MULTICOM 4 USER MANUAL

TECH-MIG (TECHLIT MIG TO DMS@ST)

DOCUMENT HISTORY

Version	Release Date	Change Designator
C	28-FEB-2011	PROCESS/PRODUCT CHANGE
Updated to address clearquest issues and support the R4.0.4. Approval required by 28.01.11.		

DOCUMENT APPROVAL

Group	Name	Date
STS GROUP	RUSSELL WAYMAN	28-FEB-2011
STS GROUP	DOUG TURNER	08-FEB-2011
STS GROUP	SIMON ELLIOTT	25-JAN-2011
DVD DOCUMENTATION,BR	JANET TEGGIN	27-JAN-2011
DVD DOCUMENTATION,BR	SUZANNE EVANS	27-JAN-2011
BRISTOL, UK	MOSIN MAHMADHANIF	08-FEB-2011
GRENOBLE, FRANCE	LAURENCE MEY	25-JAN-2011
MCDT MARKETING	RAY MOGFORD	25-JAN-2011

REFERENCED DOCUMENTS

Document Id	Document Type	Alternate Number	Document Title
No References			

CUSTOM ATTRIBUTES

Alternate Number	
Dispatcher	MEYL
Working Vault	GNB
Status	ACTIVE
Cycle type	ACTIVE
Change designator	PROCESS/PRODUCT CHANGE
Alternate name	
Application scope	N/A
Application segment	N/A
Category	User manual
Class family	N/A
Classification	N/A
Company	
Iso definition	
Key process	Product Development
Language	
Macro package	N/A
Other RPN	N/A
P&L/BU	N/A
Package family	N/A (NOT APPLICABLE)
Packing	N/A
Print Diff	NO
Product class	N/A
Product pnl	Set-Top-Box
Product subclass	N/A
RPN	MULTICOM-4
Replication sites	
end application	N/A
package	N/A
product division	HVD
product group	HED
publishing scope	Public
Revalidation Date	

STMicroelectronics

Multicom 4

User manual

8182595 Rev C

January 2011

www.st.com

BLANK

User manual

Multicom 4

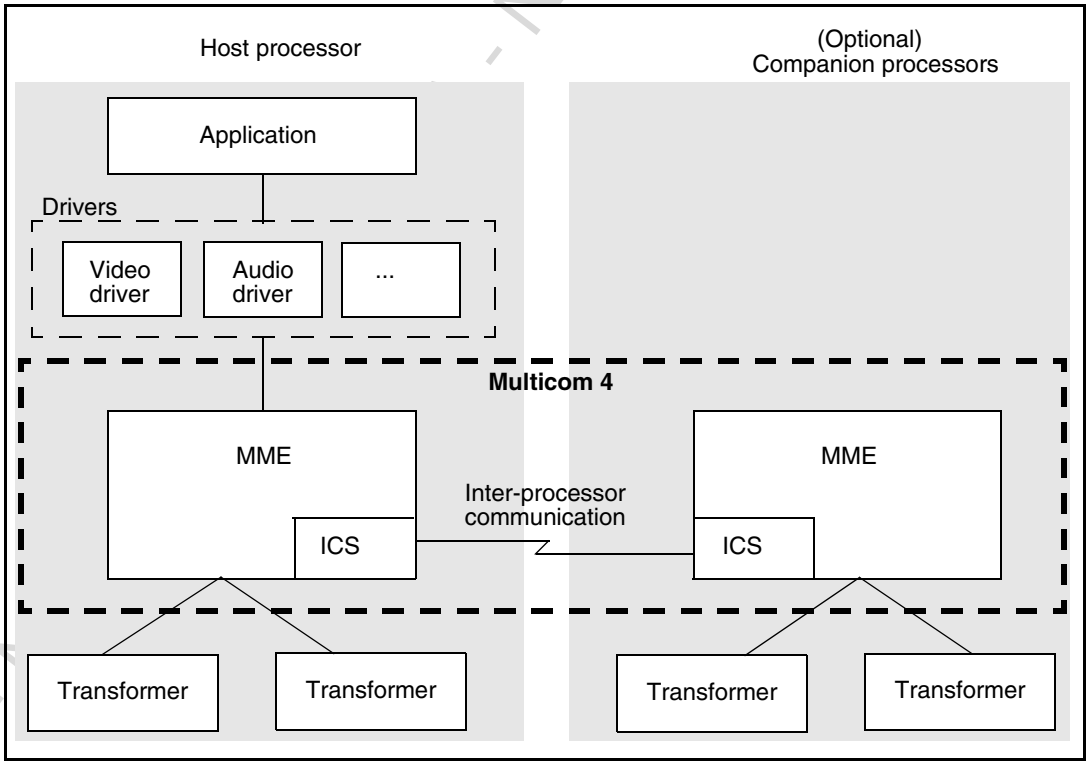
Overview

This document describes Multicom 4, which provides an inter-processor communication system for ST40 and ST200 processors on STMicroelectonics SoCs. Multicom combines the multi-media engine API (MME) and the inter-core system (ICS).

MME provides an API for controlling media transformers, for example, an MPEG2 decoder; which resides on either the local processor or a remote processor.

ICS is a run-time system that provides program execution management and communications between all the CPUs on an ST SoC. ICS replaces the EMBX layer[(a)] in previous versions of Multicom, with a simpler approach to communication, and with the addition of facilities for loading, starting, monitoring and recovering from errors in other CPUs in the SoC.

Figure 1. Multicom 4 overview



a. Extended mailbox communications system (EMBX) - a low-level communications API used to manage communication between CPUs; implemented in the Multicom 3.*x* product.

Contents

Preface

Comments on this manual should be made by contacting your local STMicroelectronics sales office or distributor.

Document identification and control

Each book carries a unique identifier of the form:

nnnnnnn Rev x

where *nnnnnnn* is the document number, and *x* is the revision.

Whenever making comments on this document, quote the complete identification *nnnnnnn Rev x*.

Terminology

The first ST Micro Connect product was named the “ST Micro Connect”; it is now known as the “ST Micro Connect 1” and the term “ST Micro Connect” is used to refer to the family of ST Micro Connect devices. The “ST Micro Connect 2” replaces the “ST Micro Connect 1”. These names are abbreviated to “STMC”, “STMC1” and “STMC2”.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- `sample code`, `keyboard input` and `file names`,
- *variables*, *`code variables`* and *`code comments`*,
- `equations` and `math`,
- **screens**, **windows**, **dialog boxes** and **tool names**,
- **instructions**.

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES,
- PIN NAMES and SIGNAL NAMES.

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF) unless otherwise specified.

- Terminal strings of the language, that is those not built up by rules of the language, are printed in teletype font. For example, `void`.
- Nonterminal strings of the language, that is those built up by rules of the language, are printed in italic teletype font. For example, *`name`*.
- If a nonterminal string of the language starts with a nonitalicized part, it is equivalent to the same nonterminal string without that nonitalicized part. For example, `vspace-`*`name`*.
- Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘`::=`’).
- Alternatives are separated by vertical bars (‘`|`’).
- Optional sequences are enclosed in square brackets (‘`[`’ and ‘`]`’).
- Items which may be repeated appear in braces (‘`{`’ and ‘`}`’).

Mathematical notation

A range of values can be shown using square braces, `[]`, and round braces, `()`. Square braces mean the nearest value is included, and round braces mean the nearest value is not included.

For example:

`[1 .. 3]`	is the values 1, 2, 3
`[1 .. 3)`	is the values 1, 2
`(1 .. 3]`	is the values 2, 3
`(1 .. 3)`	is the value 2 only

Acknowledgements

Linux® is a registered trademark of Linus Torvalds.

1 Building Multicom

1.1 Overview

Multicom 4 is supplied as a set of source files and a set of test suites tailored to support certain ST SoCs. In addition sample board support packages (BSPs) are provided.

This chapter explains how to build the sources and test suites and then to compile and link the Multicom 4 libraries before running the test suites under the required OS. It also explains how to use the sample BSP as a template for creating new BSPs.

Multicom 4 has been developed and tested for target CPUs running both OS21 and Linux (STLinux R2.3 and R2.4).

1.1.1 Code organization

Unpack the Multicom distribution on a suitable host machine which has the appropriate target toolsets installed. See *Section 1.1.2 on page 9* for a list of applicable toolsets. These toolsets must be included in your `PATH`.

The main code is split between two top level directories;

- `source`
- `test`

The source directory contains all the code for the ICS and MME system libraries. The test directory contains many self tests which test each subsystem of ICS and MME.

The Multicom distribution contains a deep directory structure. An overview of these directories is listed in *Table 1*.

Table 1. The distribution directories

Directory	Contains
`docs`	Product documentation.
`source/include`	C header files used by every processor.
`source/src/bsp`	BSP configuration files for each SoC
`source/src/ics`	Complete source code for the ICS implementation
`source/src/mme`	Complete source code for the MME implementation
`test/src/tests/ics`	Test suite for the ICS subsystem
`test/src/tests/mme`	Test suite for the MME subsystem
`lib/<os>/<arch>`	Compiled libraries for each OS and SoC
`bin/<os>/<arch>`	Compiled test codes for each OS and SoC

All makefiles supplied with Multicom use GNU **make** syntax. Therefore in order to rebuild the distribution, run the test suites or build any example, then a version of **make** compatible with GNU **make** must be available.

All target resident code supplied with Multicom is provided in source form to facilitate debugging and porting.

1.1.2 Compiler recommendations

The Multicom 4 system has been tested using the following products;

- ST200 Micro Toolset R6.2.1, R6.4.0, R6.5.0, R7.0.0
- ST40 Micro Toolset R4.4.0, R5.0.0
- ST Micro Connection Package R1.5.0
- STLinux R2.3, R2.4

It is recommended these versions or later of the compiler toolsets are used[b].

*Note: Several issues have been found when using the ST200 Micro Toolset R6.3.0. In addition issues have been found using the GNU **make** tool under Windows, so GNU **make** 3.81 or later is recommended.*

1.2 Building the library code

The source directory must be built first, before any of the tests are built. For OS21 builds, makefiles are provided for each CPU architecture.

For example, to build Multicom 4 for the MB671/STi7200, issue the following commands;

```
cd source
make -f Makefile.st40
make -f Makefile.st200
```

To build the kernel module for Linux, requires a command sequence of the form:

```
export KERNELDIR=<path_to_original_linux_kernel_sources>
export O=<path_to_built_linux_kernel_sources>
```

For example, issue the following commands (adjusting the kernel paths as necessary for the desired target kernel):

```
export KERNELDIR=/usr/src/linux-sh4-2.6.23.17_stm23_0122
export O=/lib/modules/linux-sh4-2.6.23.17_stm23_0122_mb671

cd source
make ARCH=sh CROSS_COMPILE=sh4-linux-
```

To build the Linux userspace MME interface library, issue the following commands:

```
cd source
make -f Makefile.linux
```

b. Please contact your ST FAE or ST support centre for information about Multicom 4's compatibility with later versions of ST Toolsets.

1.3 Building the test code

Having built the source tree, the tests can now be built.

For example, under OS21 on an MB671/STi7200, issue the following commands:

```
cd test
make -f Makefile.st40 PLATFORM=mb671
make -f Makefile.st200 PLATFORM=mb671_audio0
make -f Makefile.st200 PLATFORM=mb671_video0
make -f Makefile.st200 PLATFORM=mb671_audio1
make -f Makefile.st200 PLATFORM=mb671_video1
```

To build the kernel test modules for Linux, issue the following commands (adjusting the kernel paths as necessary for the desired target kernel):

```
export KERNELDIR=/usr/src/linux-sh4-2.6.23.17_stm23_0122
export O=/lib/modules/linux-sh4-2.6.23.17_stm23_0122_mb671

cd test
make ARCH=sh CROSS_COMPILE=sh4-linux-
```

To build the userspace test MME executables for Linux, issue the following commands:

```
cd test
make -f Makefile.linux
```

1.4 Compiling and linking against the Multicom 4 libraries

In order to compile a test application using the Multicom 4 libraries you need to link against the binary library.

For example, to compile an OS21 ST40 test program for the MB671/STi7200 you could issue a command similar to the following;

```
sh4gcc -mboard=mb671 -mruntime=os21 -o test40.out test.c\
       -Isource/include   \
       -Lsource/lib/os21/st40 -lmme -lics \
       -Lsource/lib/os21/st40/stx7200/st40 -lics_bsp\
       -rmain -Wl,--wrap=_write_r
```

For the Audio0 ST231 CPU the command would be:

```
st200cc -mboard=mb671_audio0 -msoc=sti7200 -mcore=st231\
       -mruntime=os21 -o test200.out test.c\
       -fno-dismissible-load\
       -Isource/include\
       -Lsource/lib/os21/st231 -lmme -lics\
       -Lsource/lib/os21/st231/stx7200/audio0 -lics_bsp \
       --rmain -Wl,--wrap=_write_r
```

To compile a Linux userspace ST40 MME test program, issue a command, similar to the following:

```
sh4-linux-gcc -o test.out test.c
       -Isource/include -Lsource/lib/linux/st40 -lmme
```

Note: 1 The ST231 `-fno-dismissible-load` *option is only necessary if the companion code has not correctly enabled SCU speculation for all possible memory locations that will be accessed by the ST200.*

2 The linker `--wrap=_write_r` *argument allows the ICS system to intercept all text output from the target executable and log it into the CPU cyclic log buffers, for each CPU.*

1.5 Building dynamic modules for use with Multicom 4

ICS includes infrastructure that allows code modules to be dynamically loaded into the running CPUs (OS21 only). See the *Chapter 5: Overview of the inter-core system (ICS) on page 106* for more details.

In order to compile a compatible ELF module a command similar to the following can be used;

```
st200cc -mcore=st231 -mruntime=os21\
       dyn.c -o dyn.out\
       -Isource/include\
       -nostdlib -fpic --rlib -fvisibility=protected\
       -Wz,-z,max-page-size=1
```

Note: The `-Wz,-z,max-page-size=1` *option is a temporary workaround for a bug detected in the ST200 R6.3.0 toolset.*

1.6 Debugging support

A lot of debug tracing is included in the Multicom 4 code, but by default these messages are not compiled into the libraries. In order to be able to use the debug logging, the library code will need to be rebuilt using the following **make** command line or environment option:

```
# ICS debugging
export DEBUG_CFLAGS="-DICS_DEBUG"

# MME debugging
export DEBUG_CFLAGS="-DMME_DEBUG"
```

Debug libraries can be built alongside the default non-debug libraries by adding the following option to the **make** command line:

```
MULTILIB="/dbg"
```

This causes the debug enabled libraries to be placed in a separate `dbg` subdirectory in the build tree. To cause the test executables to be linked against these libraries, the same **make** command line option should be used when compiling the tests.

As soon as the debug enabled libraries have been built, debug logging can be configured dynamically by using the `ics_debug_flags()` and `MME_DebugFlags()` API calls. The debug logging level can also be configured statically by specifying the following **make** command line or environment option:

```
# ICS debugging
export DEBUG_CFLAGS="-DICS_DEBUG -DICS_DEBUG_FLAGS=1"

# MME debugging
export DEBUG_CFLAGS="-DMME_DEBUG -DMME_DEBUG_FLAGS=1"
```

Where `ICS_DEBUG_FLAGS` and `MME_DEBUG_FLAGS` can be set to a bitmask for each subsystem for which logging is required. See *Section 5.9: Debug logging support on page 119* for further details.

For the Linux kernel modules the ICS and MME debugging level can be set by using the `ics_debug_flags()` and `MME_DebugFlags()` function calls or by setting the module parameters, see *Section C.1: Debugging assertions and logging on page 208*.

1.6.1 Debug logging

By default all ICS and MME debug log messages are logged to a cyclic buffer. Under Linux these logs can be easily accessed using the `procfs` filing system. For example, to dump out the current log for CPU #1, issue the command:

```
cat /proc/ics/cpu01/log
```

This command dumps out all the messages logged since the last time the log was dumped. If the logging output has exceeded the size of the cyclic buffer then only the newer messages are seen.

Note: By linking against debug built libraries the size of the cyclic log buffer is greatly increased.

1.7 Running the tests under OS21

Running the OS21 based tests is just a matter of loading and running the primary test executable. For tests that make use of multiple CPUs the corresponding executables are loaded directly by the test harness. All test executables are compiled into the `test/bin` directory and should be executed from the top level test directory.

For example running a test on an MB671 may use a command such as;

```
cd test
sh4xrun -t stmc:mb671:st40,tapmux_mux=1 \
  -e bin/os21/st40/mb671/st40/ics/channel/channel00_main.elf
```

Note: The `.elf` files are simply stripped versions of the `.out` files to reduce load times.

1.8 Running the tests under Linux

To run the test modules under Linux, first the appropriate Linux kernel needs to be booted. Log into the target CPU via a root equivalent account and then the *`ics.ko`* module should be inserted using;

```
cd test

insmod ../source/src/ics/ics.ko init=0 debug_flags=0
insmod ../source/src/ics/ics_user.ko
insmod ../source/src/mme/mme.ko init=0 debug_flags=0
insmod ../source/src/mme/mme_user.ko
```

Finally the individual test module can be loaded using:

```
insmod src/tests/ics/channel/channel00_main.ko
```

However, the tests which require further executables to be loaded require that all binaries are copied into `/lib/firmware` on the target filing system. Because `/lib/firmware`

does not support directory hierarchies, each file needs to be renamed to include the CPU name.

For example, on an MB671 we might do:

```
cp bin/os21/st231/mb671/audio0/ics/channel/channel11_main.elf \
     /lib/firmware/channel11_main.audio0.elf
```

Note: 1 The Linux test suite fails if these binaries are not present or have a different filename.

2 The Linux `/lib/firmware` file loading system has a maximum filename length of 32 characters including the terminating '`\0`'.

1.9 BSP configuration

Currently the ICS is supplied with BSPs for a limited number of SoCs.

Appendix A: ICS board support package on page 195 describes how to create a BSP for other SoCs.

2 Using the MME API

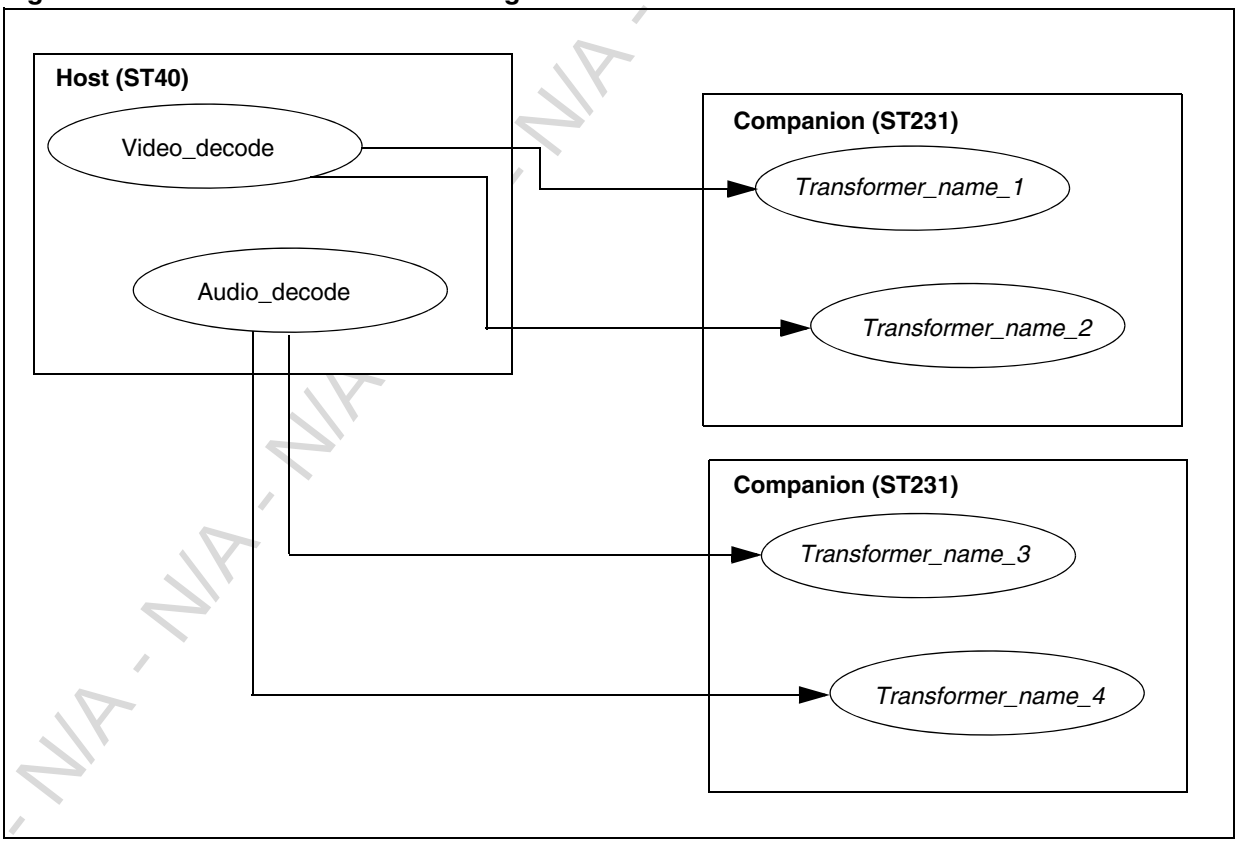
2.1 Overview

The MME API provides a means for an application program running on the host processor to control and manipulate a codec or similar media transformer running either on the same processor or on a different companion processor. The aim of a companion processor is to assist the host in transforming data in real time and it communicates with the host using the ICS communication interface, see *Chapter 5: Overview of the inter-core system (ICS) on page 106*. Both host and companion transformers may, optionally, make use of hardware accelerators to off-load some or all of the work. The MME API remains the same independent of the location or type of the transformer, effectively hiding the (potentially complex) structure of the system from the application. The MME API is intended to form part of the driver layer of typical multimedia software stacks. See *Figure 1 on page 1*.

2.1.1 Transformers and transformer instances

Within the MME system the division of roles between the host and companion processors is logically described by what are known as transformers. Transformers are named software modules that normally reside in the companion processors. A companion CPU will register a transformer for each codec or media transformation it can perform (see *Section 2.3.2: Registering transformers on page 20*). The host will then use the symbolic name of each of the transformers it requires, to lookup and instantiate them, see *Figure 2*.

Figure 2. Transformer instances - single host



When a transformer is instantiated, the abstract transformer is combined with parametric and state information; it is then capable of processing data. This is called a **transformer instance**.

Typically, transformers that rely on hardware accelerators can only have one instance at a single point in time due to there only being one accelerator. However, for software transformers, it is unusual for anything other than available memory to limit the number of instances of a particular transformer.

2.1.2 Multi-hosting support

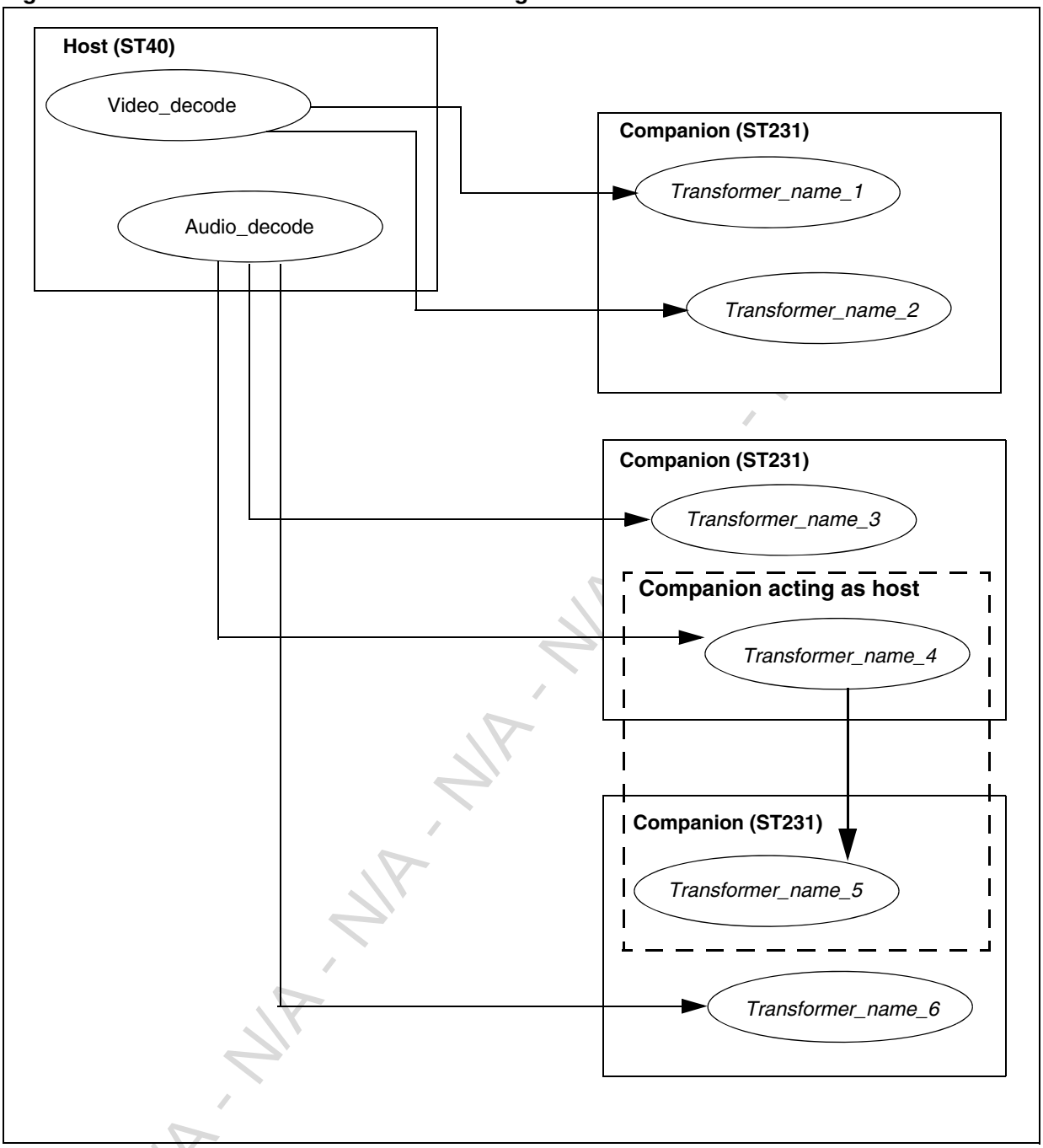
This version of Multicom also supports a usage model known as **Multi-hosting**. Traditionally in an MME system there would have been one host CPU (for example, an ST40) and multiple companion CPUs (for example, ST231s), however newer SoCs now incorporate multiple ST40 CPUs. Multi-hosting means that any CPU in the MME system can act as a host and hence instantiate and issue commands to transformers. So in practise any CPU in an MME system can act either as a host or a companion or both simultaneously. The only exception to this is that a program in Linux user space cannot register transformers.

