree_T *locBuf,
    uDbg_uStoreFree_T *pUStore );
```

### C Syntax

```
uDbg_MeCamStatus_T *Endian_uDbg_MeCamStatus(
    uDbg_MeCamStatus_T *locBuf,
    uDbg_MeCamStatus_T *pMeCamStatus );
```

### C Syntax

```
uDbg_Bkpt_T *Endian_uDbg_Bkpt( uDbg_Bkpt_T *locBuf, uDbg_Bkpt_T *pBkpt );
```

### C Syntax

```
uDbg_Bkpt_Array_T *Endian_uDbg_Bkpt_Array(
    uDbg_Bkpt_Array_T *locBuf,
    uDbg_Bkpt_Array_T *pBkpt_Array );
```

### C Syntax

```
uDbg_UEngStatus_T *Endian_uDbg_uEngStatus(
    uDbg_UEngStatus_T *locBuf,
    uDbg_UEngStatus_T *pUEngStatus );
```

The above return either 2nd `buffer(if no ENDIAN conversion or the first buffer NULL (Read Mode))`, or the first buffer with the required ENDIAN swapping done.

The following four routines allow the WTX send data to be ENDIAN converted and a buffer to be allocated and freed if the count exceeds the `EBSize` value (64 units or uints64). The conversion occurs prior to the `wtxMemWrite` calls. The pointer passed back is the pointer to the data that is to be written to the `Wtx` Target buffer, via the `wtxMemWrite` call. The routines also allow WTX read data to be `ENDIAN` converted in the callers read buffer, prior to control being passed back to the caller. These calls are made with a `NULL locBuf`. The `freeXX` routines deallocate `pSendData`, if conversion was indicated and `pSendData` does not equal `locBuf`.

### C Syntax

```
uint *Endian_uDbg_ULONGS (uint *locBuf, uint *pCallerData, uint Count);
```

### C Syntax

```
void  Endian_uDbg_freeULONGS (uint *locBuf, uint *pSendData);
```

### C Syntax

```
uint64 *Endian_uDbg_ULONGS64 (uint64 *locBuf, uint64 *pCallerData,
    uint Count);
```

#### C Syntax

```
void  Endian_uDbg_freeULONGS64 (uint64 *locBuf, uint64 *pSendData);
```

## 3.4.5 Breakpoint Handling

The breakpoints are handled by either an RPC SERVER implementation or by a callback Event handling mechanism in the case of the WTX transport. Both mechanisms have been enhanced to allow for multiple IXP Target processors.

### 3.4.5.1 RPC SERVER Breakpoint Callbacks

The RPC SERVER is implemented on a single dedicated thread, which registers as a SERVER for the XSC2000VERSION service and supplies a callback handler for all service requests. The service requests come in, in the form of an `rs_uDbg_Bkpt_CK_T` breakpoint callback message.

The MEv2 Breakpoint callback handling has been enhanced over the MEv1 product by providing support for multiple IXP Targets.

The parameter `uint Hhandle;` has been added to the `rs_uDbg_Breakpoint_T (SetBreakpoint)` and `rs_uDbg_Bkpt_CK_T (Breakpoint callback)` messages.

The above value is set to the `RS_uDbg_RPC` C++ class object, that is controlling the RPC session to the IXP Target processor. The value is saved by the Target RPC SetBreakpoint handler, and passed back by the Breakpoint callback handler. The RPC SERVER service handler casts the value back to the controlling RPC C++ class, which then makes the appropriate calls to handle the Breakpoint callback.

### 3.4.5.2 WTX Breakpoint Callbacks

The WTX Breakpoint callback mechanism is done via an Event Notification and associated message. The Host WTX software registers for the IXP2000 Event Notification and supplies a handler address.

The WTX SetBreakpoint handler makes a WTX remote call to a WTX only handler (`WTX_dbgMe_SetBreakpoint`). As with the RPC interface, the WTX SetBreakpoint handler now passes a `Hhandle` value, set to the `RS_uDbg_WTX` C++ class object, that is controlling the WTX session to the IXP Target processor. The value is saved by the Target WTX SetBreakpoint handler, and passed back by the Breakpoint callback handler in the IXP2000 Event message. The Host WTX IXP2000 Event handler casts the value back to the controlling WTX C++ class, which then makes the appropriate calls to handle the Breakpoint callback.

## 3.4.6 `Rs_udebug` Library

The `rs_udebug` library provides the Target Side handling of Remote Services for either a VxWorks, QNX, or Embedded Linux Target and provides for either RPC over TCP/IP or VxWorks WTX transport support. A very limited control interface is provided on the Target by a single API. The interface is shown below.

#### C Syntax