Multi-hosting can be used to build complex systems where decode and compute are off-loaded to one or more of the companions CPUs. Here are a couple of example usage scenarios.

- On a multi ST40 SoC such as the STi7141 both the ST40 CPUs could act as hosts and issue commands to any one of the three ST231 CPUs simultaneously.
- On an SoC such as the STi7200, which has four ST231s, the Audio decode could be partitioned and off-loaded to both Audio CPUs, as shown in *Figure 3*. This can be done transparently to the application running on the ST40 host by 'nesting' transformations.

  For example, the host CPU issues an Audio decode transformation command to the Audio codec which is registered on the Audio0 CPU, this transformer has in turn instantiated a transformer registered on the Audio1 CPU. As each command arrives, the Audio codec running on Audio0 issues transformation commands to Audio1 to off-load some of the compute. Once these complete it can then complete the original transformation command from the host.

Figure 3. Transformer instances: multi-hosting



2.1.3 Commands and events

Transformer instances are controlled by sending them commands. Each command is a self-contained unit of work consisting of a due time, a command code, some transformer specific parametric information and the data buffers to be transformed, by the command. All commands of the same priority are executed in due time order.

Note: 1 The priority of a command is inherited from the transformer instance with which it is associated. The priority of a transformer instance is supplied by the application when the transformer is instantiated.

2 Because commands are executed on different processors and, potentially, can be deferred for execution by different hardware accelerators, this does not imply that across the system as a whole, all commands will be issued or complete in due time order.

Each command is associated with a status structure that, among other things, provides the unique identifier by which the command can be managed together with an indication of the command's current state.

Commands are submitted for execution asynchronously, that is, the function to issue the command completes successfully before the command has completed.

The MME can generate events when a command completes or fails. Event notification may be optionally enabled by the application programmer when a command is submitted. Events are delivered to the application by using callbacks.

2.1.4 Callbacks

A callback function and application-specific callback data is associated with a transformer when a transformer is instantiated. When a command is sent to a transformer, the application can choose whether or not it will be notified of any events associated with the command, by the associated callback.

2.1.5 Due time

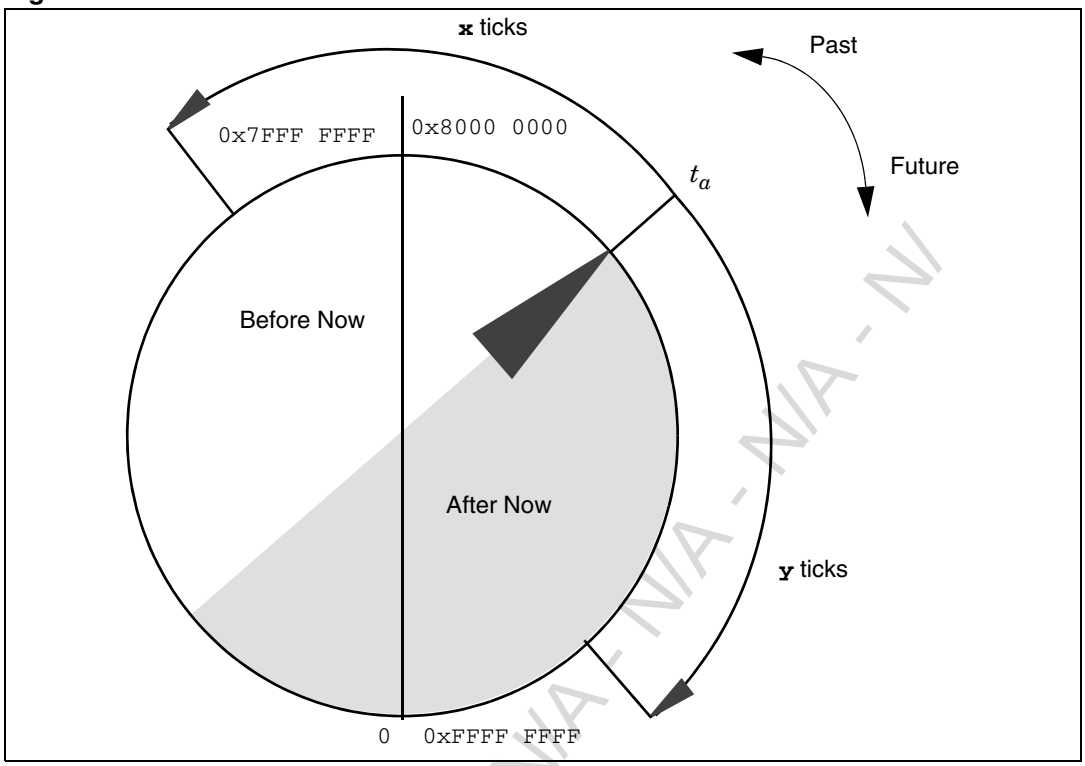
The due time is used by the MME implementation to determine in what order to process commands. The command queue for each transformer is maintained in due time order and when a command is dispatched all the queues are examined and the one with the lowest due time is selected for execution.

Note: The due time is only relevant to `MME_SET_GLOBAL_TRANSFORM_PARAMS` *and* `MME_TRANSFORM`; `MME_SEND_BUFFERS` *commands are executed in strict FIFO order and can pre-empt currently running commands. See* *Section 2.8: Fault detection and recovery on page 28*.

Neither the MME host nor the companion is aware of the current system time. This leaves the choice of what time unit to use entirely at the application designers discretion. In most cases using the host processor's system clock is recommended. Although the MME implementation does not know the unit of time, it does know that as time progresses the due time will eventually reach `0xffffffff` and overflow. For this reason when due times are compared it is not a simple magnitude comparison. Instead the times are arranged, such that t_{after} - t_{before} is less than `0x7fffffff`.

Figure 4 shows what this comparison means in practice by showing how t_a will be compared against all possible 32-bit values.

Figure 4. Time arithmetic



When due times are exactly equal then the least recently issued command will be executed first. This permits commands to be executed in strict FIFO order if their due times are always the same value; zero is a good candidate value in this case although any value can be used.

There are three obvious ways an application may choose to utilize the due time:

- As constant value across all transformers. This results in FIFO scheduling within a transformer and round robin scheduling among transformers.
- As unique constant values. This results in FIFO scheduling within a transformer and prioritized scheduling among transformers. This differs from normal prioritized scheduling because low priority transforms will not be pre-empted. This may yield slightly better utilization of processor bandwidth at the expense of latency.
- As true due time. This results in due time scheduling within all commands irrespective of which transformer queue they appear on.

2.1.6 Transformer priorities

The due time mechanism allows commands to be executed in a deterministic sequence. However, an application may require short-duration transforms (such as a series of audio frame decodes) to complete while a lengthy transform operation (such as a JPEG decode) is being handled by another transformer instance.

To facilitate this MME supports five transformer priorities. A priority is assigned to a transformer instance when the instance is created. Transformer priorities are mapped onto the underlying operating system thread priorities; an **execution thread** is created for each priority for which a transformer is instantiated.

Therefore a transform command executing on a high priority transformer instance takes precedence over a command executing on a lower priority transformer instance. Commands at a particular priority are submitted sequentially to their transformer instances in due time order.

2.1.7 Structure size

The MME API uses MME structures to pass data. Typically there is a size field `StructSize` which must be set to the size of the structure in bytes, see *Section 4.2: MME constants, enums and types on page 75*.

2.2 Summary of MME facilities

The following is a summary of the main facilities provided by the MME API.

- MME initialization, see *Section 2.3 on page 20*

  MME provides API calls to allow the application to set up MME and to register transformers.
- Managing transformer lifetimes, see *Section 2.4 on page 21*

  Transformer instances can be created or destroyed using the MME API. It is also possible to examine the capabilities of a transformer before it is instantiated.
- Buffer and cache management, see *Section 2.5 on page 22*

  Data buffers are used extensively by MME to transport unstructured data between the application and the transformers. MME provides API calls to allocate and manage data buffers as well as subdividing them. Flags are provided to influence cache management of data buffers.
- Application and transformer specific data, see *Section 2.6 on page 26*

  MME provides a mechanism for passing application-specific or transformer data to and from the transformer.
- Issuing commands, see *Section 2.7 on page 26*

  The central function of the MME API is to perform transformations. Three types of commands may be sent to a transformer, as described in *Section 2.8 on page 28*. *Section 2.10 on page 30* describes the difference between a **frame-based** transformer and a **stream-based** transformer.

The chapter finishes with a description of linking and loading issues for different operating systems, *Section 2.11 on page 31*.

2.3 Initialization

Initialization of a system is divided into two stages:

1. Initializing MME (which presupposes that the ICS is initialized.)
2. Registering transformers, if they are being used

It is essential that all host and companion processors in a system are initialized. However, some stages of the initialization may be omitted for particular operating environments supported by MME:

- For multi-processor OS21 systems, all steps are required for companion processors. Registering transformers is optional for host processors.
- For Linux **kernel space** operation, MME is initialized automatically when the MME module is loaded. Registering local transformers is optional.

  Section 2.11.2: Linux on page 31 contains further details about loading Linux kernel modules.

Note: It is not possible to register transformers in Linux user space.

2.3.1 Initializing MME

The MME library is initialized using the following function:

```
MME_ERROR MME_Init(void)
```

The MME library must be loaded and initialized on each processor and user space process in the system.

Note: The Linux kernel MME implementation automatically calls `MME_Init` *during module load.*

No other API can be called until the library is initialized.

In a system where multiple threads use MME, it is permissible for each thread to call `MME_Init`. The first call performs initialization, returning `MME_SUCCESS` if no error occurs. Any subsequent calls simply return `MME_ALREADY_INITIALIZED`. That is, a second call to `MME_Init` does not re-initialize MME.

Note: Calls to `MME_Init` *are not counted. Thus particular care must be taken de-initializing MME when sharing the MME between multiple threads.*

2.3.2 Registering transformers

A transformer is registered with a name and associated function pointers by using the following function:

```
MME_ERROR MME_RegisterTransformer(
      const char *name,
      MME_AbortCommand_t abortFunc,
      MME_GetTransformerCapability_t getTransformerCapabilityFunc,
      MME_InitTransformer_t initTransformerFunc,
      MME_ProcessCommand_t processCommandFunc,
      MME_TermTransformer_t termTransformerFunc)
```

Each of the functions pointed to is described in detail in *Chapter 3: Writing an MME transformer*.

2.3.3 Example

This section provides examples of how MME is started on a host CPU and on a companion CPU. The examples illustrate the startup sequence for the host application and for the companion. It is assumed that the operating system has been started on each CPU.

Note: For brevity, return code checks have been omitted from the examples.

Host-side example

```
/* Initialize ICS */
err = ICS_cpu_init(0);

/* Intitialize the MME system for a host */
res = MME_Init();

/* Init transformer */
res = MME_InitTransformer("com.st.mcdt.mme.test_transformer",
                                &initParams, &transformerHandle);

/* Send a transform command */
res = MME_SendCommand(transformerHandle, MME_TRANSFORM, ...);
res = MME_TermTransformer(transformerHandle);
```

Companion-side example

```
/* Initialize ICS */
err = ICS_cpu_init(0);

/* Intitialize the MME system for a companion */
res = MME_Init();

/* Register the transformers active on this CPU */
res = MME_RegisterTransformer("com.st.mcdt.mme.test_transformer",
                                abortFunc, getCapabilityFunc,
                                initFunc, processCommandFunc, termFunc);
```

2.4 Managing transformer lifetimes

Transformer instances can be created and destroyed using the following functions:

```
MME_ERROR MME_InitTransformer(
     const char *name,
     MME_TransformerInitParams_t *params_p,
     MME_TransformerHandle_t *handle_p)

MME_ERROR MME_TermTransformer(MME_TransformerHandle_t handle_p)
```

The `name` argument specifies the name of the previously registered transformer.

`params_p` is used to specify one of five priority levels for the transformer together with details of the callback function used to communicate any events associated with this transformer and its commands. Additionally the parameter structure may contain a pointer to transformer specific parameters containing any initial state the transformer may require. See *Section 2.6: Application and transformer specific data on page 26* and *Section 3.7: Parameter passing on page 42*.

If `MME_InitTransformer` returns successfully then `handle_p` is supplied with a handle used to issue commands and terminate the transformer. Once initialized, a transformer can execute an arbitrary number of commands before finally being terminated.

`MME_TermTransformer` is used to destroy a transformer instance thus freeing any resources used by the transformer.

Note: It is not possible to terminate a transformer if there are any outstanding commands pending. If this is attempted an error is returned.

2.4.1 Querying the capabilities of a transformer

It is sometimes useful to examine the capabilities of a transformer before it is instantiated, either for error checking or to ensure the correct transformer is being used. MME allows transformers to publish their capabilities without requiring a handle. This enables transformers to be examined before any calls to `MME_InitTransformer`.

The following function is used for this purpose:

```
MME_ERROR MME_GetTransformerCapability(
     const char *transformerName,
     MME_TransformerCapability_t *transformerCapability)
```

The transformer is able to describe its preferred input and output formats together with its version number. Transformer specific details can also be copied into a user-supplied buffer.

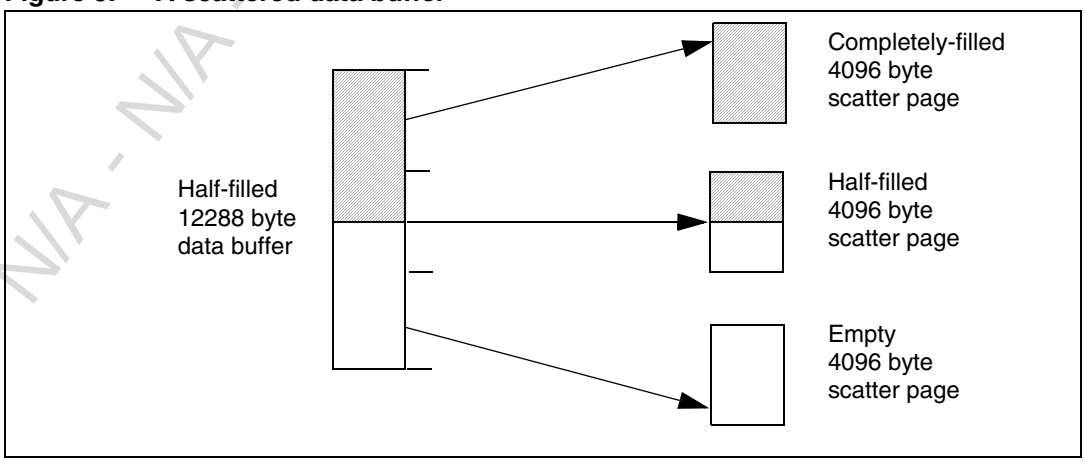
2.5 Buffer and cache management

Data buffers are used throughout MME to transport unstructured data between the application and transformers. In this case, unstructured is used to mean that data has identical representation on all processors regardless of endianness or similar concerns; it is a simple stream of bytes. A data buffer describes a logical group of memory locations that contain or are intended to contain media data. A data buffer is represented by the structure `MME_DataBuffer_t`.

Each data buffer is composed of one or more scatter pages. A scatter page describes a single sequential group of memory locations, or more specifically, a base pointer and a size. A data buffer comprised of a single scatter page is a **linear** buffer while a data buffer consisting of multiple scatter pages is a **scattered** buffer.

A scatter page is represented by the structure `MME_ScatterPage_t`.

Figure 5. A scattered data buffer



Note: Although both linear and scattered buffers are properly handled by the MME API, some transformers are not able to efficiently support scattered buffers. For example, using scattered buffers makes it difficult to delegate work to accelerators that only support linear DMA.

2.5.1 Allocating data buffers

Data buffers can be allocated and freed using the following functions:

```
MME_ERROR MME_AllocDataBuffer(
     MME_TransformerHandle_t Handle,
     MME_UINT Size,
     MME_AllocationFlags_t Flags,
     MME_DataBuffer_t **DataBuffer_pp)


MME_ERROR MME_FreeDataBuffer(MME_DataBuffer_t *DataBuffer_p)
```

Buffers allocated using `MME_AllocDataBuffer` come from a common buffer pool which is of a fixed size. The size of this buffer pool can be varied by adjusting the tuneable value `MME_TUNEABLE_BUFFER_POOL_SIZE` before calling `MME_Init`.

All buffers allocated from the common buffer pool are guaranteed to be accessible by all transformers, and both cached and uncached translations can be requested by making use of the `Flags` argument.

If the user is intending to manually manage the MME data buffers then the default buffer pool size can be set to a small value. However, even in this case, the common buffer pool will be used for the MME command meta-data and hence should be sized accordingly.

2.5.2 Manually managing data buffers

MME permits any memory locations accessible by the host to be used in data buffers and scatter pages.

Note: Linux User Mode applications are the exception to this, see Section 2.5.4.

This allows most applications to manage memory for themselves and construct data buffers and scatter pages as required.

All user managed buffer memory must first be registered with MME by using the `MME_RegisterMemory` function. Failure to do this causes the `MME_SendCommand` operation to fail. If cached and uncached translations of this buffer memory are required, then memory registration calls with both cached and uncached translations of the memory region must be made.

Memory can be registered and deregistered using the following functions:

```
MME_ERROR MME_RegisterMemory (MME_TransformerHandle_t Handle,
     void *Base,
     MME_SIZE Size,
     MME_MemoryHandle_t *Handle_p)


MME_ERROR MME_DeregisterMemory (MME_MemoryHandle_t Handle)
```

Note: Cached buffers that are not aligned to the largest cache line size in the system pose significant problems because this makes writes by the companion CPU to those addresses unsafe.

2.5.3 Subdividing a data buffer

The application may divide the scatter pages returned by `MME_AllocDataBuffer()` into application-oriented scatter pages, so long as the divided pages reside entirely within the allocated pages.

An application must not make assumptions about the number of scatter pages returned by `MME_AllocDataBuffer` unless the flag `MME_ALLOCATION_PHYSICAL` is specified, in which case a single page is returned.

A simplified example of dividing a physical scatter page is shown below. This example takes the scatter page returned by `MME_AllocDataBuffer` and divides it into `NUM_SCATTER_PAGES` scatter pages:

```
MME_DataBuffer_t* dataBuffer;
MME_ScatterPage_t* origPage;
MME_ScatterPage_t scatterPage[NUM_SCATTER_PAGES];
int newPageSize;
unsigned char* pageBase;


/* Allocate a buffer of 'size' bytes */
MME_AllocDataBuffer(hdl, size, MME_ALLOCATION_PHYSICAL, &dataBuffer);


/* Keep a record of the original scatter page array */
origPage = &dataBuffer->ScatterPages_p;

/* Calculate size of each new scatter page - ignore the remainder bytes */
newPageSize = origPage->Size/NUM_SCATTER_PAGES;
pageBase = origPage->Page_p;


/* Set the data buffer to use the new array of scatter pages */
dataBuffer->ScatterPages_p = scatterPage;


for (i=0; i<NUM_SCATTER_PAGES; i++) {
    dataBuffer->ScatterPages_p[i].Page_p = pageBase;
    dataBuffer->ScatterPages_p[i].Size = newPageSize;
    dataBuffer->ScatterPages_p[i].BytesUsed = newPageSize;
        pageBase += newPageSize;
  }
/* Now use the data buffer with the scatter pages */
  ...

/* Free the data buffer when no longer required */


dataBuffer->ScatterPages_p = origPage;
MME_FreeDataBuffer(dataBuffer);
```

2.5.4 Data buffers in Linux user mode

A Linux user application writer should endeavor to use `MME_AllocDataBuffer()` to allocate a data buffer for use by `MME_SendCommand()`. If this is not feasible because a data buffer has been allocated in kernel space by another agent (for example a video

driver), the corresponding user space address of this buffer may be used in the `Page_p` field of a scatter page (`MME_ScatterPage_t`).

`MME_SendCommand` ensures that the physical pages that comprise the buffer:

- belong to the calling process
- are physically contiguous

If either of these criteria are not met, `MME_SendCommand` returns `MME_INTERNAL_ERROR`.

The cacheability of these contiguous pages is determined from the cacheability flag within the Virtual Memory Area (VMA), in which the pages reside.

2.5.5 Cache management

When a data buffer is held in cached memory, MME is forced to make a pessimistic assumption regarding whether it is held in a particular processor's cache, in order to guarantee correctness. In many cases, the application is in a position to provide hints that can reduce this pessimistic behavior. These hints have no effect if memory is uncached, and can therefore be applied by an application even for affinity-allocated memory (see *Section 2.5.1: Allocating data buffers on page 23*).

For example, a buffer populated by an incoherent DMA peripheral (and not subsequently read by the CPU) is known not to be in a processor's cache. It is therefore wasteful to spend time flushing such a buffer from memory.

For this reason, each MME scatter page can be marked with the cache management hints shown in *Table 2 on page 25*, prior to being made available to the host.

Table 2. MME_ScatterPage_t FlagsIn and FlagsOut

Flag	FlagsIn(1)	FlagsOut(2)	Description
`MME_DATA_CACHE_COHERENT`	✓	✓	For a host this means that all input buffers are coherent with memory and the output buffers are not present in the cache. For a companion this means that **no** buffers are present in the cache. This directs MME to avoid any unnecessary cache invalidations or cache flushes.
`MME_DATA_TRANSIENT`	✓		Data is reused by the companion without being read by another processor or hardware accelerator. This directs MME not to flush the companion's data cache.

1. The `FlagsIn` field is set by the host application before the command is issued to the MME.
2. The `FlagsOut` field is set by the companion transformer before notifying the host that the transform is complete.

2.6 Application and transformer specific data

MME provides a mechanism for passing application-specific or transformer data to and from the transformer. When such data is to be passed it is specified by an address (of type `MME_GenericParams_t`) at which the data starts and a length in bytes. A mechanism for managing the portability of such data is described in *Section 3.7: Parameter passing on page 42*.

2.7 Issuing commands

Commands are issued using the following function:

```
MME_ERROR MME_SendCommand(
     MME_TransformerHandle_t Handle,
     MME_Command_t *CmdInfo_p)
```

`MME_SendCommand` is asynchronous; it returns before the command has completed processing. For this reason it is possible to examine the state of the command before it has completed.

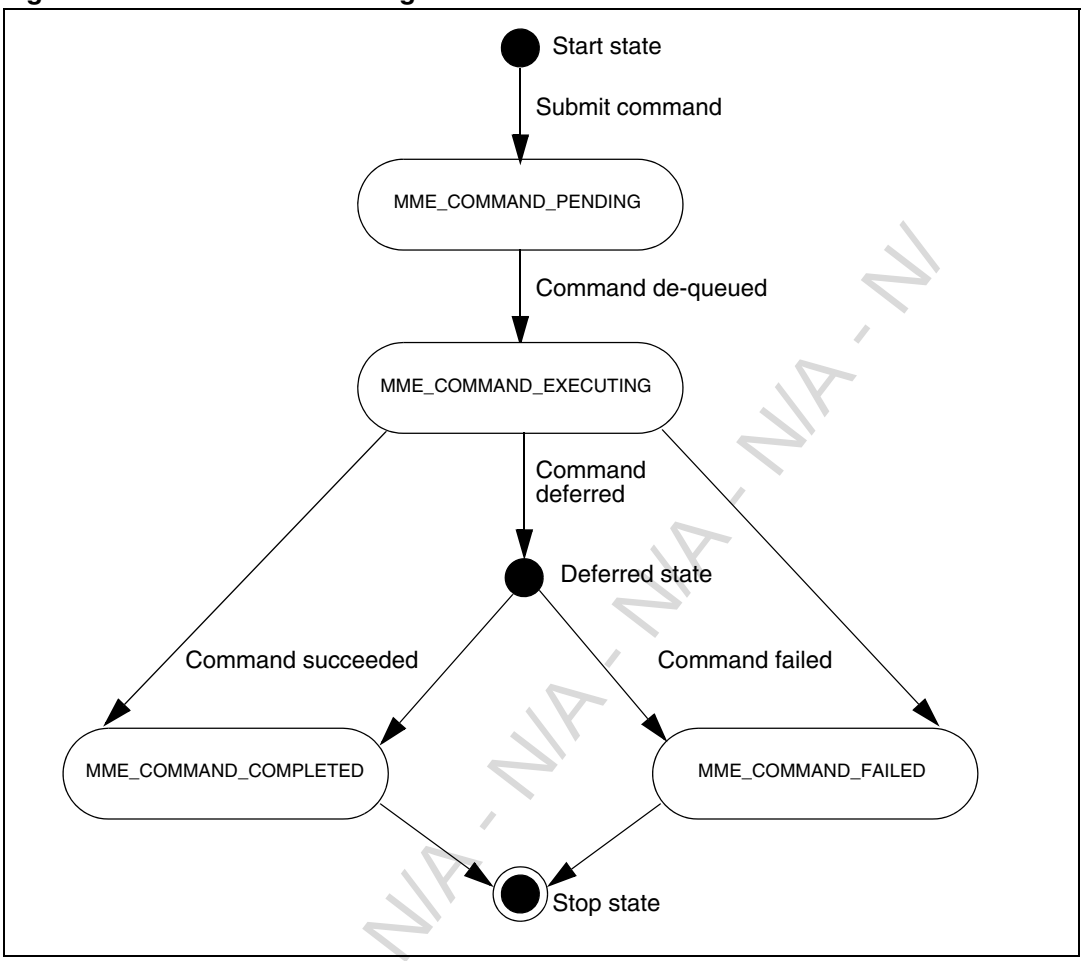
The application must fill in several fields of the `MME_Command_t` structure:

- `StructSize` - see *Section 2.1.7: Structure size on page 19*
- `CmdCode` - with the command to perform - see *Section 2.4: Managing transformer lifetimes on page 21*
- `CmdEnd` - to specify whether events such as a “command completion” cause the callback function to be called
- `Due time` - see *Section 2.1.5: Due time on page 17*

  (zero for all commands ensures FIFO processing of commands)
- `NumberInputBuffers` - the number of input data buffers
- `NumberOutputBuffers` - the number of output data buffers
- `DataBuffers_p` - a pointer to an array of pointers to the input buffers and output buffers. The input buffer pointers must precede the output buffer pointers in this array
- `Param_p` and `ParamSize` - with command specific parameters. These should be set to `NULL` and zero respectively if there are no parameters. See *Section 2.6: Application and transformer specific data on page 26* and *Section 3.7: Parameter passing on page 42*.

The state is held in the `MME_CommandStatus_t` structure within the `MME_Command_t` structure. The command changes from state to state as shown in *Figure 6 on page 27*.

Figure 6. Command state diagram



When a command is issued it will be in the `MME_COMMAND_PENDING` state for as long as it is enqueued, waiting for processing time to become available.

The command's transition to the `MME_COMMAND_EXECUTING` state, occurs when it is allocated processor time.

Note: When a command is scheduled for execution on a companion processor, it is not possible to distinguish between the `MME_COMMAND_PENDING` *and* `MME_COMMAND_EXECUTING` *states due to the requirement of minimizing communication between the host and companion. All commands that are scheduled on a companion processor appear to the application to be in the* `MME_COMMAND_EXECUTING` *state.*

If execution has terminated, either by normal completion of processing or due to an error, the state changes to one of the final states: `MME_COMMAND_COMPLETED`, or `MME_COMMAND_FAILED`. Callbacks occur whenever execution of a command completes or blocks due to command overflow or underflow conditions, which are described in *Section 2.10.2: Stream-based and hybrid operation on page 30*.

Executing commands enter the deferred state when a transformer delegates processing to the hardware block. (The deferred state is a conceptual state which is not observable by the host). When a command is deferred, subsequent commands are executed by the processor while the original command is executed by an independent hardware accelerator. A

deferred command can either proceed directly to one of the final states or re-enter the MME_COMMAND_PENDING state and wait for processor time.

Note: Applications should never repeatedly poll the state of a command as this has the potential to deny other threads the use of the processor. Instead of polling, the application should utilize callbacks.

2.7.1 Aborting commands

It is possible for the application to request that a command is aborted. This is achieved using the following function:

```
MME_ERROR MME_AbortCommand(MME_CommandId_t CmdId)
```

This function is asynchronous, it returns successfully before the request for abortion has been passed on.

An abort is, effectively, a request for a command to immediately enter the MME_COMMAND_FAILED state. As such the application will be notified if an abort was possible by the command entering this state. The application will receive a callback, assuming these are requested. If the command could not be aborted it will complete as normal entering either the MME_COMMAND_COMPLETED state or the MME_COMMAND_FAILED state with a different error code.

Although a command in the MME_COMMAND_PENDING state can always be aborted, it is not always possible for commands in the MME_COMMAND_EXECUTING or deferred[c] states to be aborted, as support for abort from these states is transformer-specific.

Note: Since commands can asynchronously move from state to state, it is possible for a command to change state between the application observing that command in the MME_COMMAND_PENDING state and issuing the abort. As such the application should always consult the state of the aborted command before assuming success.

2.8 Fault detection and recovery

MME and the underlying ICS system provide simple facilities to aid the device driver writer in coping with unexpected failures of the companion CPUs.

In particular, CPU crashes are detected, and any outstanding commands and transformers that are hosted on the failed CPU are terminated. Further facilities are provided to allow the programmer to 'Ping' the remote transformers periodically (See *MME_PingTransformer on page 66*), allowing them to check that the target transformer has not hung or got stuck in an infinite loop.

Using these facilities the device driver writer should be able to detect and recover from an unexpected companion CPU failure. Generally this involves tidying up all the device driver state associated with the failed CPU and transformer instances, and then reloading and restarting the code on the companion CPU.

Multicom cannot recover from a failure of the host CPU or errors which cause a system wide lockup. Also, if a companion failure results in corruption of memory belonging to the host CPU, then it may fail subsequently.

c. The technicalities of aborting a deferred command is discussed in detail in *Section 3.4.2: Deferred commands on page 38*.

2.9 Types of commands

There are three types of command that can be issued through a call to `MME_SendCommand`:

- transform data
- provide additional data buffers
- alter global parameters

A command type is selected by setting the `MME_CommandCode_t` field in the `MME_Command_t` structure.

These types are described in *Section 2.9.1* through to *Section 2.9.3*.

2.9.1 Transforming data

Data transformations are at the heart of the MME API. All the other API calls and transformer commands exist solely to assist with data transformations.
The command code for data transformations is `MME_TRANSFORM`.

Wherever possible, data transformations are supplied with both input and output buffers as part of the transform command.

Most complex transformers also take some transformer specific command parameters. Typically this parametric information is purely local affecting only the current transformation although it is quite legitimate for this to affect some global state. For example, when changing channel in a set-top box application a transformer reset might be requested.

2.9.2 Providing supplementary buffers

In some cases, it is not possible for a data transformation to be supplied with all the data buffers required for the transformation to complete. The reasons for this are outlined in *Section 2.10.1* and *Section 2.10.2*. If a transformation cannot be supplied with all the data buffers initially then it is necessary to supply supplementary buffers to the transformer. This is achieved using one or more `MME_SEND_BUFFERS` commands.

`MME_SEND_BUFFERS` commands do not complete, that is, enter one of the final states, at the point the buffers are supplied to the transformer. They complete only when the buffers have been consumed by the transformer. This allows the application to know whether a buffer contains valid or in-use data without having to identify which transform request was responsible for filling it. Should such information be required, it is possible for transformer specific command status structures to be filled in by the transformer.

2.9.3 Altering global parameters

Global parameters form part of each instantiated transformer and are manipulated using the `MME_SET_GLOBAL_TRANSFORM_PARAMS` command code. Examples of global parameters include the chosen output format for data, gain for each channel of a mixer or the magnitude of reverb effects.

Global parameters can also be used to change less obvious items of global transformer state. For example, the transformer could be directed to move any command in the deferred state to the `MME_COMMAND_FAILED` state. By effectively abandoning any commands it has deferred it will then be possible for a transformer instance to be terminated.

When altering the global parameters, no data buffers are passed into or out of the transformer. All information required by the transformer should be passed using the

transformer specific command parameters. Similarly any information returned by the transformer should be contained in the transformer specific command status parameters.

2.10 Common types of transformer

This section distinguishes between types of transformer, based on how the input and output buffers are managed, in order to discuss the advantages and disadvantages of each approach.

A transformer is considered **frame-based** if its entire input and output buffers can be determined at the point the transform `MME_TRANSFORM` command is issued. Any transformer that does not require the use of `MME_SEND_BUFFERS` can therefore be considered to be frame-based.

A transformer is considered **streaming** if both its input and output buffers require the use of `MME_SEND_BUFFERS`.

Pure streaming transformers, while perfectly legitimate, are quite rare. Much more common are **hybrid** transformers consisting of frame-based input buffers and streaming output buffers or vice versa.

2.10.1 Frame-based operation

Frame-based operation is considered to be the default within MME. Although other modes of operation are possible, if ever there is a choice between frame-based or streaming operation then the frame-based approach is recommended.

By adopting frame-based operation, the amount of interrupt activity on both CPUs can be minimized. Particularly for media processors this maximizes processing bandwidth. Because all buffers are available before the transformation begins, there is no risk of the transformer blocking, which has the potential to seriously degrade performance.

Frame-based operation is a particularly good approach for decoding multiplexed audio/video streams common in embedded multimedia processing. For both audio and video decode, the size of the output frame or picture is known in advance of processing. It is also fairly easy to identify complete input frames, because the device performing the demultiplex can readily identify end-of-frame markers.

If a frame-based transformer is not supplied with sufficient buffers, it does not block, instead it moves to the `MME_COMMAND_FAILED` state and sets the appropriate error code in the command status structure.

When a frame-based transformer completes, it issues a `MME_COMMAND_COMPLETED_EVT` event and the application receives a callback.

2.10.2 Stream-based and hybrid operation

Unfortunately there are number of reasons why it may not be possible for a transformer to be wholly frame-based:

- transformations create an unknown quantity of output
- transformations consume an unknown quantity of input
- transformations are required to start before all available buffers are ready

The first situation is common in variable or average rate encoders. From a given input it is computationally unfeasible to estimate how much output will be created.

The second situation occurs when a stream format makes it difficult to identify end-of-frame markers or simply when a stream is not divided into frames.

The final situation is comparatively rare. One example is that of decoding a large JPEG image stored on 'slow' media, such as disk. It is desirable that the time between opening and displaying the image is minimized by starting to decode data as it becomes available. As such, once the first part of the JPEG is available, it is useful to initiate the transform, knowing that the remaining part of the image will be provided by the application when (or before) the input buffer underflows.

Unlike frame-based transformers, stream-based transformers (or the stream-based side of a hybrid transformer) will not return an error during data underflow or overflow. Instead a `MME_DATA_UNDERFLOW_EVT` or `MME_NOT_ENOUGH_MEMORY_EVT` event is issued and the command suspends its execution waiting for further data.

Note: *Both underflow and overflow are exceptional events and should not occur during normal operation of a streaming transformer. They exist only to allow these circumstances to be handled gracefully. When underflow or overflow occurs, all processing at the same priority is halted on that processor. This has the potential to waste significant processor bandwidth, particularly in single-purpose companion processors. Other transformer priorities are not effected. See* *Section 3.4.3: Streaming and hybrid transformers on page 40**.*

In order to prevent these problems, the application should normally buffer sufficient input data or output space for this situation to be avoided. Where buffering exposes latency problems when changing streams (for example, in trick modes or during channel change) then `MME_AbortCommand` *provides a means to mitigate this.*

When a data buffer has been consumed or filled completely, the `MME_SEND_BUFFER` command completes.

If an input data buffer is only partially consumed, the command will not complete; instead it remains pending until the next `MME_TRANSFORM` command consumes it.

Whether a partially filled output data buffer completes, before it is full, is transformer-dependant. If the output consists of variable length frames, it is normal to complete a partially-filled frame and move on to the next one. Where the output does not contain frame markers, the buffer will not complete until it is completely filled.

2.11 Linking and loading

This section describes linking and loading issues for different operating systems.

2.11.1 OS21

On OS21, the main application must link against both the ICS and MME libraries. Assuming the correct library search path is defined, this should be achieved by adding the following to the link command line:

```
-lmme -lics -lics_bsp
```

2.11.2 Linux

The MME API can be used on a host processor running Linux, in either kernel mode, user mode or both concurrently. There is a single kernel module that must be loaded and a single library that must be linked with a user application.

However, the MME kernel module must be loaded after the ICS kernel module is loaded.

Also, the user application must be linked against the MME user library. Assuming the correct library search path is defined, this should be achieved by adding the following to the link command line:

```
-lmme
```

It is not valid for a Linux kernel module to call `MME_Init` or `MME_Term` because initialization and de-initialization occur automatically when then module is loaded and unloaded.

2.11.3 Dynamic module linking

Multicom also supports dynamic modules which can be loaded and unloaded during application runtime. Dynamic modules are only supported on the CPUs that run OS21.

For example, in order to compile an ST231 compatible ELF module, a command similar to the following can be used;

```
st200cc -mcore=st231 -mruntime=os21\
       -nostdlib -fpic --rlib -fvisibility=protected\
       dyn.c -o dyn.out
```

Note: It is recommended that ST200 Micro Toolset R6.4.0 or later is used when building dynamic modules.

3 Writing an MME transformer

3.1 Overview

This chapter describes the process of interfacing a transformer to the MME API for use by applications. It is assumed that the reader is completely familiar with the MME API described in *Chapter 2: Using the MME API on page 14*.

The function to register transformers, `MME_RegisterTransformer`, is introduced in *Section 2.3.2: Registering transformers on page 20*. This function takes as arguments, five function pointers which are called by MME when a transformer-specific operation is requested by the application, see *Table 3*. All transformers are therefore required to provide five functions corresponding to the function pointers required by `MME_RegisterTransformer`. The specification of each of these functions is described in this chapter.

Table 3. Transformer function pointers

Function pointer type	Description
`MME_AbortCommand_t`	Transformer API function to abort a command.
`MME_GetTransformerCapability_t`	Transformer API function to return a transformer capability.
`MME_InitTransformer_t`	Transformer API function to initialize a transformer.
`MME_ProcessCommand_t`	Transformer API function to process a command.
`MME_TermTransformer_t`	Transformer API function to terminate a transformer.

In addition to providing the five functions, all but the most basic transformer will require parameters to control how it processes data. The parametric information is typically defined in a transformer specific header file included by both the application and the transformer code. Detailed information on passing parameters in a portable, endian-neutral manner is discussed in *Section 3.7 on page 42*.

The process for writing a transformer is identical whether you are targeting the host or a companion processor although obviously where the host and companion have different CPU architectures then optimization decisions (such as data buffer management) may have to be revisited. Transformers that utilize a hardware accelerator require only a small amount of extra complexity in the MME interface to manage asynchronous processing. This is described further in *Section 3.4.2: Deferred commands on page 38*.

3.2 Managing transformer lifetimes

Two of the function pointers required by `MME_RegisterTransformer` are concerned with managing the lifetime of a transformer. The transformer must implement the corresponding functions, one is responsible for initializing a transformer instance while the other is responsible for terminating it.

This initialization function pointer is of type `MME_InitTransformer_t`:

```
typedef MME_ERROR (*MME_InitTransformer_t) {
        MME_UINT initParamsLength,
        MME_GenericParams_t initParams,
        void **context)
```

The termination function pointer is of type `MME_TermTransformer_t`:

```
typedef MME_ERROR (*MME_TermTransformer_t) (void *context)
```

3.2.1 Instantiation

`MME_InitTransformer_t` is called as a result of an application call to `MME_InitTransformer`, see *Section 2.4: Managing transformer lifetimes on page 21*.

`MME_InitTransformer_t` is supplied with three parameters:

- a pointer to the transformer specific parameter block `initParamsLength`
- the length of this block
- context, in which it must store a pointer to its state information

If the transformer does not take any specific initialization parameters (or the application neglected to provide them) `initParams` is `NULL` and `initParamsLength` is zero.

If it is not possible to initialize a transformer instance, either because the supplied parameters are incorrect, or because there is no available hardware resource, then `MME_InitTransformer_t` can return an error code. Otherwise as a result of this function being called the transformer must:

- reserve any hardware resources required by the transformer when running
- allocate memory to contain state and parametric information of the transformer instance and return the address of this in the context arguments
- initialize any relevant state within the context structure
- copy any relevant parametric information from the parameter block into the context structure.

  The parameter information **must** be copied since the transformer can no longer address any part of the parameter block once this initialization function has completed.
- provide the MME framework with a pointer to context data specific to the transformer instance

3.2.2 Context data

The context data is key to managing multiple instances of a transformer, and must contain all state relevant to a transformer instance. Any temporary working values must be stored in the context data. A transformer that can be instantiated multiple times must avoid global variables. In fact, because the transformer may be instantiated at different priority levels (allowing one instance to pre-empt another), global variables should not be used even for temporary values.

It is possible for a single-instance transformer to statically allocate its context structure and supply a pointer to this global variable. The initialization function for single-instance transformers should return `MME_NOMEM` if the application attempts multiple instantiation.

Note: Any transformer that is not forced to operate as single-instance, through hardware dependency should be implemented as a multiple -instance transformer. In fact, extensive use of global variables should be avoided even for hardware transformers that are currently single instance, because this may limit their utility in future SoC devices that may contain multiple instances of that hardware.

3.2.3 Termination

`MME_TermTransformer_t` is called as a result of an application call to `MME_TermTransformer`, see *Section 2.4: Managing transformer lifetimes on page 21*. `MME_TermTransformer_t` takes the `context` parameter described in *Section 3.2* to specify the transformer instance that should be terminated.

`MME_TermTransformer_t` reverses all the steps performed during initialization. It releases any hardware resources it is using and frees any memory.

3.3 Querying the capabilities of a transformer

The transformer must implement a function that permits the application to query whether a transformer meets its requirements, see *Section 2.4.1: Querying the capabilities of a transformer on page 22*. This function must be compatible with the function pointer type, `MME_GetTransformerCapability_t`, which is an argument of `MME_RegisterTransformer()`:

```
typedef MME_ERROR (*MME_GetTransformerCapability_t) (
     MME_TransformerCapability_t *capability)
```

`MME_GetTransformerCapability_t` is called as a result of an application call to `MME_GetTransformerCapability`, see *Section 2.4.1 on page 22*.

Note: This function pointer is not provided with a context pointer because it is used to describe the capabilities of the abstract transformer rather than that of a specific transformer instance.

If the capability structure `capability`, or the transformer-specific parameter block it contains, is in some way incorrect then `MME_GetTransformerCapability_t` should return an error. Otherwise it should populate `MME_TransformerCapability_t` and, if applicable its transformer specific parameter block, `capability`.

The generic information a transformer must provide is its version number, which can be any 32-bit integer, and its preferred input and output data format in four character code (FOURCC) format - see *MME_DataFormat_t* on page 86 for more details.

For most transformers, the application knows the size of the parameter block in advance. Storage must be provided by the application. Such transformers should return an error if the parameter block is incorrectly sized.

In order to cope with transformer specific parameters of a variable size, the transformer must provide a means for the application to query how much memory it should provide for the parameters to be correctly stored.

There are many ways this could be achieved. The recommended approach is to define a fixed size parameter block that contains the actual size the parameter block is required to be, and use this to tell the application how much memory to allocate.

The following example shows part of the header file for a transformer that requires a capability structure of varied size:

```
enum STExampleInfoSize {
        MME_OFFSET_STExampleInfoSize_StructSize,
        MME_LENGTH_STExampleInfoSize

#define MME_TYPE_STExampleInfoSize_StructSize U32
};
typedef MME_GENERIC64 STExampleInfoSize_t[MME_LENGTH_STExampleInfoSize];

enum STExampleInfo {
        ...
}
/* can not typedef STExampleInfo since it is of variable size */
```

The above transformer would be queried from application code in the following way:

```
MME_ERROR err;
MME_TransformerCapability_t capability;
STExampleInfoSize_t query;
MME_GENERIC64 *info;

capability.StructSize = sizeof(MME_TransformerCapability_t);
capability.TransformerInfoSize = sizeof(STExampleInfoSize_t);
capability.TransformerInfo_p = &query;
err = MME_GetTransformerCapability("STExample", &capability);
/* check for errors */

capability.TransformerInfoSize = MME_PARAM(query, STExampleInfoSize_Structsize);
info = malloc(capability.TransformerInfoSize));
capability.TransformerInfo_p = info;
err = MME_GetTransformerCapability("STExample", &capability);
/* check for errors */
```

3.4 Processing a command

The transformer must implement a function that supports the processing any of the three types of commands introduces in *Section 2.8: Fault detection and recovery on page 28*. The function must be compatible with the function pointer type, `MME_ProcessCommand_t`, which is an argument of `MME_RegisterTransformer()`:

```
typedef MME_ERROR (*MME_ProcessCommand_t) (
     void *context,
     MME_Command_t *commandInfo)
```

`MME_ProcessCommand_t` is called as a result of an application call to `MME_SendCommand`, see *Section 2.7: Issuing commands on page 26*.

This function is supplied with the context pointer described in *Section 3.2: Managing transformer lifetimes on page 34*. It is also supplied with the command structure `commandInfo`, describing the actions requested of the transformer. The command structure consists of two parts: the incoming command request and the outgoing command status.

The command request portion is filled in by the application before calling `MME_SendCommand` and contains incoming parameters and any data buffers relevant to the command.

`MME_Command_t` contains a status structure `MME_CommandStatus_t` that is updated by MME before and after calling `MME_ProcessCommand_t`. During processing only the status structure parameter block and command identifier contain useful values. The parameter block is filled in by the transformer in order to pass back state information to the application, see *Section 2.6: Application and transformer specific data on page 26*. The command code is used to uniquely identify a particular command; in particular this identifier is used if ever a command must be aborted, see *Section 3.5: Aborting commands on page 41*.

Note: *Although the transformer specific parameter block held in the command's status structure should be filled in by the transformer, the status structure itself must be treated by the transformer as read-only. All fields are updated automatically by MME.*

If the command request is malformed in any way then this function should return an error code. The following list, though not exhaustive, provides a few ways in which a command can be badly formed:

- Wrong number of input or output buffers
- Incorrectly sized input or output buffers
- Badly formed or incorrectly sized parameter block attached to the command request
- Incorrectly sized parameter block attached to the command status `MME_CommandStatus_t`. (It is not possible for the outgoing parameter block to be badly formed because they are assumed to be uninitialized data when `MME_ProcessCommand_t` is called)

Note: *In systems where data buffers can be corrupted during transit, transformers are required to gracefully handle badly formed input buffers. It is possible to handle this by simply returning an error, but this often makes it difficult for the application to handle failures. For this reason it is usually preferable for a transformer to make the best possible attempt to decode the data and use the transformer specific status parameters to indicate to the application that the output may be incorrect.*

For correctly formed commands, the exact action required by the transformer depends upon the command code supplied by the application. For this reason, typical implementations of this command simply examine the command code and call a helper function. For example:

```
MME_ERROR EXMPL_ProcessCommand(void *ctx, MME_Command_t *cmd)
{
   switch (cmd->CmdCode) {
   case MME_TRANSFORM:
     return EXMPL_Transform(ctx, cmd);
   case MME_SEND_BUFFERS:
     return EXMPL_SendBuffers(ctx, cmd);
   case MME_SET_GLOBAL_TRANSFORM_PARAMS:
     return EXMPL_SetParameters(ctx, cmd);
   };

   return MME_INVALID_ARGUMENT;
}
```

Note: *Frame based transformers do not usually support the* `MME_SEND_BUFFERS` *command so that value is often omitted from the* `switch` *statement.*

The MME_TRANSFORM command instructs the transformer to perform a data transformation either on buffers supplied with the command or, for streaming transformers, on buffers sent using the MME_SEND_BUFFERS command. The MME_TRANSFORM command should not complete until at least a single frame of data has been processed.

Note: If the transformer has the concept of frames and follows the streaming model then the transformer must be provided with a parameter identifying how many bytes of data should be processed before the command completes.

The MME_SET_GLOBAL_TRANSFORM_PARAMS command is used to update parameters that affect all subsequent transforms. Such command typically have very short execution times since they need only alter a few parts of the context structure prior to returning.

The MME_SEND_BUFFERS command partners with the MME_TRANSFORM command to supply data buffers to a streaming transformer. This command is discussed in detail in *Section 3.4.3: Streaming and hybrid transformers on page 40*.

3.4.1 Communicating with the application

For simple transformers, all communication with the application is managed automatically. When a command completes its processing, and returns an error code by its processing function, MME automatically notifies the application that the command has finished processing.

When a transformer needs to initiate communication with the application, the following function is used:

```
MME_ERROR MME_NotifyHost(
     MME_Event_t event,
     MME_Command_t* commandInfo,
     MME_ERROR errorCode)
```

The circumstances when this function is required are identified by the event type:

- MME_COMMAND_COMPLETED_EVT, used to mark a deferred command (see *Section 3.4.2: Deferred commands on page 38*) as completed,
- MME_DATA_UNDERFLOW_EVT and MME_NOT_ENOUGH_MEMORY_EVT, used by streaming transformers (see *Section : Underflow and insufficient memory handling on page 40*) to indicate to the application that they have have exhausted either input or output buffers respecitvely.

commandInfo is the pointer originally supplied to the processing function while the error code is the value that the implementation will store in the error field of the command status prior to making any callbacks.

Note: It is not safe to call MME_NotifyHost *from an interrupt handler.*

3.4.2 Deferred commands

Deferred commands provide a means for a transformer to delegate some or all of its functionality to an asynchronous processing device such as a hardware accelerator.

A transformer indicates that it has deferred a command through a special error code, MME_TRANSFORM_DEFERRED. This return code indicates to MME that the command has not completed but that no further processing can be performed by the processor.

Note: Since the command has not completed no callback will take place on the host after the processing function returns MME_TRANSFORM_DEFERRED*. The host can be notified*

explicitly by the transformer code through calls to `MME_NotifyHost` *described in* *MME_NotifyHost on page 63*.

Before returning `MME_TRANSFORM_DEFERRED` the transformer must ensure the command will complete at some point in the future. This is achieved by carrying out the following actions.

- Remember the `MME_Command_t *` pointer passed into the processing function. This pointer is required when the time comes to notify the host processor that processing is complete.
- Set up an asynchronous event that will cause the command to complete at some point in the future. This is typically achieved by configuring an interrupt handler/task pair that will be signaled when the hardware accelerator has completed its work. A task will be required because it is not permitted to call any of the MME API functions from an interrupt handler.

If an `MME_SEND_BUFFERS` command returns `MME_SUCCESS`, it is deferred just as if it had returned `MME_TRANSFORM_DEFERRED`. This is because it is incorrect for a `MME_SEND_BUFFERS` command to complete successfully without asynchronous processing by a matching `MME_TRANSFORM` command. In this case there is no need for the transformer to configure an asynchronous event because command execution by the MME already provides this.

Pipelined transformers

A pipelined transformer is a special case of a deferred transformer. Pipelined transformers avoid having to setup an asynchronous handler, by checking the state of the hardware accelerator from a subsequent transform command. This is broadly analogous to the classic fetch-decode-execute pipeline common in microprocessor architectures.

The following example shows the management code for a simple two-stage pipelined transformer.

```
MME_ERROR Pipelined_Transform(void *ctx, MME_Command_t *cmd)
{
   MME_ERROR err1, err2;

   /* Do the first part of the transform in software */
   err1 = Pipelined_FirstHalfInSoftware(ctx, cmd);

   if (ctx->lastCmd) {
      /* There is a deferred command - wait for it to complete */
      /* err2 is the MME_ERROR code to set in the command's status */
      err2 = Pipelined_WaitForPreviousSecondHalf(ctx, ctx>lastCmd);
      MME_NotifyHost(MME_COMMAND_COMPLETED_EVT, ctx->lastCmd, err2);
   }

   if (MME_TRANSFORM_DEFERRED == err1) {
      /* The first part is in a deferred state - remember this */
      ctx->lastCmd = cmd;
      /* this function returns MME_TRANSFORM_DEFERRED if successful */
      err1 = Pipelined_SetupSecondHalf(ctx, cmd);
   } else {
      ctx->lastCmd = NULL;

   return err1;
}
```

Pipelined transformers block execution for the previous stages of the pipeline to complete. If the software side is running ahead of the hardware side then the transform function blocks

and at this point, execution of all commands on the current processor, of the same priority, halt. In the above example `Pipelined_WaitForPreviousSecondHalf` blocks, using an operating system primitive, until the hardware side has completed.

Poorly-tuned pipelined transformers can be harmful to processor bandwidth. Pipelined transformers are therefore best implemented only for single purpose companion processors.

Note: `Pipelined_WaitForPreviousSecondHalf` *must not busy wait as this will disrupt processing at lower priorities.*

3.4.3 Streaming and hybrid transformers

Streaming and hybrid transformers are required to support the `MME_SEND_BUFFERS` command since this is how their data buffers are delivered.

Like all other commands the `MME_SEND_BUFFERS` command should return an error code if the command structure is in some way invalid. It is also permissible to return an error if the transformer instance's buffer queue is full. If the processing function returns an error for a send buffers command the application will be immediately notified.

Otherwise, on receipt of an `MME_SEND_BUFFERS` command a streaming transformer must store the command within its context structure ready for it to be used by the corresponding `MME_TRANSFORM` command.

Note: *The* `MME_SEND_BUFFERS` *command interrupts the currently executing command in order to deliver the buffers. The implementation of the send buffers command should therefore perform the smallest amount of work possible in order to minimize the cost of this interruption. Examination of data buffers and chaining of scatter pages are best left to the* `MME_TRANSFORM` *command.*

After storing the `MME_SEND_BUFFERS` command in the context structure the processing function should return `MME_SUCCESS`. At this point the command will be deferred until it is marked as completed through execution of a `MME_TRANSFORM` command.