```
int WbSvr_init();    // Initialize the rs_udebug interface
```

RPC over Ethernet is the default interface, and is the only interface available for Embedded Linux Targets.

### 3.4.6.1 Target RPC Handling

An RPC CLIENT interface is provided for the RPC based Breakpoint callback, which is handled on the Host, and a RPC SERVER interface is provided for all other APIs. There is no change to the Breakpoint functionality and as such is not discussed further.

The `rs_uDbg_doitxx_T` packet, the `RS_UDBG_DOITXX` identifier, and the `xdr_rs_uDbg_doitxx_T` handlers are the same as used on the Host and described in Section 3.4.2, "Host RPC CLIENT Handling Support" on page 3-49.

For each new or modified handler, listed in section Section 3.4, "Remote Services Design" on page 3-48, the following design items are supplied:

- Add a line to declare rs_uDbg_doitxx_T in the union argument in rs_udebug_svc.c, to allow the packet to be passed to local handler below.
- Add a case `RS_UDBG_DOITXX: handler` (`rs_udebug_svc.c`) to setup the following variables:

| | |
|---|---|
| `_xdr_argument` | `xdr_rs_uDbg_doitxx_T handler` to handle request |
| `_xdr_result` | `xdr_xxx_T handler` to handle response |
| `local` | `rs_udbg_dointxx_1_svc handler` (see the following †) |

- Define and implement an `rs_udbg_dointxx_1_svc handler (rs_udebug_server.c)`, to convert the request data from the `rs_uDbg_doitxx_T` packet, to the required Target format and, makes the proper `CNTL_doitXX` or `uDbg_doitXX` call on the Target. The routine prototype is shown below.

#### C Syntax

```
result *
rs_udbg_doitxx_1_svc(rs_uDbg_doitxx_T *argp, svc_req *rqstp);
```

### 3.4.6.2 Target WTX Handling

All of the VxWorks WTX Host/Target setup is done on the Host, with the exception of the BreakPoint handling. `WTX_uDbg_SetBreakpoint` and `WTX_BKCallback` routines are provided by `rs_udebug.a` (`rs_udebug_server.c`) to provide the Breakpoint handling. All other Target WTX communication is handled by VxWorks.

As per the Host changes mentioned in Section 3.4.5, "Breakpoint Handling" on page 3-53, `WTX_uDbg_SetBreakpoint` now passes an additional parameter `Hhandle` to the target. The parameter is saved and passed back as an additional message parameter by the `WTX_BKCallback` routine in the IXP2000 Event message. The Host CallBack Event handler uses the `Hhandle` value, to pass the Breakpoint Callback message, to the controlling RS_uDbg_WTX C++ class, for proper handling.

## 3.5 Control Layer Callback Usage

The following are examples of two callback routines, one to handle PORT's and the other to handle UOF Image download events.

### C Example Code

```
int
cntl_CallBackPorts(unsigned int Event, unsigned int param1,
            unsigned int param2, unsigned int UserData)
{
char*pMsg = Emsgs[Event-1];
CNTL_Info_Tstatus;
unsigned inti, mask = 1;

    printf("\nCallBackPorts %s param1 = %x, param2 = %x, UserData = %x\n",
            pMsg, param1, param2, UserData);

    CNTL_GetInfo(&status);

/* Do any PORT specific processing here...*/

    return DBGME_SUCCESS;
}
```

### C Example Code

```
int
cntl_CallBackUof(unsigned int Event, unsigned int param1,
            unsigned int param2, unsigned int UserData)
{
char*pMsg = Emsgs[Event-1];
CNTL_Info_Tstatus;
unsigned inti, mask = 1;

    printf("\nCallBackUof %s param1 = %x, param2 = %x, UserData = %x\n",
            pMsg, param1, param2, UserData);

    CNTL_GetInfo(&status);

    /* Perform any Application UOF Image Processing, such as below.. */

    for (i=0; i<nextSymbol; i++){
       // bind symbol only if matching image_name and symbol_name are in the
microcode
       UcLo_BindSymbol(uclo_handle, boundSymbols[i].image_name,
boundSymbols[i].symbol_name, boundSymbols[i].val);
    }
    return DBGME_SUCCESS;
}
```

The following is an example of a network application that registers the two callback routines, shown in the previous pages, and brings up the Remote Services RPC server. In the event the application comes down, the callback routines are unregistered.

## C Example Code

```
#if (OS != VXWORKS)
int
main (int argc, char **argv)/* Linux or NT start */
#else
void
WBSrvr_Start()/* VxWorks start     */
#endif
{
  int rc;
  CHandlehandle = 0;
  CHandlehandle1 = 0;
  CNTL_Register_TCallBackInfo, CallBackInfo1, CallBackInfo2;

  CallBackInfo.callback = &cntl_CallBackPorts;/* init callback registration 
data */
  CallBackInfo.NotifyMask = CNTL_RESET_NOTIFY | CNTL_START_NOTIFY | 
CNTL_STOP_NOTIFY;
  CallBackInfo.UserData = 0x123;
  CallBackInfo.Priority = 0x123;

  CallBackInfo1.callback = &cntl_CallBackUof;/* init callback registration 
data */
  CallBackInfo1.NotifyMask = CNTL_UOF_LOADED_NOTIFY;
  CallBackInfo1.UserData = 0x12;
  CallBackInfo1.Priority = 0x12;

  printf("Starting WB Server\n");


  rc = CNTL_RegisterCallback(&CallBackInfo, &handle);/* Register callback */
  if (rc != DBGME_SUCCESS)
      printf("CNTL_RegisterCallback FAILED with ret = %x\n", rc);


  rc = CNTL_RegisterCallback(&CallBackInfo1, &handle1);/* Register callback 
*/
  if (rc != DBGME_SUCCESS)
      printf("CNTL_RegisterCallback FAILED with ret = %x\n", rc);


  rc = WbSvr_init();/* start up Remote Services RPC Server */
  if (rc)
    {
   printf("WB Server init failed, WTX Serial Only\n");
    }
  else printf("WB Server Started\n");
  delayMs(1000 * 60 * 60);/* wait a long long time..*/
  CNTL_UnregisterCallback(handle);/* unregister callbacks */
  CNTL_UnregisterCallback(handle1);
exit (0);
    /* NOTREACHED */
}
```