When the `MME_TRANSFORM` command, by calling `MME_NotifyHost`, marks a `MME_SEND_BUFFERS` command as completed, it guarantees that it will no longer access any part of that command, including its data buffers.

Underflow and insufficient memory handling

When a streaming transformer has insufficient input data to continue, it is required to emit `MME_DATA_UNDERFLOW_EVT` by using `MME_NotifyHost`. Similarly if there is insufficient output data, it is required to emit `MME_NOT_ENOUGH_MEMORY_EVT`.

In both cases, after emitting an event, the transformer must then use an operating system primitive such as a semaphore to suspend execution of the current thread.

Note: *The transformer must not busy wait as this will disrupt processing at lower priorities.*

When an `MME_SEND_BUFFERS` command is received that permits the processing of the command to continue, the transformer should use the operating system primitive to wake up the previously blocked thread.

3.5 Aborting commands

The transformer must implement a function that permits commands to be aborted. This function must be compatible with the function pointer type, `MME_AbortCommand_t`, which is an argument of `MME_RegisterTransformer()`:

```
typedef MME_ERROR (*MME_AbortCommand_t) (
      void *context,
      MME_CommandId_t commandId)
```

`MME_AbortCommand_t` is called as a result of an application call to `MME_AbortCommand`, see *Section 2.7.1: Aborting commands on page 28*.

A call to this function is not a demand to abort the command but a request that the transformer may, in certain circumstances, choose to ignore. If the transformer is not able to abort the command it should return `MME_INVALID_ARGUMENT`.

MME never attempts to terminate a transformer with outstanding commands. Any command that is unable to complete without some further action being performed on the transformer must support aborts. Some examples of commands that are required to support abortion include:

- all `MME_SEND_BUFFERS` commands
- commands blocked after data underflow or overflow
- pipelined commands

Commands are marked as aborted by one of the following methods:

- by returning `MME_SUCCESS` from `MME_AbortCommand_t`
- by calling `MME_NotifyHost` with the event code `MME_COMMAND_EVT` and the error code `MME_COMMAND_ABORTED`
- by returning `MME_COMMAND_ABORTED` from `MME_Process_Command_t`.

The call to `MME_NotifyHost` can be made from any thread, including the abort function, the processing function or from asynchronous threads owned by a deferred transformer.

Note: When aborting a command using any of the above methods, the transformer guarantees that the following conditions are met.

- *No further execution time is spent on the command.*
- *No further use is made of any part of the* `MME_Command_t` *structure, including data buffers. This means that no further calls to* `MME_NotifyHost` *are made.*
- *No further attempt is made to mark the command aborted. Thus if* `MME_AbortCommand_t` *calls* `MME_NotifyHost` *or expects the currently executing command to read from the command structure or to return any value except* `MME_TRANSFORM_DEFERRED` *then the abort command must itself return* `MME_TRANSFORM_DEFERRED`.

Thus `MME_AbortCommand_t` should return `MME_SUCCESS` only if the command has already been aborted and the host has not been notified by another means.

3.6 Scheduling and re-entrancy

MME utilizes multiple threads and it is important that transformer functions are written in such a manner that thread safety is maintained.

Neither initialization nor termination have any thread safety issues. The functions are not re-entered, nor will any other transformer interface function be called during these operations.

Note: It is a pre-condition of the termination function that all outstanding commands complete, and this is assured by MME.

Similarly calls to `MME_GetTransformerCapability_t` (see *Section 3.3 on page 35*) are serialized.

For a single instantiated transformer, up to three threads can operate over the same context structure at the same time. These are:

- An execution thread that calls the processing function with a command code of either `MME_TRANSFORM` or `MME_SET_GLOBAL_TRANSFORM_PARAMS`.
- A manager thread that calls the processing function with a command code of `MME_SEND_BUFFERS`.
- A manager thread that calls the abort function.

In summary the `MME_ProcessCommand_t` can be re-entered but not with the same command code, while `MME_AbortCommand_t` cannot be re-entered. It is a requirement however, for transformers to protect access to any variable or list that is manipulated by multiple threads.

Note: Transformers that do not support `MME_SEND_BUFFERS` or `MME_AbortCommand_t` are implicitly thread-safe.

3.7 Parameter passing

Many of the MME functions take transformer specific parameter blocks specified in MME structures as `MME_GenericParams_t`, see *Section 2.6: Application and transformer specific data on page 26*. In each case the parameter block is described using a pointer to a generic 64-bit type and a size in bytes. Transformers that require parameters to correctly process their input typically specify parameters in a header file shared by the application.

Note: It is not possible for a parameter block to contain pointers to other data because it may not be possible to dereference these pointers on other processors.

On systems with identical endianness, the parameter block is presented byte for byte identically as it passes between the application and the transformer. However on mixed endian systems the parameter block will be treated as an array of 64-bit quantities each of which will be byte swapped in 64-bit units. For example, the 64-bit hexadecimal integer 0x0011 2233 4455 6677 would after swapping become 0x7766 5544 3322 1100 if it were examined on the originating processor.

It is the transformer's responsibility to define its parameter block in such a way that it may be safely passed between processors of mixed endianness. The way MME handles mixed endian machines implies that the parameter block should be implemented as an array of 64-bit entities.

MME provides a number of macros that can be used to assist the transformer and application author to access data contained in such an array in a convenient but endian neutral manner. These macros use a combination of constants and C preprocessor string

concatenation in order to provide access, both named and typed, to elements of the parameter array.

Note: Existing transformers and their applications may continue to pass parameters as a sequence of bytes instead of an array of 64-bit entities. This is typically achieved by mapping a C structure as a parameter block. This approach is not portable since it relies upon matching compiler behavior and endianness on all processors. It is not recommended that new transformers pass their parameters in this manner.

3.7.1 Data representation

The macros provided by MME to store data into an array of 64-bit entities, use a specific data representation. This representation allows 8, 16, 32 and 64-bit two's complement integers to be directly written by the CPU without any manipulation. Similarly, IEEE floating point numbers are typically stored using their normal bit pattern.

Table 4 shows the same parameter array elements represented in both big endian and little endian formats. The MME implementation will automatically convert between these forms when a parameter block is copied between processors of differing endianness

Table 4. Data representation - endianness

Size	Little endian	Big endian
8 bit	0 _ _ _ _ _ _ _	_ _ _ _ _ _ _ 0
16 bit	0 1 _ _ _ _ _ _	_ _ _ _ _ _ 1 0
32 bit	0 1 2 3 _ _ _ _	_ _ _ _ 3 2 1 0
64 bit	0 1 2 3 4 5 6 7	7 6 5 4 3 2 1 0

The macros are aware of the size of the object they are storing allowing the base address of the big endian values to be automatically calculated (at compile time).

3.7.2 Mapping application data structures into MME parameters

Three forms of parameter are managed by MME to support mapping of application data structures to MME parameters. These are **individual**, **array** and **parameter array**. They are used to pass individual data elements, arrays of elements and structures within structures respectively.

The following structure is used as an example:

```
struct MyParams {
        ...
        unsigned char FooBar;
        ...
        UINT32        TeePipes[10];
        ...
        struct MySubList {
                char a;
                ...
        };
};
```

The field `FooBar` is passed as an individual parameter, the array `TeePipes` as an array parameter and the structure `MySubList` as a parameter array.

The descriptions in *Section* through to *Section* use enumerated constants to specify the offset of each entry within the MME array of 64-bit parameters. The name of an entry in an enumeration is prefixed with `MME_OFFSET_`, for example `MME_OFFSET_FooBar`.

The type of an entry is defined with a `#define` directive and prefixed with `MME_TYPE_`, for example `MME_TYPE_FooBar`.

Macros are used to access the value of an entry in the MME array.

Individual parameters

An individual parameter is a single typed element of the MME parameter array. It is defined by an offset into the parameter array together with the type information for that parameter.

To define a parameter, `FooBar`, a constant, `MME_OFFSET_FooBar` is required to describe the offset into the parameter array at which `FooBar` will be found. Similarly a macro `MME_TYPE_FooBar` is required to describe the type of the parameter.

`MME_OFFSET_FooBar` can be a preprocessor macro though normally, because is it merely an integer, it is an enumerated constant.

`MME_TYPE_FooBar` must be a preprocessor macro because it contains a sequence of C tokens.

For example the following would define a unsigned character parameter, `FooBar`, at offset one.

```
enum {
   MME_OFFSET_SomeValueAtOffsetZero,
   MME_OFFSET_FooBar,
   ...

#define MME_TYPE_FooBar unsigned char
};

/* usage example - assign variable 'c' the value of the */
/* parameter FooBar */
unsigned char c = MME_PARAM(list, FooBar);
```

Array parameters

An array parameter is defined in the same way as a individual parameter but is immediately followed by unused locations within the array. This allows the array parameter and an index to be used to extract numbered elements.

Shown below is a ten element array parameter, `TeePipes`, followed by an normal parameter, `Flange`.

Note: *The array nature of* `TeePipes` *is reflected in the offset of the subsequent parameter,* `Flange`*.*

```
enum {
   MME_OFFSET_TeePipes,
   MME_OFFSET_Flange = MME_OFFSET_TeePipes + 10,
   ...

#define MME_TYPE_TeePipes UINT32
...
};
/* usage example - assign variable 'x' the value of the */
/* fifth element of the parameter TeePipes */
uint32 x = MME_INDEXED_PARAM(list, TeePipes, 5);
```

Parameter arrays as parameters

It is quite possible for a parameter array to wholly contain another parameter array to facilitate the mapping of structures within structures into MME parameters. In this case the parameter is defined only by its offset since its type is known to be `MME_GenericParam_t`. Like array parameters the length of the parameter array is defined by the offset of the subsequent element.

For example:

```
enum {
   MME_OFFSET_MySublist,
   MME_OFFSET_NextParameter = MME_OFFSET_MySubList +
MME_LENGTH(SubList),
   ...
};

/* usage example - assign variable 's' to the pointer */
/* MySubList, an element of the parameter list */
MME_GENERIC64 *s = MME_PARAM_SUBLIST(list, MySubList);
```

Recording the length of a parameter array

Once all the offsets for a particular parameter array have been defined, the length of the array must be defined in a standard form so that it can be returned by the `MME_LENGTH` macro. This symbol is derived from the name of the parameter array.

For example:

```
enum SimpleIdx {
   MME_OFFSET_AnInteger,

   MME_LENGTH_Simple

#define MME_TYPE_AnInteger int
};

/* usage example */
l = MME_LENGTH(Simple);
```

3.7.3 Namespace management

The names of elements of all parameter arrays and their sub-lists occupy a single shared namespace. For this reason care must be taken to choose parameter names such that independent transformers do not interfere with each other. This chapter provides guidelines on the selection of appropriate names.

Naming parameter arrays

All parameter arrays must have a unique name. To ensure this, it is recommended, to divide the name of the parameter into the following three components:

1. A company or division name, for example 'ST'. This divides the namespace and radically reduces the chance of namespace collision.
2. Purpose or role of the transformer (such as Ac3Decoder or Mixer).
3. The operation to which this parameter list is targeted. *Table 5: Recommended postfixes for parameter array names* contains guidance for standard MME operations.

Table 5. Recommended postfixes for parameter array names

Operation	Postfix
`MME_GetTransformerCapability()`	`Info`
`MME_InitTransformer()`	**`Init`**, or **`Global`** if the initialization parameters are not distinict.
`MME_SendCommand()` [`MME_SET_GLOBAL_TRANSFORM_PARAMS` and reply]	`Global/GlobalStatus`
`MME_SendCommand()` [`MME_TRANSFORM` and reply]	`Transform/TransformStatus`
`MME_SendCommand()` [`MME_SEND_BUFFERS` and reply]	`Send/SendStatus`

3.7.4 An example

This example maps the following C structure into MME parameters:

```
struct STExampleTransform {
  U32 STExampleTransformNormal;
  U8  STExampleTransformArray[10];

  struct STExampleSub {
    U32 STExampleSubNormal;
  };
};
```

To copy the data listed above as an MME parameter array the transformer would add the following definitions to its header file:

```
/* As a parameter sub list this list does not use the standard */
/* postfixes from the table above */

enum STExampleSub {
   MME_OFFSET_STExampleSubNormal,

   MME_LENGTH_STExampleSub
```

```
#define MME_TYPE_STExampleSubNormal U32
};

typedef MME_GenericParams_t
      MME_STExampleSub_t[MME_LENGTH(STExampleSub)];

enum STExampleTransform {
   MME_OFFSET_STExampleTransformNormal,
   MME_OFFSET_STExampleTransformArray,
   MME_OFFSET_STExampleTransformSublist =
      MME_OFFSET_STExampleTransformArray + 10,

   MME_LENGTH_STExampleTransform =
      MME_OFFSET_STExampleTransformSublist + MME_LENGTH(STExampleSub)

#define MME_TYPE_STExampleTransformNormal U32
#define MME_TYPE_STExampleTransformArray U8
/* no need for MME_TYPE_STExampleTransformSublist */
};
typedef MME_GENERIC64
      MME_STExampleTransform_t[MME_LENGTH(STExampleTransform)];
```

The following example demonstrates how an application would use the parameter array above. It is assumed that the transformers header file will be included by the application.

```
/* Send some parameters with a command /*

  MME_Command_t command = { sizeof(MME_Command_t),
                            MME_SET_GLOBALTRANSFORM_PARAMS,
                            MME_COMMAND_END_RETURN_NOTIFY,
                            ... };
  MME_STExampleTransform_t transformParams;
  MME_STExampleTransform_t transformSubParams;

  /* Set the individual element to 45 */
  MME_PARAM(transformParams, STExampleTransformNormal) = 45;

  /* Set the array element at index 2 to 70 */
  MME_INDEX_PARAM(transformParams, STExampleTransformArray, 2) = 70;

  /* Set the sub-structure element to 8 */
  MME_PARAM(transformSubParams, STExampleSubNormal) = 8;

  /* Setup the substructure within the parameter structure */
  MME_PARAM_SUBLIST(transformParams, STExampleTransformSublist, transformSubParams);

  /* Specify the parameters with the MME_Command_t structure */
  command.Params_p = &transformParams;
  command.ParamSize = MME_LENGTH_BYTES(STExampleTransform);

  MME_SendCommand(handle, &command);
```

4 MME API

This chapter describes the MME API in terms of its functions, constants, enums and types.

4.1 Function definitions

This section provides detailed descriptions of the MME functions. The functions are listed in alphabetical order.

MME_AbortCommand — Abort command submitted to transformer

Definition:

```
#include <mme.h>

MME_ERROR MME_AbortCommand(
              MME_TransformerHandle_t  Handle,
              MME_CommandId_t          CmdId)
```

Arguments:

`Handle`	Handle of the targeted transformer.
`CmdId`	Command identity of the command to abort.

Returns:

`MME_SUCCESS`	An abort request has been submitted - this does not imply the command has been aborted.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_INVALID_HANDLE`	The transformer handle is invalid.
`MME_INVALID_ARGUMENT`	The `CmdId` is invalid.

Description: Attempt to abort a command that has been submitted to a transformer.

The behavior of this function is transformer and implementation specific. Commands can always be aborted when in the `MME_COMMAND_PENDING` state, that is **before** being processed. However, depending on their implementation some transformers may also accept to abort command during their processing (`MME_COMMAND_EXECUTING` state). When a command has been aborted, the `Error` field of the `MME_CommandStatus_t` is set to `MME_COMMAND_ABORTED`. The callback function on the host is called when the command is successfully aborted.

Comments: Call type: non-blocking function call (the operation completes when the callback function has been called).

MME_AllocDataBuffer — Allocate MME data buffer

Definition:

```
#include <mme.h>

MME_ERROR MME_AllocDataBuffer(
              MME_TransformerHandle_t Handle,
              MME_UINT            Size,
              MME_AllocationFlags_t   Flags,
              MME_DataBuffer_t      **DataBuffer_p)
```

Arguments:

`Handle`	Handle of the targeted transformer.
`Size`	Number of bytes to allocate.
`Flags`	Specify special requirements of the memory allocated, see *MME_AllocationFlags_t on page 76*.
`DataBuffer_p`	Pointer to a pointer to an allocated data buffer structure to be populated.

Returns:

`MME_SUCCESS`	The operation completed correctly.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_NOMEM`	The memory required to complete this command is not available.
`MME_INVALID_HANDLE`	The transformer handle is invalid.
`MME_INVALID_ARGUMENT`	`Flags` or `DataBuffer_p` is invalid.

Description:

Allocate a new MME data buffer.

This command allocates memory that can be optimally communicated to the transformer indicated by `Handle`. `Flags` can be used to specify additional useful properties required of the allocated memory. For example, its cache ability or whether it is required to be contiguous.

Comments:

Call type: blocking function call.

See also:

MME_FreeDataBuffer

MME_DebugFlags — Set MME debug logging flags

Definition:

```
#include <mme.h>

MME_ERROR MME_DebugFlags (MME_DBG_FLAGS Flags)
```

Arguments:

`Flags`	Bitmask of debugging flags. See *MME_DBG_FLAGS on page 87*.

Returns:

`MME_SUCCESS`	The debug flags were successfully set.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_ICS_ERROR`	An ICS subsystem error occurred.

Description: Sets the MME debugging flags to the supplied bitmask value.

MME_DeregisterMemory — Deregister memory region

Definition:

```
#include <mme.h>

MME_ERROR MME_DeregisterMemory (MME_MemoryHandle_t Handle)
```

Arguments:

`Handle`	Handle of the memory region registered with `MME_RegisterMemory`.

Returns:

`MME_SUCCESS`	The memory region was successfully deregistered.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_INVALID_HANDLE`	The handle does not refer to an existing memory region registration.
`MME_ICS_ERROR`	An ICS subsystem error occurred.

Description: Deregister a memory region that was previously registered with `MME_RegisterMemory`. specific transformer.

Comments: Call type: blocking function call.

See also: *MME_RegisterMemory*

MME_DeregisterTransformer — Deregister transformer

Definition:

```
#include <mme.h>

MME_ERROR MME_DeregisterTransformer(const char* name)
```

Arguments:

`name`	The name of a transformer that has been registered with `MME_RegisterTransformer()`.

Returns:

`MME_SUCCESS`	The transformer has been successfully deregistered.
`MME_INVALID_ARGUMENT`	The transformer name is not registered.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.

Description: Deregister a transformer, that was previously registered on the CPU, from which the call is made.

Comments: Call type: blocking function call.

See also: *MME_RegisterTransformer*

Section 2.3.3: Example on page 21

MME_ErrorStr — Returns an MME error string

Definition:

```
#include <mme.h>

const char * MME_ErrorStr (MME_ERROR err)
```

Arguments:

`err`	An MME_ERROR error code.

Returns: A pointer to the corresponding MME error string.

Errors: None

Context: Callable from task and interrupt context. Can be called before `MME_Init()`.

Description: `MME_ErrorStr()` returns a pointer to the corresponding MME error string based on the supplied `err` code. This function is a useful way to display/log a text string describing the MME error code when an error occurs.

`err` should be a valid MME error code as returned by an MME API function.

MME_FreeDataBuffer — Release MME data buffer

Definition:

```
#include <mme.h>

MME_ERROR MME_FreeDataBuffer(
                MME_DataBuffer_t *DataBuffer_p)
```

Arguments:

`DataBuffer_p`	Pointer to an allocated data buffer structure to be freed.

Returns:

`MME_SUCCESS`	The operation completed correctly.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_INVALID_ARGUMENT`	`DataBuffer_p` is invalid.

Description: Release memory previously allocated with `MME_AllocDataBuffer()`.

This command releases memory previously allocated with `MME_AllocDataBuffer()`. The behavior is undefined if the memory was not previously allocated by the MME API or if pointers within the structure have been modified.

Comments: Call type: blocking function call.

See also: *MME_AllocDataBuffer*

MME_GetTransformerCapability Return details of transformer capability

Definition:

```
#include <mme.h>

MME_ERROR MME_GetTransformerCapability(
              const char   *TransformerName,
              MME_TransformerCapability_t
*TransformerCapability_p)
```

Arguments:

`TransformerName`	Name of the transformer whose capability is to be queried.
`TransformerCapability_p`	Pointer to an allocated `MME_TransformerCapability_t` structure that will be filled with the capability of the corresponding transformer.

Returns:

`MME_SUCCESS`	The operation completed correctly.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_UNKNOWN_TRANSFORMER`	No transformer of the specified name exists.
`MME_INVALID_ARGUMENT`	`TransformerCapability_p` is invalid.

Description: Return capability and requirement for a given transformer type.

The following fields of the *`MME_TransformerCapability_t`* structure must be initialized prior to calling this function: `StructSize`, `TransformerInfoSize` and `TransformerInfo_p`. All subsequent fields will be filled in by MME as a result of the call.

Comments: Call type: blocking function call.

See also: *MME_TransformerCapability_t*

MME_INDEXED_PARAM — Extract indexed parameter

Definition:

```
#include <mme.h>

#define MME_INDEXED_PARAM(params, name, index)
```

Arguments:

`params`	Pointer to a parameter array of type `MME_GenericParams_t`.
`name`	Name of the parameter to be extracted from the array.
`index`	Index of the parameter to extract.

Returns: An `lvalue` (an object that can be assigned to) whose type is selected by the named parameter.

Description: Extract a indexed named parameter from a parameter array. This macro uses other special purpose macros or named constants to process the name.

For example, the following macros define an indexed parameter name called `ThisIsIndexed`.

```
enum {
...
MME_OFFSET_ThisIsIndexed = 2,
...
#define MME_TYPE_ThisIsIndexed U32
}
```

This parameter can be extracted as follows:

```
MME_INDEXED_PARAM(params, ThisIsIndexed, 0) = 0xAC3;
MME_INDEXED_PARAM(params, ThisIsIndexed, 1) = 0xDDD;
```

See also: *MME_PARAM*

MME_LENGTH

MME_PARAM_SUBLIST

MME_Init — Initialize the MME infrastructure

Definition:

```
#include <mme.h>

MME_ERROR MME_Init(void)
```

Arguments: None

Returns:

`MME_SUCCESS`	The operation completed correctly.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver could not be initialized because underlying resources were not initialized yet.
`MME_DRIVER_ALREADY_INITIALIZED`	`MME_Init()` has been called already.
`MME_NOMEM`	The memory required to complete this command is not available.

Description: Initialize the MME infrastructure. This function must be called prior to calling any other MME functions. It must be called at least once on each processor and by each Linux user mode process. Once initialized further calls return `MME_DRIVER_ALREADY_INITIALIZED`.

See also: *MME_InitTransformer*

MME_Term

MME_InitTransformer — Initialize an instance of a transformer

Definition:

```
#include <mme.h>

MME_ERROR MME_InitTransformer(
                const char                    *Name,
                MME_TransformerInitParams_t   *Params_p,
                MME_TransformerHandle_t       *Handle_p)
```

Arguments:

`Name`	Name of the transformer registered with `MME_RegisterTransformer`.
`Params_p`	Pointer to an allocated `MME_TransformerInitParams_t` that contains the parameters with which the transformer will be initialized.
`Handle_p`	Pointer to an `MME_TransformerHandle_t` that will contain the handle of the initialized transformer.

Returns:

`MME_SUCCESS`	The device has been successfully initialized.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_NOMEM`	The memory required to complete this command is not available.
`MME_UNKNOWN_TRANSFORMER`	No transformer of the specified name exists.
`MME_INVALID_ARGUMENT`	`Params_p` or `Handle_p` is invalid.

Description:

Creates and initializes an instance of a specific transformer on a CPU.

The name argument must not be longer than `MME_MAX_TRANSFORMER_NAME` bytes. The error code `MME_INVALID_ARGUMENT` will be returned if the name is too long.

The name of the transformer implicitly describes the type of that transformer, for example "`STAC3DecoderMacro`". This can be confirmed by using `MME_GetTransformerCapability()` to examine the capability of transformer if required.

If the host has to synchronize with transformer registration on a companion, this function should be called iteratively until `MME_SUCCESS` is returned.

`MME_Init` must be called prior to the `MME_InitTransformer` function.

Comments:

Call type: blocking function call.

See also:

MME_Init

MME_TermTransformer

MME_LENGTH — Return the length of a parameter array

Definition:

```
#include <mme.h>

#define MME_LENGTH(name)
```

Arguments:

`name`	Name of the parameter array.

Returns: The length of the named parameter array.

Description: Find the length of a named parameter array.

This macro uses other special purpose macros to process the name.

For example, the following macros define a parameter name called `ThisIsAGlobalName`.

```
#define MME_PARAMS_LENGTH_ThreeParameters 3
```

The macro should be used as follows:

```
MME_GenericParams_t params[MME_LENGTH(ThreeParameters)];


MME_PARAMS(params, ParamOne) = 1;
MME_PARAMS(params, ParamTwo) = 2;
MME_PARAMS(params, ParamThree) = 3;
```

See also: *MME_INDEXED_PARAM*

MME_PARAM

MME_PARAM_SUBLIST

MME_LENGTH_BYTES

MME_LENGTH_BYTES Return the length of a parameter array in bytes

Definition:

```
#include <mme.h>

#define MME_LENGTH_BYTES(name)
```

Arguments:

`name`	Name of the parameter array.

Description: Find the length of a named parameter array in bytes.

Returns: The length of the named parameter array in bytes.

See also: *MME_INDEXED_PARAM*

MME_LENGTH

MME_PARAM

MME_PARAM_SUBLIST

MME_ModifyTuneable — Modify system-wide configuration values

Definition:

```
#include <mme.h>

MME_ERROR MME_ModifyTuneable(MME_tunable_t key,
                             MME_UNIT value)
```

Arguments:

`key`	Specify the tuneable value to be modified.
`value`	The new value for the tuneable.

Returns:

`MME_INVALID_ARGUMENT`	The supplied key is invalid or not support on this operating system.
`MME_SUCCESS`	The tuneable has been successful updated.

Description: Modify tuneable system-wide configuration values. MME uses sensible default values for parameters such as thread priority, however, some users need to tune such values to optimize thread interactions. Many values have been made tuneable to allow users to modify their systems without recompiling MME. This call alters a single tuneable parameter selected by key, which can be one of the values in *Table 6*:

Table 6.Tuneable values for MME parameters

Value(1)	Description
`MME_TUNEABLE_BUFFER_POOL_SIZE`	Modify the default size of the MME data buffer pool as used by `MME_AllocDataBuffer`.
`MME_TUNEABLE_MANAGER_THREAD_PRIORITY`	Tune the priority of the MME manager thread. This thread is responsible for administrative operations such as responding to: `MME_GetTransformerCapability` and `MME_InitTransformer`. This should have a high priority to prevent background batch processes (such as audio encode) from interfering with transformer creation.
`MME_TUNEABLE_TRANSFORMER_TIMEOUT`	Timeout period in microseconds before a transformer administration operation returns an error.
`MME_TUNEABLE_TRANSFORMER_THREAD_PRIORITY`	Tune the priority of the MME transformer thread. This thread is responsible for receiving `MME_SEND_BUFFERS` commands, together with requests to abort the current function or terminate the transformer. This should have a priority greater than or equal to the highest execution loop priority.

Table 6.Tuneable values for MME parameters (continued)

Value[1]	Description
MME_TUNEABLE_EXECUTION_LOOP_HIGHEST_PRIORITY	Tune the priority of each of the execution loop threads. The execution loops are responsible for performing transform requests and altering global parameters. No execution loop should have a higher priority than the manager or transformer threads. The priorities of the execution loops should be such that MME_TUNEABLE_EXECUTION_LOOP_HIGHEST_PRIORITY has the highest priority and MME_TUNEABLE_EXECUTION_LOOP_LOWEST_PRIORITY has the lowest.
MME_TUNEABLE_EXECUTION_LOOP_ABOVE_NORNAL_PRIORITY	
MME_TUNEABLE_EXECUTION_LOOP_NORNAL_PRIORITY	
MME_TUNEABLE_EXECUTION_LOOP_BELOW_NORNAL_PRIORITY	
MME_TUNEABLE_EXECUTION_LOOP_LOWEST_PRIORITY	

1. Value interpreted as an OS priority level)

Unlike most MME calls it is possible to modify tuneables before the MME API has been initialized.

Note: Most tuneables do not affect behavior once MME has been initialized and hence they should modified before calling `MME_Init`.

MME_NotifyHost — Notify host that transformer has generated an event

Definition:

```
#include <mme.h>

MME_ERROR MME_NotifyHost(MME_Event_t event,
                         MME_Command_t *commandInfo,
                         MME_ERROR      errorCode)
```

Arguments:

`event`	The event that should be passed to the host callback. See `MME_NotifyHost` description.
`commandInfo`	The `MME_Command_t*` passed into the transformer function `MME_ProcessCommand_t`.
`errorCode`	The error state of the command.

Returns:

`MME_SUCCESS`	The host has been notified.
`MME_INVALID_ARGUMENT`	An argument is invalid.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.

Description:

Informs the host that the transformer has generated an event.

This function must *not* be called from an interrupt handler.

The `event` argument can be one of the events listed in *MME_Event_t* on page 90.

This call will cause the application-supplied callback function on the host to be called with the its event parameter set to the value of `event`.

Comments:

Call type: non-blocking function call.

MME_PARAM — Extract named parameter

Definition:

```
#include <mme.h>

#define MME_PARAM(params, name)
```

Arguments:

`params`	Pointer to a parameter array of type `MME_GenericParams_t`.
`name`	Name of the parameter to be extracted from the array.

Returns: An `lvalue` (an object that can be assigned to) whose type is selected by the named parameter.

Description: Extract a named parameter from a parameter array.

This macros uses other special purpose macros or named constants to process the name.

For example, the following macros define a parameter name called `ThisIsAGlobalName`.

```
enum {
...
MME_OFFSET_ThisIsAGlobalName = 2,
...
#define MME_TYPE_ThisIsAGlobalName U32
}
```

This parameter can be extracted as follows:

```
MME_PARAM(params, ThisIsAGlobalName) = 0xAC3;
printf("%d\n", MME_PARAM(params, ThisIsAGlobalName));
```

See also: *MME_INDEXED_PARAM*

MME_PARAM_SUBLIST

MME_LENGTH

MME_PARAM_SUBLIST — Extract named parameter sub-array

Definition:

```
#include <mme.h>

#define MME_PARAM_SUBLIST(params, name)
```

Arguments:

`params`	Pointer to a parameter array of type `MME_GenericParams_t`.
`name`	Name of the parameter to be extracted from the array.

Returns: A pointer to a sub-list of parameters (has type `MME_GenericParams_t`).

Description: Extract a named parameter sub-array from a parameter array.

This macro uses other special purpose macros or named constants to process the name.

For example, the following macros define a parameter name called `ThisIsASublist`.

```
enum {
...
MME_OFFSET_ThisIsASublist = 2,
...
}
```

This parameter can be extracted as follows:

```
sublist = MME_PARAM_SUBLIST(params, ThisIsASublist);
MME_PARAM(sublist, SomeParameter) = 0xAC3;
```

Returns: A pointer to a sub-list of parameters (has type `MME_GenericParams_t`).

See also: *MME_PARAM*

MME_INDEXED_PARAM

MME_LENGTH

MME_PingTransformer — Check that a transformer is still responding

Definition:

```
#include <mme.h>

MME_ERROR MME_PingTransformer(
              MME_TransformerHandle_t Handle,
              MME_Time_t Timeout)
```

Arguments:

`Handle`	Handle of the targeted transformer.
`Timeout`	Timeout period in milliseconds.

Returns:

`MME_SUCCESS`	The ping request was successfully responded to by the target transformer.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_NOMEM`	The memory required to complete this command is not available.
`MME_INVALID_HANDLE`	The handle does not refer to an existing transformer
`MME_COMMAND_TIMEOUT`	The ping request was not responded to by the remote transformer within the specified timeout period.

Description: Send a 'ping' command request to a specific transformer. This acts like a very high priority command to the target transformer and hence, should be processed as the very next command the transformer executes. If the command has not responded within the specified timeout then `MME_COMMAND_TIMEOUT` is returned. The `Timeout` value should be carefully chosen so it does not produce false failures due to the target transformer or CPU simply taking too long to respond.

Comments: Call type: blocking function call.

See also: *MME_SendCommand*

MME_WaitCommand

MME_Time_t

MME_RegisterMemory Register memory region with a specific transformer

Definition:

```
#include <mme.h>

MME_ERROR MME_RegisterMemory (MME_TransformerHandle_t Handle,
                              void *Base,
                              MME_SIZE Size,
                              MME_MemoryHandle_t *Handle_p)
```

Arguments:

`Handle`	Handle of the transformer to associate with this memory registration.
`Base`	Virtual address of the base of the memory region being registered.
`Size`	Size in bytes of the memory region being registered.
`Handle_p`	Pointer to an `MME_MemoryHandle_t` that will contain the handle of the registered memory region.

Returns:

`MME_SUCCESS`	The memory region was successfully registered.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_INVALID_HANDLE`	The handle does not refer to an existing transformer
`MME_INVALID_ARGUMENT`	One or more of the supplied arguments are invalid.
`MME_ICS_ERROR`	An ICS subsystem error occurred.

Description: Register a memory region with a specific transformer. This must be done for all[d] memory regions that are used as data buffers to `MME_SendCommand`.

If cached and uncached translation of the physical memory region are to be used, then the region must be registered with both its cached and uncached virtual addresses.

`Base` and `Size` are automatically aligned and rounded to the associated system page size.

Comments: Call type: blocking function call.

See also: *MME_DeregisterMemory*

d. This does not apply to data buffers allocated using MME_AllocDataBuffer.

MME_RegisterTransformer — Register a transformer for instantiation on a CPU

Definition:

```
#include <mme.h>

MME_ERROR MME_RegisterTransformer(
                const char                      *name,
                MME_AbortCommand_t               abortFunc,
                MME_GetTransformerCapability_t
getTransformerCapabilityFunc,
                MME_InitTransformer_t
initTransformerFunc,
                MME_ProcessCommand_t
processCommandFunc,
                MME_TermTransformer_t
termTransformerFunc)
```

Arguments:

`name`	A unique name for the transformer.
`abort`	The transformer function to call when an abort request is made.
`getTransformerCapabilityFunc`	The transformer function to call when a capability request is made.
`initTransformerFunc`	The transformer function to call when a transformer is initialized.
`processCommandFunc`	The function to call when a command is sent to the transformer.
`termTransformerFunc`	The function to call when a transformer instance is terminated.

Returns:

`MME_SUCCESS`	The device has been successfully initialized.

Description: Registers a transformer for later instantiation on the CPU from which this call is made.

The name argument must not be longer than `MME_MAX_TRANSFORMER_NAME` bytes. The error code `MME_INVALID_ARGUMENT` will be returned if the name is too long.

The name of the transformer implicitly describes the type of that transformer, for example '`com.st.dvd.acc.Ac3DecoderMacro`'. This can be confirmed by using `MME_GetTransformerCapability()` to examine the capability of transformer if required.

Comments: Call type: blocking function call.

See also: *MME_InitTransformer*

MME_DeregisterTransformer

MME_SendCommand — Send a command to a specific transformer

Definition:

```
#include <mme.h>

MME_ERROR MME_SendCommand(
              MME_TransformerHandle_t Handle,
              MME_Command_t          *CmdInfo_p)
```

Arguments:

`Handle`	Handle of the targeted transformer.
`CmdInfo_p`	Pointer to an allocated structure that contains the parameters of the command.

Returns:

`MME_SUCCESS`	The command has been successfully inserted in the command queue waiting to be processed by MME.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_NOMEM`	The memory required to complete this command is not available.
`MME_INVALID_HANDLE`	The handle does not refer to an existing transformer
`MME_INVALID_ARGUMENT`	`CmdInfo_p` is invalid.

Description: Send a command and its associated parameters to a specific transformer. The command's parameters are passed using the structure `MME_Command_t`. This in turn uses a number of sub-structures to describe specific parameters.

When inserted the `MME_CommandState_t` of the command is set to `MME_COMMAND_PENDING`.

Comments: Call type: non blocking function call.

See also: *MME_AbortCommand*

MME_WaitCommand

MME_Command_t

MME_Term

Terminate a connection with MME

Definition:

```
#include <mme.h>

MME_ERROR MME_Term(void)
```

Arguments: None

Returns:

MME_SUCCESS	The operation complete correctly.
MME_DRIVER_NOT_INITIALIZED	The MME driver has not been initialized.
MME_HANDLES_STILL_OPEN	Could not terminate, not all transformers have been terminated.
MME_COMMAND_STILL_EXECUTING	Could not terminate due to locally registered transformer instantiations still being active.

Description: Terminate a connection with a MME.

Free all the associated memory space.

Comments: Call type: blocking function call.

See also: *MME_Init*

MME_TermTransformer — Terminate a transformer instance

Definition:

```
#include <mme.h>

MME_ERROR MME_TermTransformer(MME_TransformerHandle_t handle)
```

Arguments:

`handle`	Handle of the transformer to terminate.

Returns:

`MME_SUCCESS`	The operation complete correctly.
`MME_DRIVER_NOT_INITIALIZED`	MME has not been initialized.
`MME_INVALID_HANDLE`	Invalid transformer handle.
`MME_COMMAND_STILL_EXECUTING`	A command is still executing.

Description: Terminate a transformer instance and free all the associated resources.

A transformer can not be terminated while a command is executing.

Comments: Call type: blocking function call.

See also: *MME_InitTransformer*

MME_AbortCommand

MME_WaitCommand — Block waiting for command completion

Definition:

```
#include <mme.h>

MME_ERROR MME_WaitCommand (MME_TransformerHandle_t Handle,
                           MME_CommandId_t CmdId,
                           MME_Event_t *Event_p,
                           MME_Time_t Timeout)
```

Arguments:

`Handle`	Handle of the associated transformer.
`CmdId`	Command identity of the command to wait for.
`Event_p`	Pointer to an `MME_Event_t` to return associated event type in.
`Timeout`	Timeout period in milliseconds or `MME_TIMEOUT_INFINITE`.

Returns:

`MME_SUCCESS`	The command was successfully waited for.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not been initialized.
`MME_INVALID_HANDLE`	The handle does not refer to an existing transformer, or the supplied `CmdId` is invalid.
`MME_INVALID_ARGUMENT`	One or more of the supplied arguments are invalid.
`MME_COMMAND_TIMEOUT`	The command did not complete before the `Timeout` period expired, this might be due to the CPU hosting the transformer crashing or failing to respond.
`MME_SYSTEM_INTERRUPT`	The wait operation was interrupted.
`MME_ICS_ERROR`	An ICS subsystem error occurred.
`MME_DATA_UNDERFLOW`	A data underflow event occurred.
`MME_DATA_OVERFLOW`	A data overflow event occurred.
`MME_TRANSFORMER_TIMEOUT`	The command has been timed out due to

Description:

`MME_WaitCommand` can be used to block waiting for a command to complete that has been issued with `MME_SendCommand`, and given the `MME_CommandEndType_t CmdEnd` value of `MME_COMMAND_END_RETURN_WAKE` (passed in using `MME_Command_t`).

On successful completion of `MME_WaitCommand`, `MME_SUCCESS` is returned and the associated event type is updated through `Event_p`. For a normal completion this is set to `MME_COMMAND_COMPLETED_EVT`. Once a `MME_WaitCommand` has completed successfully it is invalid to issue a subsequent `MME_WaitCommand` using the same `CmdId`.

It is also possible for `MME_WaitCommand` to return `MME_DATA_UNDERFLOW` or `MME_DATA_OVERFLOW` status values. In this case the event is set to `MME_DATA_UNDERFLOW_EVT` or `MME_NOT_ENOUGH_MEMORY_EVT` respectively.

When such an error occurs the driver should respond as appropriate to the error condition, it can then subsequently call `MME_WaitCommand` on the same `CmdId`.

If the command does not complete before the specified timeout then `MME_COMMAND_TIMEOUT` is returned. This error code is also returned if the CPU hosting the transformer fails. The event completion code is set to `MME_TRANSFORMER_TIMEOUT`.

Setting the `Timeout` parameter to `MME_TIMEOUT_INFINITE` causes `MME_WaitCommand` to block indefinitely for command completion. However, it returns `MME_COMMAND_TIMEOUT`, if the command is executing on a CPU that is determined to have failed.

Comments: Call type: blocking function call.

See also: *MME_SendCommand*

MME_Command_t

MME_CommandEndType_t

MME_Version — Query the MME system version string

Definition:

```
#include <mme.h>

const char MME_Version (void)
```

Arguments: None.

Returns: A pointer to the MME version string.

Errors: None.

Context: Callable from task and interrupt context. Can be called before `MME_Init`.

Description: Return a pointer to the MME version string.

This string takes the form:

{major number}.{minor number}.{patch number} : [*text*]

That is, a major, minor and release number, separated by decimal points, and optionally followed by a colon and a text string.

4.2 MME constants, enums and types

MME_AbortCommand_t — Abort a transform command

Definition:

```
#include <mme.h>

MME_ERROR (*MME_AbortCommand_t) (
              void            *context,
              MME_CommandId_t  commandId)
```

Arguments:

`context`	Transformer context data.
`commandId`	The command identifier.

Returns:

`MME_SUCCESS`	Success.
`MME_INVALID_ARGUMENT`	An invalid `commandId` parameter has been specified.
`MME_INVALID_COMMAND`	The transformer is active and cannot be aborted.

Description: Abort a transform command.

This function will be called when an abort command request is made on the host and the command has been submitted to the transformer. The behavior is transformer-specific; transformers that do not support command aborting must return `MME_INVALID_COMMAND`.

MME_AllocationFlags_t — Describes properties of allocated memory

Definition:

```
#include <mme.h>

typedef enum
{
                MME_ALLOCATION_PHYSICAL,
                MME_ALLOCATION_CACHED,
                MME_ALLOCTION_UNCACHED
} MME_AllocationFlags_t;
```

Description: Flags to describe the memory properties of allocated memory.

Flags may be 'or-ed' together to obtain sensible combinations of allocation properties.

In general the use of `MME_ALLOCTION_UNCACHED` should be used with caution since it is potentially harmful to performance. Its use should be limited to applications where the underlying MME data buffer is used outside of the MME interface by cache incoherent hardware. Even in this case it is preferable to use cached memory and manage the caches if the host performs any reads or writes to the data buffer.

Constants:

`MME_ALLOCATION_PHYSICAL`	Require the allocated memory to be contiguous within its physical address space.
`MME_ALLOCATION_CACHED`	Require the allocated memory to be accessed through the cache on the host processor.
`MME_ALLOCATION_UNCACHED`	Require the allocated memory to be accessed directly by the host processor.

See also: *MME_Command_t*

MME_SendCommand

MME_Command_t

Defines the parameters of the command sent to a transformer as a number of sub-structures

Definition:

```
#include <mme.h>

typedef struct
{
                MME_UINT              StructSize;
                MME_CommandCode_t     CmdCode;
                MME_CommandEndType_t  CmdEnd;
                MME_Time_t            DueTime;
                MME_UINT              NumberInputBuffers;
                MME_UINT              NumberOutputBuffers;
                MME_DataBuffer_t    **DataBuffers_p;
                MME_CommandStatus_t   CmdStatus;
                MME_UINT              ParamSize;
                MME_GenericParams_t   Param_p;
} MME_Command_t;
```

Description: Defines the parameters of the command passed to the `MME_SendCommand` function.

While the command is in progress the master copy of all data structures passed by pointer is owned by the companion which may not be cache coherent with the host processor. As such writes to any data structure are illegal and reads from output buffers should be avoided.

Fields:

`StructSize`	Size of the structure (in bytes).
`CmdCode`	Command to be performed.
`CmdEnd`	Command mode completion. Specify whether or not an event shall be generated when the command completes. (refer to `MME_CommandEndType_t` definition).
`DueTime`	Time before the command has to be completed by the MME.
`NumberInputBuffers`	Number of read only buffers to be supplied to the transformer.
`NumberOutputBuffers`	Number of read/write buffers to be supplied to the transformer. Note that overuse of output buffers, leads to poor cache utilization due to excess cache purging.
`DataBuffers_p`	Pointer to an array of pointers to data buffers containing all the input buffers followed by all the output buffers. As such the length of the array is equal to or greater than `NumberInputBuffers` + `NumberOutputBuffers`.

CmdStatus	An MME_CommandStatus_t structure that will evolve during the processing of the command. Fields of this structure will be filled by MME on either the host or the companion (refer to MME_CommandStatus_t definition).
ParamSize	Size in bytes of the associated parameter array, typically obtained using MME_LENGTH_BYTES().
Param_p	Pointer to an allocated parameter array that contains information required to perform the requested operation.

See also:

MME_SendCommand

MME_CommandCode_t

MME_CommandEndType_t

MME_CommandStatus_t

MME_DataBuffer_t

MME_GenericParams_t

MME_Time_t

MME_UINT

MME_CommandCode_t

Constant used by MME_Command_t - defines command code

Definition:

```
#include <mme.h>

typedef enum
{
        MME_SET_GLOBAL_TRANSFORM_PARAMS,
        MME_TRANSFORM,
        MME_SEND_BUFFERS
} MME_CommandCode_t;
```

Description:

Defines the code of the command to be executed onto the MME.

The `MME_SET_GLOBAL_TRANSFORM_PARAMS` is defined in order to limit communication between host and companion by setting common parameters that will be shared by the next transformations. This command code is to be called only when those common parameters change and to send changes to the companion. Parameters which are transformation specific should be part of the parameters of the `MME_TRANSFORM` command.

Constants:

`MME_SET_GLOBAL_TRANSFORM_PARAMS`	Set “generic” parameters for a specific transformer for subsequent transform requests. Those parameters will be used by the transformer until parameters are changed by another call to set generic parameters.
`MME_TRANSFORM`	Commence a transform operation, that is process data according to the current context and the parameters associated with the command.
`MME_SEND_BUFFERS`	Provide input and/or output buffers.

See also:

MME_Command_t

MME_SendCommand

MME_CommandEndType_t

Constant used by MME_Command_t - defines completion behavior of command

Definition:

```
#include <mme.h>

typedef enum
{
                MME_COMMAND_END_RETURN_NO_INFO,
                MME_COMMAND_END_RETURN_NOTIFY,
                MME_COMMAND_END_RETURN_WAKE
} MME_CommandEndType_t;
```

Description: Defines the behavior on the completion of a `MME_SendCommand` command.

Constants:

`MME_COMMAND_END_RETURN_NO_INFO`

No event will be generated when the command completes. But the `MME_CommandStatus_t` structure passed when calling `MME_SendCommand` is filled giving the application the opportunity to retrieve the command status.

`MME_COMMAND_END_RETURN_NOTIFY`

An event will be generated when the command completes by calling the callback function passed when the transformer was instantiated.

`MME_COMMAND_END_RETURN_WAKE`

When the command completes, no callback is issued. Instead the command is signalled to complete any calls of `MME_WaitCommand`.

See also:

MME_UINT

MME_Command_t

MME_SendCommand

MME_WaitCommand

MME_CommandId_t — Command identifier

Definition:

```
#include <mme.h>

typedef MME_UINT MME_CommandId_t
```

Description: Used to identify a command. This identifier is allocated by MME when a call is made to `MME_SendCommand`.

MME_CommandState_t — Defines valid states for a command

Definition:

```
#include <mme.h>

typedef enum
{
                MME_COMMAND_PENDING,
                MME_COMMAND_EXECUTING,
                MME_COMMAND_COMPLETED,
                MME_COMMAND_FAILED
} MME_CommandState_t;
```

Description: Defines the different states a command may have. See *Section 2.7: Issuing commands on page 26*.

Constants:

MME_COMMAND_PENDING	Command waiting to be processed by the MME.
MME_COMMAND_EXECUTING	The command is the currently executed by the MME.
MME_COMMAND_COMPLETED	The command has been completed by the transformer and results are available for the application.
MME_COMMAND_FAILED	Errors occurred during command processing by the transformer or by MME.

See also: *MME_CommandStatus_t*

MME_CommandStatus_t

Sub-structure to MME_Command_t - returns transformation results

Definition:

```
#include <mme.h>

typedef struct
{
                MME_CommandId_t      CmdId;
                MME_CommandState_t   State;
                MME_Time_t           ProcessedTime;
                MME_ERROR            Error;
                MME_UINT             AdditionalInfoSize;
                MME_GenericParams_t  AdditionalInfo_p;
} MME_CommandStatus_t;
```

Description:

Structure filled by MME with the results of the corresponding transformation actions performed.

With the exception of the additional parameters all members of the `MME_CommandStatus_t` are populated by the MME as part of `MME_SendCommand`.

Note: `MME_CommandStatus_t` *is not supplied directly to any MME API call (it forms part of the definition of* `MME_Command_t` *and is therefore not prefixed by a structure size).*

Fields `AdditionalInfoSize` and `AdditionalInfo_p` are filled by the caller before calling the `MME_SendCommand` function. The data pointed to by `AdditionalInfo_p` is transported bidirectionally - that is, it is sent from the host to the companion when the command is submitted and back from the companion to the host when the command completes.

Field `CmdId` is filled by the `MME_SendCommand` function.

Fields `ProcessedTime` and `Error` are filled by MME itself. These fields are relevant only when the command has been processed that is when the field `State` has turned to the `MME_COMMAND_COMPLETED` or `MME_COMMAND_FAILED` value.

This structure is owned by the transformer once passed into the `MME_ProcessCommand_t` transformer entry point and the `State` field may be modified by the transformer to reflect that a transform has been deferred. See *Section 3.4.2: Deferred commands on page 38*.

Fields:

`CmdId`	Unique identifier of the command the structure is related to. This field is filled by the `MME_SendCommand` function.
`State`	State of the command.
`ProcessedTime`	Time spent processing the command.
`Error`	Command status as a result of processing.
`AdditionalInfoSize`	Size in bytes of the associated parameter array, typically obtained using `MME_LENGTH_BYTES()`.
`AdditionalInfo_p`	Pointer to an allocated parameter array where the MME can store additional info related to the performed transformation (transformer specific).

See also: *MME_Command_t*

MME_CommandState_t

MME_SendCommand

MME_DataBuffer_t

Data buffer structure as returned by MME_AllocDataBuffer

Definition:

```
#include <mme.h>

typedef struct
{
        MME_UINT StructSize;
        void*UserData_p;
        MME_UINT Flags;
        MME_UINT StreamNumber;
        MME_UINT NumberOfScatterPages;
        MME_ScatterPage_t*ScatterPages_p;
        MME_UINT TotalSize;
        MME_UINT StartOffset;
} MME_DataBuffer_t;
```

Description: Definition of one (possibly scattered) buffer belonging to one stream.

A data buffer consists of a list of one or several scatter pages. Each page describes a contiguous, linear memory block giving the transformer a memory space to work with.

Fields:

`StructSize`	Size of the structure (in bytes).
`UserData_p`	Application specific data to aid data structure lookup from callbacks.
`Flags`	Buffer specific flags.
`StreamNumber`	Identifies the stream to which the buffer belongs.
`NumberOfScatterPages`	Number of scatter pages the buffer is composed of (that is the number of entries of the `MME_ScatterPage_t` array).
`ScatterPages_p`	Pointer to an array of scatter pages.
`TotalSize`	Amount of memory available for this buffer, that is, the sum of the memory size of the scatter pages this buffer comprises.
`StartOffset`	Points to first valid byte in (scattered) buffer.

See also: *MME_ScatterPage_t*

MME_Command_t

MME_DataFormat_t — Defines the data format of a transformer's I/O

Definition:

```
#include <mme.h>

typedef struct
{
                unsigned char FourCC[4];
} MME_DataFormat_t;
```

Description: Used to define the format of the data a transformer supports for its input or output.

Refer to *http://www.webartz.com/fourcc/* for a complete description of the FOURCC definition.

Fields:

`FourCC`	Contains the format defined using its associated Four Character Code (FOURCC).

MME_DBG_FLAGS Describes properties of the debug logging flags

Definition:

```
#include <mme.h>

typedef enum mme_debug_flags
{
                MME_DBG_ERR          = 0x0001,
                MME_DBG_INIT         = 0x0002,
                MME_DBG_MANAGER      = 0x0004,
                MME_DBG_RECEIVER     = 0x0010,
                MME_DBG_TRANSFORMER  = 0x0020,
                MME_DBG_EXEC         = 0x0040,
                MME_DBG_COMMAND      = 0x0100,
                MME_DBG_BUFFER       = 0x0200,
} MME_DBG_FLAGS;
```

Description: MME debugging flags specified as a bitmask value. The valid set of debugging flags are detailed in *Table 7*.

Table 7. MME debug logging flags

Debug flag	Description
MME_DBG_ERR	Log all error messages.
MME_DBG_INIT	Log all initialization actions.
MME_DBG_MANAGER	Log all MME administration actions.
MME_DBG_RECEIVER	Log all transformer receiver actions.
MME_DBG_TRANSFORMER	Log all client side transformer actions.
MME_DBG_EXEC	Log all transformer execution loop actions.
MME_DBG_COMMAND	Log all transformer command actions.
MME_DBG_BUFFER	Log all MME data buffer actions.

See also: *MME_DebugFlags*

MME_ERROR — Status indicator used by all MME functions

Definition:

```
#include <mme.h>

typedef enum
{
                MME_SUCCESS,
                MME_DRIVER_NOT_INITIALIZED,
                MME_NOMEM,
                MME_INVALID_HANDLE,
                MME_INVALID_ARGUMENT,
                MME_UNKNOWN_TRANSFORMER,
                MME_TRANSFORMER_NOT_RESPONDING,
                MME_HANDLES_STILL_OPEN,
                MME_COMMAND_STILL_EXECUTING,
                MME_COMMAND_ABORTED,
                MME_DATA_UNDERFLOW,
                MME_DATA_OVERFLOW,
                MME_TRANSFORM_DEFERRED,
                MME_SYSTEM_INTERRUPT,
                MME_ICS_ERROR,
                MME_INTERNAL_ERROR,
                MME_NOT_IMPLEMENTED,
                MME_COMMAND_TIMEOUT
} MME_ERROR;
```

Description: Status indicator used by all MME functions.

Note: Although `MME_SUCCESS` *is guaranteed to be zero the numeric value of all other error codes is unspecified. Additionally it is not guaranteed that these values will be contiguous.*

Constants:

`MME_SUCCESS`	Command complete successfully.
`MME_DRIVER_NOT_INITIALIZED`	MME or some of its underlying infrastructure has not yet been initialized.
`MME_DRIVER_ALREADY_INITIALIZED`	MME has been initialized already.
`MME_NOMEM`	The system has insufficient resources to complete this request.
`MME_INVALID_HANDLE`	The transformer handle is invalid or out of date.
`MME_INVALID_ARGUMENT`	One or more of the function arguments are invalid (for example: out of range, null pointer, incorrect structure size).
`MME_UNKNOWN_TRANSFORMER`	The requested transformer does not exist.
`MME_INVALID_COMMAND`	The command code is invalid.

MME_TRANSFORMER_NOT_RESPONDING	The transformer is not responding to requests for status.
MME_HANDLES_STILL_OPEN	The operation cannot complete until all transformer handles have been closed.
MME_COMMAND_STILL_EXECUTING	The operation cannot complete until the transformer is idle.
MME_COMMAND_ABORTED	The command did not complete because it was explicitly aborted by the user.
MME_DATA_UNDERFLOW	Insufficient input data to generate a frame of output.
MME_DATA_OVERFLOW	Output buffers are too small to store the transformed data.
MME_TRANSFORM_DEFERRED	A transform has been placed in the deferred state by a transformer.
MME_ICS_ERROR	ICS underlying MME has reported an error.
MME_INTERNAL_ERROR	There is an internal inconsistency.
MME_NOT_IMPLEMENTED	The function is not implemented.
MME_COMMAND_TIMEOUT	An issued command timed out.

MME_Event_t — Valid event codes associated with a command

Definition:

```
#include <mme.h>

typedef enum
{
                MME_COMMAND_COMPLETED_EVT,
                MME_DATA_UNDERFLOW_EVT,
                MME_NOT_ENOUGH_MEMORY_EVT,
                MME_TRANSFORMER_TIMEOUT
} MME_Event_t;
```

Description: Event codes associated with a command.

Events are delivered to the application by the callback mechanism.

Constants:

`MME_COMMAND_COMPLETED_EVT`

A command has been completed by MME. The error code in `MME_CommandStatus_t` describes the state.

`MME_DATA_UNDERFLOW_EVT` The transformer has run out of data before completing an output frame. The error code in `MME_CommandStatus_t` will be `MME_DATA_UNDERFLOW`.

`MME_NOT_ENOUGH_MEMORY_EVT`

The transformer has insufficient output buffers to output a frame. The error code in `MME_CommandStatus_t` will be `MME_DATA_OVERFLOW`.

`MME_TRANSFORMER_TIMEOUT`

The command has been timed out due to the CPU hosting the transformer crashing or failing to respond.

See also: *MME_Command_t*

MME_SendCommand

MME_GenericCallback_t — Call mechanism for communication between transformer and host

Definition:

```
#include <mme.h>

typedef void (*MME_GenericCallback_t)
              MME_Event_t    Event,
              MME_Command_t *CallbackData,
              void          *UserData);
```

Description: Generic callback mechanism for communication between transformer and host.

Guaranteed not to be called in a re-entrant manner.

Fields:

`Event`	Event, associated with either data buffers or command transformations.
`CallbackData`	Pointer to the command structure related to this command.
`UserData`	Reference to user data, provided with the call to `MME_InitTransformer`.

See also: *MME_SendCommand*

MME_InitTransformer

MME_GenericParams_t — Type for data exchange between CPUs

Definition:

```
#include <mme.h>

typedef void* MME_GenericParams_t
```

Description: Generic type used to exchange data between host and companion CPUs.

Fields: None

See also: *MME_Command_t*

MME_GetTransformerCapability_t — Provide transformer capabilities

Definition:

```
#include <mme.h>

MME_ERROR (*MME_GetTransformerCapability_t) (
              MME_TransformerCapability_t *capability)
```

Arguments:

`capability`	Transformer parameters.

Returns:

`MME_SUCCESS`	Success.
`MME_INVALID_ARGUMENT`	An invalid transformer parameter has been specified.

Description: Provide the capabilities of a transformer.

Comments: Call type: blocking function call.

See also: *MME_GetTransformerCapability*

MME_TransformerCapability_t

MME_InitTransformer_t — Create a transformer instance

Definition:

```
#include <mme.h>

MME_ERROR (*MME_InitTransformer_t) (
                MME_UINT               size,
                MME_GenericParams_t    params,
                void                  **context)
```

Arguments:

`size`	Size of the transformer initialization parameters in bytes.
`params`	Transformer initialization parameters.
`context`	Pointer to a location in which to store a transformer instance-specific value.

Returns:

`MME_SUCCESS`	Success.
`MME_INVALID_ARGUMENT`	An invalid transformer parameter has been specified.
`MME_NOMEM`	Insufficient memory available.

Description: Create an instance of a transformer. It is called as a result of a host call to `MME_InitTransformer()`

The `Callback` and `CallbackUserData` fields of the `MME_TransformerInitParams_t` structure are not valid for the transformer.

Comments: Call type: blocking function call.

See also: *MME_InitTransformer*

MME_MAX_TRANSFORMER_NAME

Defines maximum length of a transformer name

Definition:

```
#include <mme.h>

#define MME_MAX_TRANSFORMER_NAME <const unsigned int>
```

Description: The maximum length in bytes of a transformer name.

This constant defines the maximum length of the transformer name that may be passed to `MME_InitTransformer()` and `MME_RegisterTransformer()`.

See also: *MME_InitTransformer*

MME_RegisterTransformer

MME_MemoryHandle_t — Memory registration handle

Definition:

```
#include <mme.h>

typedef MME_UINT MME_MemoryHandle_t;
```

Description: Handle returned by `MME_RegisterMemory`.

Used to identify the memory registration for later function calls. The value of zero is invalid.

See also: *MME_RegisterMemory*

MME_DeregisterMemory

MME_Priority_t — Valid priorities for command execution

Definition:

```
#include <mme.h>

typedef enum
{
                MME_PRIORITY_HIGHEST,
                MME_PRIORITY_ABOVE_NORMAL,
                MME_PRIORITY_NORMAL,
                MME_PRIORITY_BELOW_NORMAL,
                MME_PRIORITY_LOWEST
} MME_Priority_t;
```

Description: The priority at which a command should be executed.

See also: *MME_SendCommand*

MME_ProcessCommand_t — Process the transformer command

Definition:

```
#include <mme.h>

MME_ERROR (*MME_ProcessCommand_t) (
                void          *context,
                MME_Command_t *commandInfo)
```

Arguments:

`context`	Transformer context.
`commandInfo`	Data associated with the command.

Returns:

`MME_SUCCESS`	Success.
`MME_INVALID_HANDLE`	The handle does not refer to an existing transformer.
`MME_INVALID_ARGUMENT`	The `commandInfo` argument is invalid.
`MME_INVALID_COMMAND`	The command embedded in `commandInfo` is invalid.
`MME_NOMEM`	The result of the processing of the input data does not fit in the provided memory space.
`MME_NOMEM`	The result of the processing of the input data does not fit in the provided memory space.
`MME_DATA_UNDERFLOW`	Returned when MME reaches the end of the input buffer without being able to produce the requested output.

Description: This function performs one of the following operations:

- commence a new transform
- set the transformer parameters
- handle the submission of data buffers

Comments: Call type: blocking function call.

See also: *Chapter 3: Writing an MME transformer on page 33*.

MME_ScatterPage_t — Describe a scatter page

Definition:

```
#include <mme.h>

typedef struct {
                void*Page_p;
                MME_UINT Size;
                MME_UINT BytesUsed;
                MME_UINT FlagsIn;
                MME_UINT FlagsOut;
} MME_ScatterPage_t;
```

Description:

Describe a scatter page, that is a linear memory range within a data buffer.

`BytesUsed` is meaningless until the transformation completes. It is filled by the transformer with the number of bytes it wrote into this page while processing data.

The `FlagsIn` field and `FlagsOut` fields are used to pass additional information about the scatter page. The `FlagsIn` field is used to pass state from the host to the transformer. The `FlagsOut` field is used to pass state from the transformer to the host. Both fields are divided into two regions - the MME region and the application region. The MME region is the upper 8 bits; unused bits in the MME region are reserved and **must** be set to zero.

The remaining 24 bits are available to the application and transformer, see *Table 2: MME_ScatterPage_t FlagsIn and FlagsOut on page 25*.

Fields:

`Page_p`	Address of the memory space.
`Size`	Size of the page (in bytes).
`BytesUsed`	Number of bytes used in this page.
`FlagsIn`	Combination of generic and transformer specific flags.
`FlagsOut`	Combination of generic and transformer specific flags.

See also: *MME_DataBuffer_t*

MME_TermTransformer_t — Terminate a transformer instance

Definition:

```
#include <mme.h>

MME_ERROR (*MME_TermTransformer_t) (void *context)
```

Arguments:

context	Context of the transformer.

Returns:

MME_SUCCESS	Success.
MME_INVALID_HANDLE	The handle does not refer to an existing transformer.
MME_COMMAND_STILL_EXECUTING	A command is still executing on the transformer instance.

Description: Terminate an instance of a transformer and free any resources that the instance uses.

Comments: Call type: blocking function call.

MME_Time_t — Type used by MME_Command_t - describes time

Definition:

```
#include <mme.h>

typedef MME_UINT MME_Time_t;
```

Description: Describe the time in the MME environment.

See also: *MME_Command_t*

MME_TransformerCapability_t — Describe transformer capabilities

Definition:

```
#include <mme.h>

typedef struct {
                MME_UINTStructSize;
                MME_UINTVersion;
                MME_DataFormat_tInputType;
                MME_DataFormat_tOutputType;
                MME_UINTTransformerInfoSize;
                MME_GenericParams_tTransformerInfo_p;
} MME_TransformerCapability_t;
```

Description: Describe the capabilities of a particular transformer.

Note: On multi-processor systems the contents of `TransformerInfo_p` *will only be copied in one direction (companion to host). For this reason all transformers must treat the data pointed to as uninitialized.*

Fields:

`StructSize`	Size of the structure (in bytes).
`Version`	Version of the transformer.
`InputType`	Supported input type.
`OutputType`	Supported output types.
`TransformerInfoSize`	Size in bytes of the associated parameter array, typically obtained using `MME_LENGTH_BYTES()`.
`TransformerInfo_p`	Pointer to an allocated parameter array where the transformer may store specific capabilities of the transformer (transformer specific).

See also: *MME_GetTransformerCapability*

MME_TransformerHandle_t

Transformer handle

Definition:

```
#include <mme.h>

typedef MME_UINT  MME_TransformerHandle_t
```

Description: Handle returned by `MME_InitTransformer`.

Used to identify the transformer for later function calls.

The value of zero is invalid.

See also: *MME_InitTransformer*

MME_TransformerInitParams_t Parameters used to initialize transformer

Definition:

```
#include <mme.h>

typedef struct {
                MME_UINT StructSize;
                MME_Priority_t Priority;
                MME_GenericCallback_t Callback;
                void*CallbackUserData;
                MME_UINT TransformerInitParamsSize
                MME_GenericParams_t TransformerInitParams_p;
} MME_TransformerInitParams_t;
```

Description: Parameters to use to initialize a transformer.

The `Callback` and `CallbackUserData` fields of the `MME_TransformerInitParams_t` structure are not valid for the transformer.

Fields:

`StructSize`	Size of the structure (in bytes).
`Priority`	The transform queue priority.
`Callback`	Function pointer to handle both command and data callbacks.
`CallbackUserData`	Anonymous data provided with the callback. Those data will be passed as parameters every time the transformer will call its associated `CallbackUserData` functions.
`TransformerInitParamsSize`	Size in bytes of the associated parameter array, typically obtained using `MME_LENGTH_BYTES()`.
`TransformerInitParams_p`	Pointer to an allocated parameter array that the contains additional parameters if required (transformer specific).

See also: *MME_InitTransformer*

MME_UINT Type used by MME_Command_t - defines numeric value

Definition:

```
#include <mme.h>

typedef unsigned <qualifier> int MME_UINT
```

Description: Unsigned integer type of at least 32 bits.

On MME implementations that share memory structures directly, the size of this type will be identical on all processors. An MME implementation that copies structures may define this type differently on each processor to maximize efficiency.

5 Overview of the inter-core system (ICS)

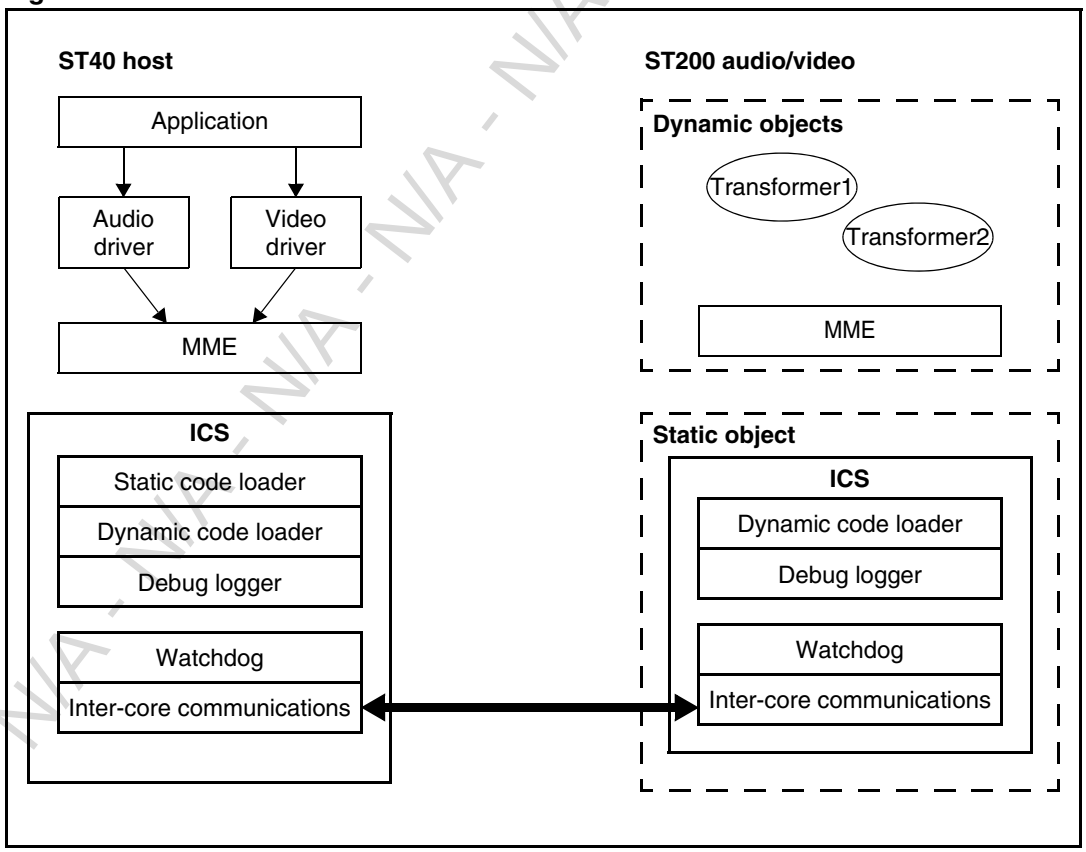
ICS in Multicom is a run-time system that provides program execution management and communications between all the CPUs on an ST SoC. ICS replaces the EMBX layer in previous versions of Multicom, with a simpler approach to communication, and with the addition of facilities for loading, starting, monitoring and recovering from errors in other CPUs in the SoC. The intention is that the ICS run-time will be loaded onto all the CPUs during system boot and initialization and then used as a building block to provide higher-level features and functionality.

The ICS API reference lists all the functions in the ICS API. The purpose of this overview is to introduce the key concepts of ICS and the key functions associated with each concept.

In some cases the ICS APIs rely on some underlying system libraries available with the tools for compiling programs on the CPUs, notably ELF file loading and dynamic module loading libraries.

*Note: ICS API function names begin with "`ics_`" (system initialization and loading functions) or "`ICS_`" (system initialization, communication, memory management and support functions). The function `ICS_cpu_init` or `ics_cpu_init` **must** be called on the current CPU before any function beginning with `ICS_` is called.*

Figure 7. Multicom 4 - ICS context



5.1 Summary of ICS facilities

The following is a summary of the main facilities provided by the ICS API.

- ICS initialization and system loading, see *Section 5.2 on page 107*

  ICS provides API calls to allow the application to set up ICS and to facilitate the loading and execution of the initial software in each CPU in the SoC.
- Channel-based communication, see *Section 5.3 on page 111*

  The lowest-level communication primitive provided by ICS is that of a channel. A channel consists of a unidirectional, fixed-length, fixed-size FIFO used for communicating between a pair of CPUs.
- Port-based communication, see *Section 5.4 on page 113*

  ICS incorporates a port-based communication system closely based on the original EMBX Port model. The ICS port API allows ASCII named ports to be registered on each CPU and then messages can be targeted at those ports.
- Memory region management, see *Section 5.5 on page 115*

  Zero-copy data passing between the CPUs is facilitated by sharing physically contiguous regions of memory between CPUs. To this end, ICS provides functions for managing physically contiguous regions of memory and mapping them for all the communicating CPUs.
- Name server, see *Section 5.6 on page 117*

  The ICS Port model is based on named port handles which can be looked up using a central name sever. This name service has been exported as a primary API so that programmers can register data objects associated with an ASCII string.
- Dynamic module loading, see *Section 5.7 on page 117*

  ICS provides a dynamic module loading system based on the relocatable loader library provided with the ST40 and ST200 Micro Toolsets, using OS21. This API allows code modules to be loaded and unloaded into any of the running CPUs on demand.
- CPU watchdog support, see *Section 5.8 on page 118*

  In order to facilitate fault detection and recovery, ICS provides a CPU watchdog API that allows callback functions to be registered. These callback functions are then triggered whenever one of the monitored CPUs fails to update a internal counter within a fixed time period.
- Debug logging support, see *Section 5.9 on page 119*

  ICS provides a debugging facility where text output from the running CPU cores is logged to a cyclic buffer. APIs are provided to dump out these logs on a per-CPU basis.

5.2 ICS initialization and system loading

There are three stages to ICS initialization and system loading. these are described in the following sections:

- *Section 5.2.1: ICS configuration and setup*
- *Section 5.2.2: CPU loading and initialization*
- *Section 5.2.3: ICS initialization and termination on page 110*

5.2.1 ICS configuration and setup

The final intended method for configuring and setting up ICS for a specific SoC and application is not yet defined; this will be done at a later stage of the project.

The ICS system library is linked into the initial application binary (or Linux kernel in the case of Linux systems) to be booted into each of the CPUs and each instantiation is provided with the information needed to set up the ICS system on that CPU so that it can share memory and (where appropriate) control the other CPUs. It is expected that this will be derived from a BSP or set of registry entries which define the required information.

5.2.2 CPU loading and initialization

The facilities for loading and starting other CPUs on the SoC are new in this version of Multicom: similar facilities were not available in earlier versions of Multicom, but are defined now so that the API provides a full self-contained set of facilities for multi-CPU system development.

Note: It is not necessary to use the Multicom APIs for this purpose; the loading, starting and restarting of CPUs can be done by application code directly and/or using other libraries as available.

In the ICS model an SoC consists of a set of CPUs, one of which is typically denoted as the *master* CPU (sometimes called the *host* CPU) which has the responsibility of loading and starting the system. The other CPUs in the SoC are typically denoted the *companion* CPUs.

It is envisaged that the master CPU will be used to load the initial software for the other cores and hence the firmware loading and CPU execution ICS functions may not be available on the other CPUs. Once ICS is up and running on all CPUs there is no significant distinction between a master and a companion, so that a companion CPU could communicate with another companion CPU and could load dynamic modules to another companion CPU, if the ICS system has been configured to allow this (and this may depend on the design of the SoC). The master CPU in an ICS system would typically run the nameserver service and monitor and manage the rest of the system in the event of a fault, but it is possible to delegate these functions to other CPUs in the SoC in a system design.

In the current implementation of ICS the master CPU may be running Linux or OS21/OSPlus. The companion CPUs are expected to be running OS21.

Note: A dual- (or multi-) core SMP processor under the control of a single OS would be considered a single CPU for the purposes of this definition.

Each CPU is given a logical number `0..`*`N`* for the purposes of ICS. The logical numbers for the set of CPUs on an SoC are defined in the Multicom BSP or Registry information.

An ICS function operating on a CPU is given the CPU logical number to which the operation should be applied. An ICS function operating on one or more CPUs is given a bitmask in which the bits that are set define the CPUs to which the operation applies.

CPU query functions

Table 8. CPU query functions

Function name	Description
`ics_cpu_name`	Query the BSP/Registry for CPU name string
`ics_cpu_type`	Query the BSP/Registry for CPU type string

Table 8. CPU query functions

Function name	Description
`ics_cpu_lookup`	Query the BSP/Registry for CPU logical number
`ics_cpu_self`	Query the CPU Logical number
`ics_cpu_mask`	Query the logical CPU bitmask

These functions query the CPU database set up from the BSP. The first two return the name string and type string respectively for a specified CPU. `ics_cpu_lookup` returns the logical number given a CPU name. `ics_cpu_self` returns the CPU logical number of the CPU on which it is executing. `ics_cpu_mask` returns the bitmask defining the set of CPUs on the SoC on which ICS has been specified to run.

The ICS functions for CPU loading and starting are given in *Table 9* and *Table 10*.

Table 9. Program loading functions

Function name	Description
`ics_load_elf_file`	Load and unpack an ELF file
`ics_load_elf_image`	Unpack an ELF memory image

`ics_load_elf_file` and `ics_load_elf_image` load an ELF file to memory, either from a file stored in a local file system, or from an ELF file image stored in memory. The ELF start/entry address is returned to the caller in the `entryAddrp` argument.

Table 10. CPU control functions

Function name	Description
`ics_cpu_reset`	Reset and stop CPU execution
`ics_cpu_start`	Start execution of a CPU

These functions reset and start code running on the CPU specified. The CPU number must not refer to the CPU on which the function is being called; in this case an error is returned. The running CPU must have the capability to control the targeted CPU; otherwise an error is returned.

Other `ics` helper functions are given in *Table 11* and *Table 12*.

Table 11. Information functions

Function name	Description
`ics_cpu_version`	Query the ICS system version string
`ics_err_str`	Return an ICS error string

Table 12. Heap functions

Function name	Description
ics_heap_create	Create an ICS heap
ics_heap_destroy	Destroy an ICS heap
ics_heap_alloc	Allocate a buffer from an ICS memory heap
ics_heap_free	Release a buffer back to an ICS heap
ics_heap_base	Query ICS heap for virtual base address
ics_heap_pbase	Query ICS heap for physical base address
ics_heap_size	Query ICS heap for size

The ics_heap functions carry out typical heap operations for ICS heaps and are intended to be used by an application in conjunction with the memory region functions for buffer management.

5.2.3 ICS initialization and termination

Code running on each of the CPUs in the SoC has to initialize the ICS system on that CPU before it can use ICS functions. This is done with the function ICS_cpu_init. All CPUs wishing to communicate, must first call this function. However, they will not synchronize or communicate with each other until necessary.

ICS_cpu_init also takes a flag argument; ICS_INIT_CONNECT_ALL and setting this flag causes initialization to be synchronized between the CPUs. In this case, all CPUs must call ICS_cpu_init (setting ICS_INIT_CONNECT_ALL) and each call will only return when all CPUs defined in the CPU set have successfully called ICS_cpu_init. If one or more CPUs in the set fail to call ICS_cpu_init then the blocked calls will eventually return after a pre-defined timeout.

Table 13. Initialization and termination functions

Function name	Description
ICS_cpu_init	Initialize the ICS system on a CPU
ICS_cpu_term	Terminate the ICS system on a CPU
ICS_cpu_info	Query the ICS CPU configuration

ICS_cpu_term is called when an application is shutting down and is used to close down the local ICS system.

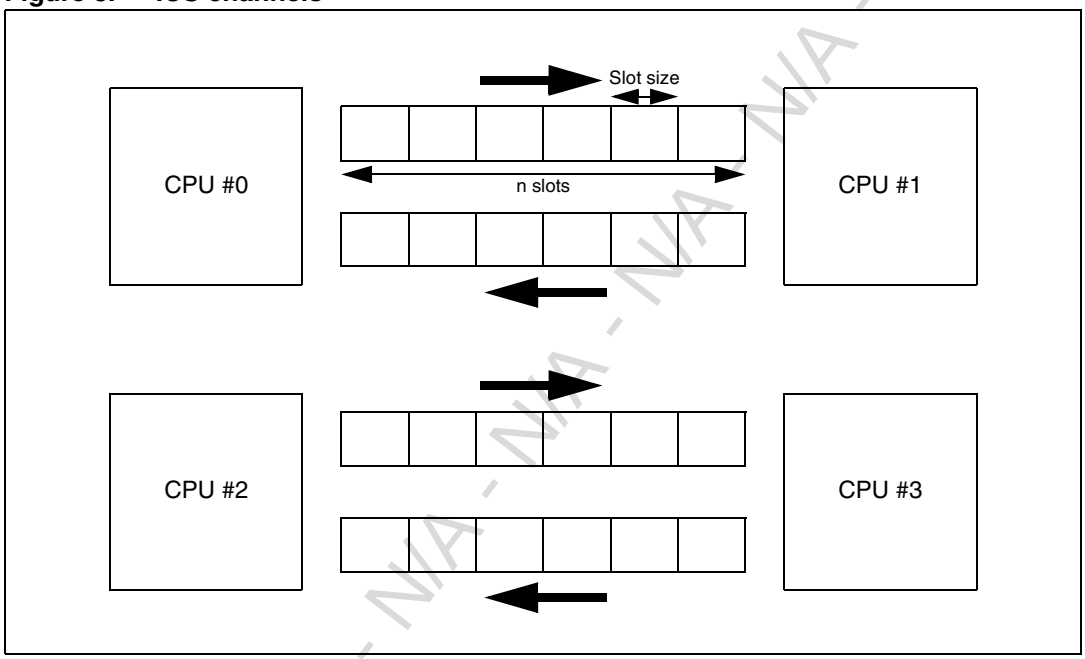
ICS_cpu_info identifies the caller's logical CPU number and the also the set of CPUs running ICS defined in a bitmask.

5.3 Channel-based communication

The lowest-level communication primitive provided by ICS is that of a channel. A channel consists of a point-to-point unidirectional FIFO (first-in, first-out) queue of message slots used for communicating between a pair of CPUs. The length of the FIFO and the size of the slots can be chosen on creation. APIs for data sending and for process and interrupt based data reception are provided. See *Figure 8*.

Channels provide the lowest-overhead and lowest-latency communications within the ICS system. As such, the programmer is responsible for all protocol handling and flow control.

Figure 8. ICS channels



No equivalent low-level API was provided in previous versions of Multicom, causing a number of alternative very low-level communication libraries to be written for different applications; the intention here is to provide this low level of communication in a standardized form.

Table 14. Channel functions in ICS

Function name	Description
`ICS_channel_alloc`	Allocate an ICS communication channel
`ICS_channel_free`	Free an ICS channel
`ICS_channel_open`	Open a send channel for communication
`ICS_channel_close`	Close a send channel
`ICS_channel_send`	Send a buffer using an ICS send channel
`ICS_channel_recv`	Blocking call to receive a buffer from an ICS channel
`ICS_channel_release`	Release an ICS channel FIFO buffer
`ICS_channel_unblock`	Unblock a blocked ICS channel

`ICS_channel_alloc` is called on the CPU that is to be the receiving end of the channel; it defines a callback handler function for the channel, the FIFO length and slot size, and optionally the memory base for the FIFO data. The callback function is to be invoked in interrupt context whenever a new entry arrives in the FIFO (and may be NULL if a process based approach to receiving is wanted).

In order to send, using an ICS channel, it must first be opened. `ICS_channel_open` opens a channel for sending and returns a handle to be used for send operations. Opening a channel is only necessary for the sender; the receiver implies opening by the allocation. The channel handle may be one opened locally by `ICS_channel_alloc`, by looking up a channel object in the name server (see *Section 5.6: Name server on page 117*) or one passed to the sender by some other mechanism.

`ICS_channel_send` sends a buffer to a channel using a send channel handle.

The buffer is processed by either:

- invoking the callback function at the receiver, or
- if no callback function is defined, the receiver calling an `ICS_channel_recv` function; on returning from the function a pointer to the buffer is supplied so that the receiving application can process it

`ICS_channel_send` will return an error if there is no free slot in the FIFO.

The callback function associated with a channel has prototype:

```
ICS_ERROR (*ICS_CHANNEL_CALLBACK) (ICS_CHANNEL channel,
ICS_VOID *param, void *buf)
```

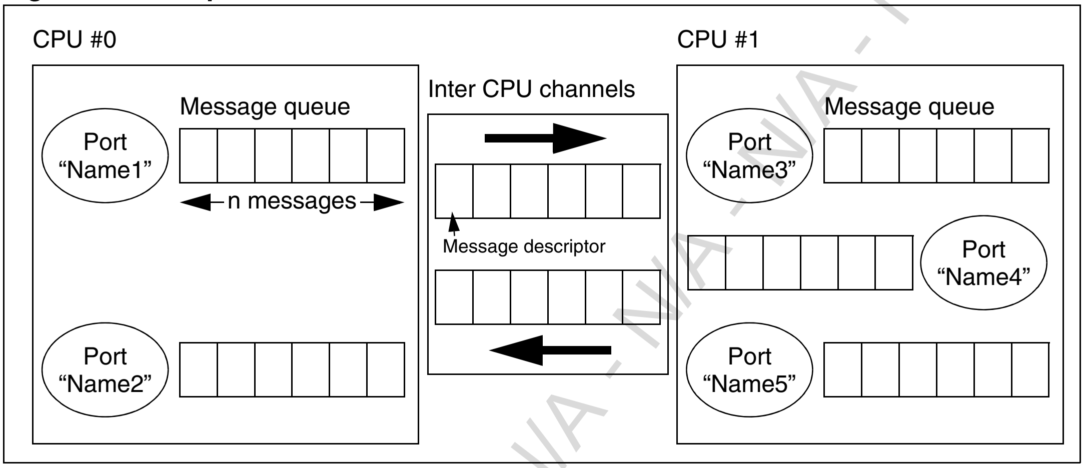
The callback function is supplied with the channel handle, a parameter defined in the `channel_alloc` call, and a pointer to the buffer sent. Normally the handler will copy and consume the buffer and then call `ICS_channel_release`. If the handler is unable to consume the buffer it will return `ICS_FULL` and the FIFO will be blocked until the function `ICS_channel_unblock` is called.

`ICS_channel_recv` blocks on a receive channel until a new buffer arrives, at which point it returns with a pointer to the new buffer. Once the application has processed the buffer it should call `ICS_channel_release` with the buffer pointer.

5.4 Port-based communication

ICS incorporates a port-based communication system similar to the EMBX port model provided in previous versions of Multicom, see *Figure 9*. The ICS port API allows named ports to be registered on each CPU and then messages can be sent to those ports. The exact location of the target port is not required to be known to the sender and hence ports allow services to be abstracted away from their CPU location. APIs are provided for sending messages on ports and for process and interrupt based handling of message reception.

Figure 9. ICS port model



Ports can provide copy-based short message passing (inline) and zero-copy message passing. For zero-copy message passing, the memory buffers must be in memory regions that have been pre-registered with the ICS system (see *Section 5.5: Memory region management on page 115*).

Ports can be either anonymous or named. Named ports are registered with a global name server.

Ports provide a many-to-one communication model. Multiple processes (potentially on different CPUs) can send to a port. Ports implement a FIFO queue of messages between senders and receiver.

Table 15. Port and message functions in ICS

Function name	Description
`ICS_port_alloc`	Allocate an ICS port
`ICS_port_cpu`	Return the logical CPU number associated with the port
`ICS_port_free`	Free and close an ICS port
`ICS_port_lookup`	Look up an ICS port handle
`ICS_msg_send`	Send a message buffer to an ICS port
`ICS_msg_recv`	Blocking call to receive a message on an ICS port
`ICS_msg_post`	Post an asynchronous receive on an ICS port
`ICS_msg_cancel`	Cancel an asynchronous port receive

Table 15. Port and message functions in ICS (continued)

Function name	Description
ICS_msg_test	Test an asynchronous port receive event
ICS_msg_wait	Block and wait for an asynchronous port receive event

ICS_port_alloc allocates an ICS port (for receiving) on the local CPU. Ports can either be named or they can be local and anonymous (if NULL is passed as the name). All named ports are registered with a global name server, from which they can be discovered using the function ICS_port_lookup. ICS_port_alloc defines a callback function to be invoked in interrupt context whenever a new message arrives at the port (and may be NULL if a process based approach to receiving is wanted). ICS_port_alloc returns a port handle for use in subsequent functions.

ICS_port_free frees up and closes a previously allocated port, including de-registering its name from the global name server.

ICS_msg_send is used to send a message to a port, based on a handle obtained from a local allocation or through a lookup of a port name. The message data is presented as a virtual address in an argument to the function. An mflags argument allows the caller to control how the data is transferred and how it is presented to the target receiver: for example, if the data is to be copied inline or if a buffer has been provided which allows the transfer to be done using zero-copy techniques.

There are two approaches to receiving a message from a port: process or interrupt based.

Interrupt based receiving uses the callback function defined at port allocation time, which has prototype:

```
ICS_ERROR (*ICS_PORT_CALLBACK) (ICS_PORT port, ICS_VOID *param,
ICS_MSG_DESC *rdesc)
```

The callback function is supplied with a parameter defined by the ICS_port_alloc call, plus a pointer to a message descriptor which supplies the data sent. Normally the handler will process the message and return ICS_SUCCESS. If the handler is unable to consume the buffer it will return ICS _FULL and the message will be held on the port message queue after which ICS_msg_recv or ICS_msg_post must be called to process the message.

The functions ICS_msg_recv, ICS_msg_post, ICS_msg_cancel, ICS_msg_test and ICS_msg_wait are used for receiving from a process. ICS_msg_recv can be used to synchronously receive a message from a port; it blocks on the port and returns a new message in a message descriptor when it arrives. ICS_msg_post posts an asynchronous receive operation to the port, supplies a message descriptor to hold the message when it arrives and returns an event handle which can be used by ICS_msg_test and ICS_msg_wait to test or wait for the arrival of the message. ICS_msg_cancel can be used to cancel an asynchronous port receive operation.

Note: New incoming messages will always be matched to the posted receiving descriptors in the order the ICS_msg_post functions were called.

5.5 Memory region management

Functions are provided in ICS to manage memory regions in the SoC's memory; these can be defined and mapped for use by all the CPUs, allowing common regions of physical memory to be accessible by all CPUs, and facilitating zero-copy message-passing between the CPUs. A memory region must be aligned on an ICS page size boundary and the size must be a multiple of the ICS page size (`ICS_PAGE_SIZE`).

Table 16. Memory region mapping functions

Function name	Description
`ICS_region_add`	Add a region to the local and remote CPU region tables
`ICS_region_remove`	Remove a region from the local and remote CPU region tables
`ICS_region_virt2phys`	Translate a local virtual address into a physical one
`ICS_region_phys2virt`	Translate a physical memory region address into a virtual address

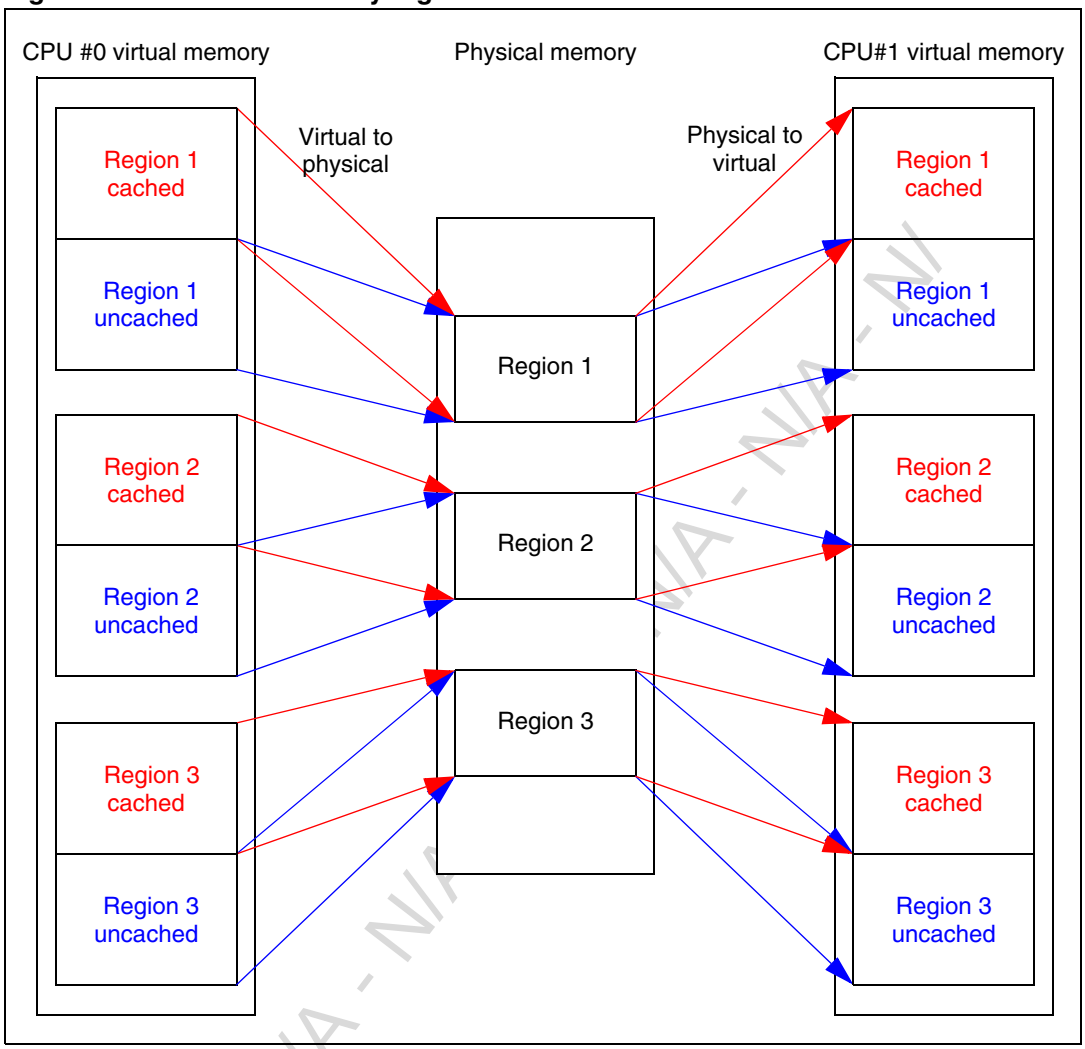
`ICS_region_add` registers and maps a memory region in both the local CPU and the remote CPUs. Given a virtual address, physical address, size, some memory attributes supplied as flags, and a set of CPUs defined in a bitmask, it sets up the region and also returns a region handle which can later be used to remove the region (by `ICS_region_remove`).

`ICS_region_virt2phys` translates a local CPU virtual address into a physical address, and `ICS_region_phys2virt` translates a physical address in a defined region into a local CPU virtual address by making use of the region tables created.

Once a memory region has been registered, addresses within that region can be used in calls to `ICS_msg_send` to allow a message to be sent using zero-copy techniques. This effectively transfers ownership of the buffer (passed to `ICS_msg_send`) to the receiving CPU which can then process it. It is the programmer's responsibility to manage the subsequent release of that memory.

The memory regions are shown in *Figure 10*.

Figure 10. Multicom memory regions



5.6 Name server

The ICS port model is based on named port handles which can be looked up using a central name server. This name service has also been exported as a primary API so that programmers can register other data objects and associate them with an ASCII string.

Table 17. Name server functions

Function name	Description
`ICS_nsrv_add`	Add a named object with the name server
`ICS_nsrv_remove`	Remove an object from the name server
`ICS_nsrv_lookup`	Look up a named object in the name server

Functions are provided to add, remove and look up names in the central name server. Lookups return the associated object handle if a match is found. Lookups can also be specified to block if a name is not found, in which case they will only return once that name is registered or a specific timeout period has expired. This facility can be used to build synchronization points between the ICS CPUs.

5.7 Dynamic module loading

ICS provides functions for dynamically loading and unloading code modules on a companion CPU. These are distinct from the basic functions described earlier for loading an initial ELF file into a CPU and starting it. The dynamic code loading functions assume that the ICS code is already running in the CPUs and are intended to add and start a new *load module* on the companion CPU at run-time. They are intended for applications where the code running in the companions is to be determined at run-time or to be modified and augmented as the application runs. The intention is that load modules can be used to modularize the software running on the ICS companion CPUs, avoiding large statically-linked firmware binaries having to be built for each CPU.

Load modules are expected to be relocatable ELF library files which can either be located in memory or loaded from a filing system.

These functions are built on the `rl_lib` library as provided with the ST40 and ST200 Micro Toolsets, using OS21. See the user manuals for these toolsets, in particular to understand how the hierarchy of load modules is defined.

Table 18. Dynamic code loading functions

Function name	Description
`ICS_dyn_load_file`	Load a dynamic ELF module from a local file
`ICS_dyn_load_image`	Load a dynamic ELF module from a memory image
`ICS_dyn_unload`	Unload a previously loaded dynamic ELF module

`ICS_dyn_load_file` and `ICS_dyn_load_image` load a dynamic ELF module, from a local file or from a memory image respectively, for a specified target CPU. They load and relocate the code for the target CPU, dynamically linking it against the parent module if one is specified. The functions return a module handle for use as a parent in subsequent load operations or for use by `ICS_dyn_unload`.

Once a dynamic module is loaded the ICS system will automatically call a `module_init` entry in the loaded module, if present. On unloading, a `module_term` entry will be called, if present.

5.8 CPU watchdog support

ICS supports a requirement to detect and recover from software failures on companion CPUs during execution, typically to be managed by the application on the master CPU. The scenario supported is when one of the companion CPUs stops operating and communicating as a result of a software failure. The recovery action required is typically for the master to reload and restart the companion CPU and then re-establish the ICS connections, while the rest of the ICS system continues to run.

Note: The `ICS_cpu_init` call takes a flag (`ICS_INIT_WATCHDOG`) which installs a system watchdog callback for each CPU present. This watchdog will be triggered whenever a CPU failure is detected and it will automatically call `ICS_cpu_disconnect`.

ICS also provides functions to install and manage a *watchdog* monitoring a set of CPUs for failure, and functions for *disconnecting* from a failed CPU and then *re-connecting* after it has been re-initialized. A watchdog is triggered whenever one of the monitored CPUs fails to update an internal counter within a fixed time period. This provides a fairly coarse-grained level of monitoring that will detect CPU crashes. However, it will not detect errors where a software bug is causing no useful work or progression to be made. Such levels of fault detection will be required to be implemented at a higher level, such as in the MME system.

Table 19. Failure management functions

Function name	Description
`ICS_watchdog_add`	Install a CPU watchdog handler
`ICS_watchdog_remove`	Remove a previously installed watchdog handler
`ICS_watchdog_reprime`	Re-prime a triggered watchdog callback
`ICS_cpu_connect`	Connect to a CPU allowing ICS communication after a disconnect
`ICS_cpu_disconnect`	Disconnect ICS communication from a CPU

`ICS_watchdog_add` installs a watchdog to monitor the operations of the set of CPUs defined by a CPU bitmask. It defines a callback function to be called when a CPU in the monitored set stops operating.

The callback function for a watchdog has the prototype:

```
void (*ICS_WATCHDOG_CALLBACK) (ICS_WATCHDOG handle,
ICS_VOID *param, ICS_UINT cpuNum)
```

The callback function is supplied with a parameter defined in the `watchdog_add` call, the handle of the relevant watchdog, and the CPU whose failure caused the watchdog to fire. Once the callback has been triggered for a particular CPU it will not fire again until `ICS_watchdog_reprime` has been called for that CPU.

Once failure of a CPU has been reported by a watchdog, the application should call `ICS_cpu_disconnect` to free up any resources associated with that CPU connection. The application should then proceed to re-load and restart the program running on that CPU, either using the ICS `ics_cpu_` functions or other libraries available for this purpose. Once

the CPU has been restarted, the application should call `ICS_cpu_connect` to re-establish communications with the re-started CPU. Then the watchdog for that CPU should be re-primed using `ICS_watchdog_reprime`.

5.9 Debug logging support

ICS provides a debugging facility where all text output from the running CPU cores is logged to a cyclic buffer. Functions are provided to dump out these logs on a per CPU basis, hence allowing diagnosis of issues when a multi-core debugger session is not possible.

Table 20.

Function name	Description
`ics_debug_flags`	Set the debug logging flags
`ics_debug_chan`	Set the debug logging output channel
`ICS_debug_dump`	Dump out the debug log

`ics_debug_flags` and `ics_debug_chan` are used to set up the debugging system prior to `ICS_cpu_init` being called. `ics_debug_flags` defines the subsystems of ICS for which debug logging is required. `ics_debug_chan` determines whether debugging is logged to `stdout`, `stderr`, or to a cyclic buffer in memory. `ICS_debug_dump` is called while ICS is running and dumps out all the messages logged to the cyclic buffer of that CPU.

6 Inter-core system (ICS) API

This chapter describes the ICS API in terms of its functions and macros.

The functions are listed in alphabetical order. Functions beginning with "ics_" are listed in *Section 6.1: ics_ function definitions* and functions beginning with "ICS_" are listed in *Section 6.2: ICS_ function definitions on page 143*. Macro definitions are listed in *Section 6.3: Macro definitions on page 194*.

*Note: The function ICS_cpu_init or ics_cpu_init **must** be called on the current CPU before any function beginning with ICS_ is called.*

6.1 ics_ function definitions

This section provides detailed descriptions of the ics_ functions.

ics_cpu_init — Initialize the ICS system on a CPU

Definition:

```
#include <ics.h>

ICS_ERROR ics_cpu_init (ICS_UINT cpuNum, ICS_VLONG cpuMask,
                        ICS_UINT flags)
```

Arguments:

cpuNum	The logical CPU number of calling CPU.
cpuMask	Bitmask of participating CPUs.
flags	Debug channel flag bits, see *Table 21*.

Returns:

ICS_SUCCESS	Successfully initialized.

Errors:

ICS_ALREADY_INITIALISED	ICS is already initialized.
ICS_ENOMEM	Failed memory/resource allocation.
ICS_SYSTEM_ERROR	A system error occurred.
ICS_SYSTEM_TIMEOUT	Failed to synchronize with other CPUs.

Context: Callable from task context only.

Definition: ics_cpu_init() is provided primarily for debugging purposes where a subset of CPUs is being tested. For example, just a pair of CPUs could be specified in the bitmask. It is recommended that the function *ICS_cpu_init on page 156* is adopted as the usual method of initializing the ICS system,

ics_cpu_init() can be called to start the ICS system on each participating CPU. If used, it must be called before any of the other ICS_ functions are called. It should be called from a task context and only be called once per CPU.

cpuNum is the logical CPU number of the calling CPU. ics_cpu_self() can be used to determine this value.

`cpuMask` is a bitmask of all the required CPUs, this can be a subset of all the CPUs on the SoC.

The set of valid debug channel flag bits are detailed in *Table 21*.

Table 21. ics_cpu_init flags

Init flag	Description
`ICS_INIT_CONNECT_ALL`	Connect to all CPUs in the bitmask during initialization
`ICS_INIT_WATCHDOG`	Enable the CPU watchdog for all CPUs in the bitmask

Setting the flag bit value `ICS_INIT_CONNECT_ALL` in the call to `ics_cpu_init()` causes the calling CPU to attempt to connect and synchronize with all the other CPUs which are present in the supplied CPU bitmask. It blocks until all the other CPUs have also called `ics_cpu_init()`. If one or more of the other CPU fails to call `ics_cpu_init()` then the operation fails after a pre-defined `timeout` period.

Setting the flag bit value `ICS_INIT_WATCHDOG` causes the ICS system to monitor all the CPUs in the bitmask. If any one of them fails, an automatic callback is triggered on the local CPU which, in turn, disconnects the failed CPU from further communication.

See also: *ICS_cpu_init*

ics_cpu_lookup Query the BSP/Registry for CPU number

Definition:

```
#include <ics.h>

int ics_cpu_lookup (const ICS_CHAR *cpuName)
```

Arguments:

`cpuName`	Name of CPU being queried.

Returns: A logical CPU number, if one is found.

Errors:

`-1`	CPU name not found.

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: `ics_cpu_lookup()` provides a mechanism for converting the symbollic names of the CPUs (for example “`audio`”, “`video`”) into their logical ICS CPU numbers.

An ICS system is made up of one or more logical CPUs. It is assumed that there is always a master CPU, whose logical CPU number is 0. It is also assumed that the ICS master CPU will be always booted before the others and that it will always be running.

`cpuName` should be a ’`\0`’ terminated ASCII string.

Examples of BSP/Registry tables are given in *Table 22* and *Table 23*.

Table 22. Example STi7200 (MB519) BSP/Registry

Logical CPU	CPU name	CPU type
0	"st40"	"st40"
1	"video0"	"st231"
2	"audio0"	"st231"
3	"video1"	"st231"
4	"audio1"	"st231"

Table 23. Example STi7141 (MB628) BSP/Registry

Logical CPU	CPU name	CPU type
0	"estb"	"st40"
1	"ecm"	"st40"
2	"video"	"st231"
3	"audio"	"st231"

See also: *ics_cpu_name*

ics_cpu_type.

ics_cpu_mask — Query the logical CPU bitmask

Definition:

```
#include <ics.h>

ICS_ULONG ics_cpu_mask (void)
```

Arguments: None

Returns: A logical ICS CPU bitmask.

Errors: None

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: Returns the logical CPU bitmask of the running system. Each set bit *n* representing that logical CPU number *n* is present. This information is determined by querying the BSP/Registry of the running system.

Normally this bitmask will be fulling populated, that is, for an *N* CPU system all bit positions from 0 to *N*-1 will be set. However, for debugging purposes it may be partially populated.

See also: *ics_cpu_lookup* (for details of the ICS logical CPU numbering system).

ics_cpu_name — Query the BSP/Registry for CPU name

Definition:

```
#include <ics.h>

const char *ics_cpu_name (ICS_UINT cpuNum)
```

Arguments:

`cpuNum`	Logical CPU number being queried.

Returns: BSP CPU name string.

Errors:

`NULL`	CPU not found.

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: `ics_cpu_name()` returns the CPU name string associated with the logical CPU number. Examples of BSP/Registry tables are given in *Table 22 on page 122* and *Table 23 on page 122*.

See also: *ics_cpu_lookup*

ics_cpu_type

ics_cpu_reset — Reset and stop CPU execution

Definition:

```
#include <ics.h>

ICS_ERROR ics_cpu_reset (ICS_UINT cpuNum,
                         ICS_UINT flags)
```

Arguments:

`cpuNum`	Logical CPU number being reset.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	CPU reset was issued successfully.

Errors:

`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: `ics_cpu_reset()` resets the logical CPU and stops execution. This is an asynchronous operation, and how soon the CPU stops executing code is system dependant.

It is an invalid operation to attempt to do this against the logical calling CPU number.

`cpuNum` should be a valid logical CPU number.

Currently no `flags` bits are defined and this parameter must be set to zero.

ics_cpu_self — Query the logical CPU number

Definition:

```
#include <ics.h>

ICS_INT  ics_cpu_self (void)
```

Arguments: None

Returns: The logical ICS CPU number of the calling CPU.

Errors: Returns -1 if there was an error.

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: Returns the logical CPU number of the calling CPU. This information is determined by querying the BSP/Registry of the running system.

An ICS system is made up of one or more logical CPUs numbered 0 to *N*-1, where *N* is the total number of CPUs in the system. It is assumed that there is always a master CPU, whose logical CPU number is 0.

See also: *ics_cpu_lookup* (for details of the ICS logical CPU numbering system).

ics_cpu_start — Start execution of a CPU

Definition:

```
#include <ics.h>

ICS_ERROR ics_cpu_start (ICS_OFFSET entryAddr,
                         ICS_UINT cpuNum,
                         ICS_UINT flags)
```

Arguments:

`entryAddr`	CPU execution start address.
`cpuNum`	Logical CPU number of CPU being started.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	CPU start was issued successfully.

Errors:

`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: `ics_cpu_start()` causes the logical CPU `cpuNum` to be started, with execution beginning at the supplied address. This is an asynchronous operation, and how soon the CPU starts executing code is system dependant .

It is an invalid operation to attempt to issue this call against the calling logical CPU number.

`entryAddr` should be a valid code start address for the target CPU. See `ics_load_elf_file()` and `ics_load_elf_image()`.

`cpuNum` should be the logical ICS CPU number of the CPU to be started.

Currently no `flags` bits are defined and this parameter must be set to zero.

ics_cpu_type — Query the BSP/Registry for CPU type

Definition:

```
#include <ics.h>

const char *ics_cpu_type  (ICS_UINT cpuNum)
```

Arguments:

`cpuNum`	Logical CPU number being queried.

Returns: BSP CPU type string.

Errors:

`NULL`	CPU not found.

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: `ics_cpu_type()` returns the CPU type string associated with the logical CPU number. Examples of BSP/Registry tables are given in *Table 22 on page 122* and *Table 23 on page 122*.

See also: *ics_cpu_lookup*

ics_cpu_name

ics_cpu_version — Query the ICS system version string

Definition:

```
#include <ics.h>

const ICS_CHAR * ics_cpu_version (void)
```

Arguments: None.

Returns: A pointer to the ICS version string.

Errors: None.

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: Return a pointer to the ICS version string.

This string takes the form:

major number`.`*minor number*`.`*patch number* `[:` *text*`]`

That is, a major, minor and release number, separated by decimal points, and optionally followed by a colon and a text string.

ics_debug_chan — Set the debug logging output channel

Definition:

```
#include <ics.h>

void ics_debug_chan (ICS_UINT flags)
```

Arguments:

`flags`	Bitmask of debug channel flags.

Returns: None

Errors: None

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: `ics_debug_chan()` sets the debug output channel of the ICS system. The ICS debug libraries are built with conditional debugging enabled for each subsystem. Each subsystem logs debug and error messages depending on the flags set by `ics_debug_flags()`. The output 'channel' for these debug message is controlled by calling this function. The set of valid debug channel flag bits are detailed in *Table 24*. These are encoded as bitmask values and hence combinations such as `ICS_DBG_STDOUT|ICS_DBG_LOG` are permitted.

Table 24. ICS debug channel flags

Channel flag	Description
`ICS_DBG_STDOUT`	Log all messages to console/STDOUT
`ICS_DBG_STDERR`	Log all message to console/STDERR
`ICS_DBG_LOG`	Log all messages to cyclic buffer

Note: The ICS library is built with the debug channel set to `ICS_DBG_LOG` *by default.*

ics_debug_flags — Set the debug logging flags

Definition:

```
#include <ics.h>

typedef enum
{
                ICS_DBG_ERR       = 0x0001,
                ICS_DBG_INIT      = 0x0002,
                ICS_DBG_CHN       = 0x0004,
                ICS_DBG_MAILBOX   = 0x0008,
                ICS_DBG_MSG       = 0x0010,
                ICS_DBG_ADMIN     = 0x0020,
                ICS_DBG_PORT      = 0x0040,
                ICS_DBG_NSRV      = 0x0100,
                ICS_DBG_WATCHDOG  = 0x0200,
                ICS_DBG_STATS     = 0x0400,
                ICS_DBG_HEAP      = 0x1000,
                ICS_DBG_REGION    = 0x2000,
                ICS_DBG_LOAD      = 0x4000,
                ICS_DBG_DYN       = 0x8000
} ICS_DBG_FLAGS;

void ics_debug_flags (ICS_UINT flags)
```

Arguments:

`flags`	Bitmask of debugging flags.

Returns: None

Errors: None

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Definition: `ics_debug_flags()` sets the debug logging level of the ICS system. The ICS debug libraries are built with conditional debugging enabled for each subsystem. In order to log the debug messages from a particular subsystem the corresponding bit the debug flags needs to be set by using this function. The valid set of debug flag bits are detailed in *Table 25*.

Table 25. ICS debug logging flags

Debug flag	Description
ICS_DBG_ERR	Log all error messages
ICS_DBG_INIT	Log initialization actions
ICS_DBG_CHN	Log channel actions
ICS_DBG_MAILBOX	Log mailbox actions
ICS_DBG_MSG	Log message actions
ICD_DBG_ADMIN	Log administration actions
ICS_DBG_PORT	Log port actions
ICS_DBG_NSRV	Log name server actions
ICS_DBG_WATCHDOG	Log watchdog actions

Table 25. ICS debug logging flags (continued)

Debug flag	Description
ICS_DBG_HEAP	Log heap actions
ICS_DBG_REGION	Log region actions
ICS_DBG_LOAD	Log load actions
ICS_DBG_DYN	Log dynamic loader actions

ics_err_str — Returns an ICS error string

Definition:

```
#include <ics.h>

const ICS_CHAR * ics_err_str (ICS_ERROR err)
```

Arguments:

`err`	An ICS_ERROR error code.

Returns: A pointer to the corresponding ICS error string.

Errors: None

Context: Callable from task and interrupt context. Can be called before `ICS_cpu_init()`.

Description: `ics_err_str()` returns a pointer to the corresponding ICS error string based on the supplied `err` code. This function is a useful way to display/log a text string describing the ICS error code when an error occurs.

`err` should be a valid ICS error code as returned by an ICS API function.

ics_heap_alloc — Allocate a buffer from an ICS memory heap

Definition:

```
#include <ics.h>

ICS_VOID  *ics_heap_alloc (ICS_HEAP heap,
                ICS_SIZE size,
                ICS_MEM_FLAGS mflags)
```

Arguments:

`heap`	Heap from which to allocate the buffer.
`size`	Size of the buffer in bytes.
`mflags`	Memory region attributes.

Returns: Non `NULL` address of the buffer allocated.

Errors: `NULL` is returned on error.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_heap_alloc()` allocates a buffer of size bytes from a heap previously created with `ics_heap_create()`. When all resources from the heap are exhausted, `NULL` will be returned. Both cached and uncached translations of the heap buffers can be obtained by specifying the relevant flag in the `mflags` parameter.

To avoid cache coherency issue between the CPUs, all buffers allocated will be `ICS_CACHELINE_SIZE` aligned.

`size` is the size in bytes of the buffer to be allocated. The actual size allocated will be rounded up by the heap allocator for cache coherency and alignment constraints.

`mflags` specifies the memory attributes of the buffer being requested. Valid values are `ICS_CACHED` and `ICS_UNCACHED`.

ics_heap_base — Query ICS heap for virtual base

Definition:

```
#include <ics.h>

ICS_VOID  *ics_heap_base (ICS_HEAP heap, ICS_MEM_FLAGS mflags)
```

Arguments:

`heap`	Heap handle.
`mflags`	Request cached or uncached mapping.

Returns: Associated heap parameter.

Errors:

`NULL`	Virtual base address of heap not found.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_heap_base()` returns the virtual base address of the supplied ICS heap.

This is one of three functions that allow the caller to query the base and size of an ICS heap, these calls are useful for presenting to the ICS region management code. Doing this allows local heaps to be mapped into the remote CPUs so that it can be used for zero-copy message passing and buffer sharing.

`heap` should be a valid heap handle as allocated with `ICS_heap_create()`.

Valid `mflags` values are `ICS_CACHED` and `ICS_UNCACHED`.

See also: *ics_heap_pbase*

ics_heap_size

ics_heap_create — Create an ICS heap

Definition:

```
#include <ics.h>
ICS_ERROR  ics_heap_create (ICS_VOID *heapBase,
                ICS_SIZE heapSize,
                ICS_UINT flags,
                ICS_HEAP *heapp)
```

Arguments:

heapBase	Optional base address of heap.
heapSize	Size in bytes of heap being created.
flags	Various flag bits which affect behavior.
heapp	Heap handle pointer used to return allocated handle.

Returns:

ICS_SUCCESS	CPU start was issued successfully.
heapp	Allocated heap handle.

Errors:

ICS_INVALID_ARGUMENT	An invalid argument was supplied.
ICS_ENOMEM	Failed memory/resource allocation.
ICS_SYSTEM_ERROR	A system error occurred.

Context: Callable from task context only. Can be called before ICS_cpu_init().

Description: ics_heap_create() creates a heap from a contiguous physical memory region. This heap can then be used to allocate and free buffers.

heapBase can be supplied by the caller if a physically contiguous memory region has already been allocated, otherwise NULL should be supplied. If a memory region address is supplied, then it must be ICS_PAGE_SIZE aligned.

heapSize is the size of the heap to be created in bytes. This value must be a whole multiple of ICS_PAGE_SIZE.

Currently no flags bits are defined and this parameter must be set to zero.

heapp is a pointer to an ICS_HEAP descriptor in which the allocated heap handle is returned.

ics_heap_destroy — Destroy an ICS heap

Definition:

```
#include <ics.h>
ICS_ERROR  ics_heap_destroy (ICS_HEAP heap,
                             ICS_UINT flags)
```

Arguments:

`heap`	Heap handle.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully destroyed heap.

Errors:

`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_heap_destroy()` should be called to destroy a heap previously created with `ics_heap_create()`. It will release all resources associated with the heap. All allocated buffers must have been returned to the heap before it is destroyed.

`heap` should be a valid heap handle.

Currently no `flags` bits are defined and this parameter must be set to zero.

ics_heap_free — Release a buffer back to an ICS heap

Definition:

```
#include <ics.h>

ICS_ERROR  ics_heap_free (ICS_HEAP heap,
                          ICS_VOID *buffer)
```

Arguments:

`heap`	Heap handle.
`buffer`	Buffer address.

Returns:

`ICS_SUCCESS`	Successfully freed buffer.

Errors:

`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_heap_free()` returns a previously allocated buffer to the heap. The buffer address supplied must be the exact one supplied by `ics_heap_alloc()`.

`heap` should be a valid ICS heap handle and be associated with the returned buffer address.

`buffer` should be a valid address, previously returned by `ICS_heap_alloc()`.

ics_heap_pbase — Query ICS heap for physical base

Definition:

```
#include <ics.h>

ICS_OFFSET ics_heap_pbase (ICS_HEAP heap)
```

Arguments:

`heap`	Heap handle.

Returns: Associated heap parameter.

Errors:

`ICS_BAD_OFFSET`	Physical base address of heap not found.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_heap_pbase()` returns the physical address base of the supplied ICS heap.

This is one of three functions that allow the caller to query the base and size of an ICS heap, these calls are useful for presenting to the ICS region management code. Doing this allows local heaps to be mapped into the remote CPUs so that it can be used for zero-copy message passing and buffer sharing.

`heap` should be a valid heap handle as allocated with `ICS_heap_create()`.

See also: *ics_heap_base*

ics_heap_size

ics_heap_size — Query ICS heap for size

Definition:

```
#include <ics.h>

ICS_SIZE ics_heap_size (ICS_HEAP heap)
```

Arguments:

`heap`	Heap handle.

Returns: Associated heap parameter.

Errors:

`0`	Size of heap not found.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_heap_size()` returns the full size of the supplied ICS heap.

This is one of three functions that allow the caller to query the base and size of an ICS heap, these calls are useful for presenting to the ICS region management code. Doing this allows local heaps to be mapped into the remote CPUs so that it can be used for zero-copy message passing and buffer sharing.

`heap` should be a valid heap handle as allocated with `ICS_heap_create()`.

See also: *ics_heap_base*

ics_heap_pbase

ics_load_elf_file — Load and unpack an ELF file

Definition:

```
#include <ics.h>
ICS_ERROR ics_load_elf_file (const ICS_CHAR *fname,
                             ICS_UINT flags,
                             ICS_LOAD *entryAddrp)
```

Arguments:

`fname`	Local OS filename of ELF file.
`flags`	Various flag bits which affect behavior.
`entryAddrp`	Returns loaded object ELF start/entry address.

Returns:

`ICS_SUCCESS`	Successfully loaded the ELF image.
`entryAddrp`	ELF start/entry address.

Errors:

`ICS_NAME_NOT_FOUND`	Filename was not found.
`ICS_INVALID_ARGUMENT`	An invalid argument or ELF image was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_load_elf_file()` is used to load an ELF file image on the master CPU and unpack it so that it can be executed on the Slave/Companion CPUs. On successful completion, the ELF start/entry address is returned to the caller in the `entryAddrp` argument.

`fname` should be a local OS filename from where the ELF image can be obtained.

Currently no `flags` bits are defined and this parameter must be set to zero.

`entryAddrp` should be a valid pointer to an `ICS_OFFSET` sized object.

ics_load_elf_image — Unpack an ELF memory image

Definition:

```
#include <ics.h>
ICS_ERROR  ics_load_elf_image (ICS_CHAR *image,
                               ICS_UINT flags,
                               ICS_OFFSET *entryAddrp)
```

Arguments:

`image`	Virtual memory address of ELF file image.
`flags`	Various flag bits which affect behavior.
`entryAddrp`	Returns loaded object ELF start/entry address.

Returns:

`ICS_SUCCESS`	Successfully unpacked the ELF image.
`entryAddrp`	ELF code start/entry address.

Errors:

`ICS_INVALID_ARGUMENT`	An invalid argument or ELF image was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only. Can be called before `ICS_cpu_init()`.

Description: `ics_load_elf_image()` is used to unpack an ELF memory image on the master CPU so that it can be executed on the Slave/Companion CPUs. On successful completion the ELF start/entry address is returned to the caller in the `entryAddrp` argument.

`image` should reference a complete ELF file image.

Currently no `flags` bits are defined and this parameter must be set to zero.

6.2 ICS_ function definitions

This section provides detailed descriptions of the ICS_ functions.

*Note: The function `ICS_cpu_init` or `ics_cpu_init` **must** be called on the current CPU before any function beginning with `ICS_` is called.*

ICS_channel_alloc — Allocate an ICS communications channel

Definition:

```
#include <ics.h>

typedef ICS_ERROR (*ICS_CHANNEL_CALLBACK) (ICS_CHANNEL channel,
                ICS_VOID *param,
                void *buffer)

ICS_ERROR ICS_channel_alloc (ICS_CHANNEL_CALLBACK callback,
                ICS_VOID *param,
                ICS_VOID *base,
                ICS_UINT nslots,
                ICS_UINT ssize,
                ICS_UINT flags,
                ICS_CHANNEL *channelp)
```

Arguments:

`callback`	Callback handler function to be associated with this channel.
`param`	Handle to be supplied to the callback function.
`base`	Optional base memory address of channel.
`nslots`	Number of slots in the channel FIFO.
`ssize`	Slot size in bytes, of each FIFO slot.
`flags`	Various flag bits which affect behavior.
`channelp`	Channel handle pointer used to return allocated handle.

Returns:

`ICS_SUCCESS`	Successfully allocated channel.
`channelp`	Contains allocated channel handle.

Errors:

`ICS_NOT_INITIALISED`	ICS is not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_channel_alloc()` allocates a uni-directional inter-cpu communication channel on the local CPU. Channels are formed as uni-directional, fixed length FIFOs. These channels can then be used to send arbitrary byte formatted data

between the CPUs. All data successfully inserted into a channel is guaranteed to be delivered to the target CPU in the order is was sent.

The channel API provides the lowest overhead communication system between the ICS CPUs. As such it provides a very raw interface, with no protocol being added by ICS. Programmers can use either interrupt or polling techniques to receive data through the channels.

`callback` is a pointer to function of type `ICS_CHANNEL_CALLBACK` which will be invoked each time a new entry arrives in the FIFO. It will be invoked in interrupt context supplying the `param` and `buffer` pointer to a FIFO entry . If callbacks are not required, then these parameters can be set to NULL. In this case the FIFO messages can only be retrieved using `ICS_channel_recv()`.

`base` is an optional pointer to an area of memory that can be used as the inter-CPU FIFO channel. As such it must be at least (`nslots*ssize`) bytes in size and the address must also be `ICS_PAGE_ALIGNED` aligned. Supplying a NULL value for this parameter will cause the call to allocate the memory itself.

`nslots` is the number of FIFO slots required. It must be greater than one and also a power of 2 in size.

`ssize` is the size of each FIFO slot in bytes, it must be a whole multiple of `ICS_CACHELINE_SIZE`.

Currently no `flags` bits are defined and this parameter must be set to zero.

`channelp` should be a pointer to an `ICS_CHANNEL` object in which the allocated channel handle will be returned on successful completion.

ICS_CHANNEL_CALLBACK — Channel callback function

Definition:

```
#include <ics.h>

typedef ICS_ERROR (*ICS_CHANNEL_CALLBACK) (ICS_CHANNEL channel,
                ICS_VOID *param,
                void *buffer)
```

Arguments:

`channel`	Handle of associated channel.
`param`	Parameter as supplied with the allocation function.
`buffer`	Channel buffer pointer.

Returns:

`ICS_SUCCESS`	Successfully consumed FIFO buffer.

Errors:

`ICS_FULL`	Failed to consume FIFO buffer.

Context: Called from interrupt context only

Description: This is the channel callback function that is invoked for each entry that is inserted into the channel FIFO. It will be called using the same `param` argument as supplied during `ICS_channel_alloc()`.

The `buffer` pointer will point to an area of memory of up to `ssize` bytes, as specified during the `ICS_channel_alloc()` call.

No protocol is added by the ICS channel interface, so the actual buffer size will need to be determined by the programmer.

Normally the callback function will copy and then consume the FIFO entry by calling `ICS_channel_release()`. In this case it should return `ICS_SUCCESS`.

However, if for some reason it is not possible to consume this entry, then `ICS_FULL` should be returned. This will cause the FIFO buffer to be left at the head of the FIFO (and to block it). No further callbacks will be generated on this channel until it is subsequently unblocked by a call to `ICS_channel_unblock()`.

ICS_channel_close — Close a send channel

Definition:

```
#include <ics.h>
ICS_ERROR ICS_channel_close (ICS_CHANNEL_SEND schannel)
```

Arguments:

`schannel`	Send channel handle.

Returns:

`ICS_SUCCESS`	Successfully closed the send channel.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_HANDLE_INVALID`	An invalid send channel handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_channel_close()` closes a send channel that was previously opened using `ICS_channel_open()`.

Any channel sends currently in progress will still be delivered to the receiving channel.

`schannel` should be a valid send channel handle.

ICS_channel_free — Free an ICS channel

Definition:

```
#include <ics.h>

ICS_ERROR ICS_channel_free (ICS_CHANNEL channel,
                            ICS_UINT flags);
```

Arguments:

`channel`	Channel handle.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully freed channel.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid channel handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_channel_free()` frees a local CPU channel previously allocated with `ICS_channel_alloc()`.Freeing a channel will cause all resources associated with it to be freed and also any tasks blocked on the channel to be awoken. Any unprocessed entries in the FIFO will be ignored.

`channel` should be a valid channel handle

Currently no `flags` bits are defined and this parameter must be set to zero.

ICS_channel_open — Open a send channel for communication

Definition:

```
#include <ics.h>

ICS_ERROR ICS_channel_open (ICS_CHANNEL channel,
                            ICS_UINT flags,
                            ICS_CHANNEL_SEND *schannelp)
```

Arguments:

`channel`	Channel handle.
`flags`	Various flag bits which affect behavior.
`schannelp`	Send channel handle pointer.

Returns:

`ICS_SUCCESS`	Successfully opened the send channel.
`schannelp`	Contains allocated send channel handle.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid channel handle was supplied.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: In order to send data using a CPU channel, it must first be opened by calling `ICS_channel_open()`. This will allocate an associated send channel handle which is returned in `schannelp`.

The channel handle parameter can either be one allocated locally with `ICS_channel_alloc()` or one supplied from a different CPU by looking it up (for example, in the name server) or being supplied the opaque channel handle through another mechanism.

It is an error to attempt to open multiple send channels to a given channel.

`channel` should be a valid channel handle as allocated by `ICS_channel_alloc()` either on the the local CPU or on a remote one.

Currently no `flags` bits are defined and this parameter must be set to zero.

`schannelp` should be a valid pointer to an `ICS_CHANNEL_SEND` object.

ICS_channel_recv Blocking call to receive a buffer from an ICS channel

Definition:

```
#include <ics.h>

ICS_ERROR ICS_channel_recv (ICS_CHANNEL channel,
                            ICS_VOID **bufferp,
                            ICS_LONG timeout)
```

Arguments:

`channel`	Channel handle.
`bufferp`	Pointer to a buffer pointer.
`timeout`	Time in milliseconds to block waiting for a buffer.

Returns:

`ICS_SUCCESS`	Successfully received a buffer.
`bufferp`	Buffer pointer returned.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid channel handle was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Received timed out.

Context: Callable from task context only.

Description: `ICS_channel_recv()` blocks on the supplied channel handle awaiting a new FIFO entry arrival. The channel handle must be a valid local channel as created with `ICS_channel_alloc()`.

`bufferp` should be a pointer to a unique ICS_VOID pointer used to return the address of the FIFO buffer.

`timeout` is the amount of time in milliseconds to block, before aborting and returning `ICS_SYSTEM_TIMEOUT`.

On successful completion the supplied `bufferp` pointer will have been updated with the new FIFO entry location. This should then be processed by the caller, before being later released with `ICS_channel_release()`.

No protocol is used within the channel communication system, so it is the responsibility of the programmer to determine how much data is contained within each supplied FIFO entry, but obviously it cannot exceed the FIFO slot size that was used to create the channel.

ICS_channel_release — Release an ICS channel FIFO buffer

Definition:

```
#include <ics.h>

ICS_ERROR ICS_channel_release (ICS_CHANNEL channel,
                               ICS_VOID *buffer)
```

Arguments:

`channel`	Channel handle.
`buf`	Buffer pointer.

Returns:

`ICS_SUCCESS`	Successfully released a buffer.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid channel handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task and interrupt context.

Description: `ICS_channel_release()` releases a channel buffer back to the FIFO. It should be called immediately after the caller has processed the FIFO data in order to maintain the in order nature of FIFO processing.

`channel` should be a valid channel handle.

`buffer` should be the last channel buffer address supplied by `ICS_channel_recv()` or the callback handler.

ICS_channel_send — Send a buffer using an ICS send channel

Definition:

```
#include <ics.h>

ICS_ERROR ICS_channel_send (ICS_CHANNEL_SEND schannel,
                            ICS_VOID *buffer,
                            ICS_SIZE size,
                            ICS_UINT flags)
```

Arguments:

`schannel`	Send channel handle.
`buffer`	Channel buffer pointer.
`size`	Size in bytes of data buffer.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully sent the buffer.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid send channel handle was supplied.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_FULL`	The channel FIFO was full.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task and interrupt context.

Description: `ICS_channel_send()` sends a data buffer using an open send channel by copying it into the channel FIFO. On successful return, it is guaranteed that the data will be received by the target CPU task. All buffers will be delivered in the order they were sent. The size of the data buffer must not exceed the size of the channel FIFO slots, as specified in the `ICS_channel_alloc()` call.

If the channel FIFO is full, then `ICS_FULL` will be returned. In this case it is the programmer's responsibility to re-issue the send once the flow control issue has been resolved.

`schannel` should be a valid, open send channel handle as returned by `ICS_channel_open()`.

`buffer` should be a valid virtual address of a data buffer.

`size` should be the size of the data buffer in bytes. It cannot exceed the size of the channel FIFO slots.

Currently no `flags` bits are defined and this parameter must be set to zero.

ICS_channel_unblock — Unblock blocked ICS channel

Definition:

```
#include <ics.h>

ICS_ERROR ICS_channel_unblock (ICS_CHANNEL channel)
```

Arguments:

`channel`	Channel handle.

Returns:

`ICS_SUCCESS`	Successfully unblocked the channel.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_HANDLE_INVALID`	An invalid channel handle was supplied
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task and interrupt context.

Description: `ICS_channel_unblock()` should be called to unblock a channel that became blocked due to a channel callback function returning `ICS_FULL`. In this case, a buffer would have been left at the head of the FIFO, blocking all further communications on that channel.

Until this function is called, no further callback events will be generated on the blocked channel.

Calling this function with the channel handle of a channel that is not blocked will have no effect.

ICS_cpu_connect — Connect to a CPU allowing ICS communication

Definition:

```
#include <ics.h>

ICS_ERROR ICS_cpu_connect (ICS_UINT cpuNum,
                           ICS_UINT flags,
                           ICS_LONG timeout)
```

Arguments:

`cpuNum`	Logical CPU number.
`flags`	Various flag bits which affect behavior.
`timeout`	Connection timeout period.

Returns:

`ICS_SUCCESS`	Successfully connected to CPU.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_SYSTEM_TIMEOUT`	Connection timed out.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_cpu_connect()` connects to a logical CPU so that it can be communicated with. Normally all logical CPUs are automatically connected during `ICS_cpu_init()`. But in the case of a CPU failure, they are disconnected by calling `ICS_cpu_disconnect()`. Once the failed CPU has been restarted, the programmer should call `ICS_cpu_connect()` to re-enable communications with that CPU.

`cpuNum` is the logical CPU number for which the connection is required.

Currently no `flags` bits are defined and this parameter must be set to zero.

ICS_cpu_disconnect — Disconnect ICS communication from a CPU

Definition:

```
#include <ics.h>

ICS_ERROR ICS_cpu_disconnect (ICS_UINT cpuNum,
                              ICS_UINT flags)
```

Arguments:

`cpuNum`	Logical CPU number.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully disconnected from CPU.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_cpu_disconnect()` disconnects from a logical CPU so that it can no longer be communicated with. This call also frees up all local and remote resources associated with that CPU connection. `ICS_cpu_disconnect()` should be called when a CPU fails or crashes, doing so will then enable a new connection to be made once the CPU has been restarted.

`cpuNum` is the logical CPU number for which the disconnection is required.

Setting the `ICS_CPU_DEAD` bit of `flags` causes the disconnect to avoid communicating with the failed CPU. This bit should be set in the case of disconnecting from a failed CPU.

Note: Setting the `ICS_INIT_WATCHDOG` *bit value in the call to* `ICS_cpu_init()` *or* `ics_cpu_init()` *enables an automatic callback when a CPU failure is detected. In the case of a failure,* `ICS_cpu_disconnect()` *is called automatically for the failed CPU.*

ICS_cpu_info — Query the ICS CPU configuration

Definition:

```
#include <ics.h>

ICS_ERROR ICS_cpu_info (ICS_UINT *cpuNump,
                        ICS_ULONG *cpuMaskp)
```

Arguments:

`cpuNump`	Return parameter for CPU number.
`cpuMaskp`	Return parameter for CPU mask.

Returns:

`ICS_SUCCESS`	Call was successful.

Errors:

`ICS_NOT_INITIALISED`	ICS is not initialized.
`ICS_INVALID_ARGUMENT`	Invalid arguments supplied.

Context: Callable from task and interrupt context.

Description: `ICS_cpu_info()` queries the CPU info of the currently running ICS system. On success, the ICS CPU number of the current CPU and a bitmask representing each CPU present, will be returned using the supplied parameters. In the returned bitmask, each set bit *n* represents that logical CPU number *n* is present.

`cpuNump` should be a valid pointer to an `ICS_UINT` sized object in which the logical CPU number will be returned.

`cpuMaskp` should be a valid pointer to an `ICS_ULONG` sized object in which the logical CPU bitmask will be returned.

ICS_cpu_init — Initialize the ICS system on a CPU

Definition:

```
#include <ics.h>

ICS_ERROR ICS_cpu_init (ICS_UINT flags)
```

Arguments:

`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully initialized.

Errors:

`ICS_ALREADY_INITIALISED`	ICS is already initialized.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to synchronize with other CPUs.

Context: Callable from task context only.

Definition: `ICS_cpu_init()` is the recommended function for initializing the ICS and should be used to start the ICS system on each participating CPU. It must be called before any of the other `ICS_` functions are called. It should be called from task context and only be called once per CPU.

`ICS_cpu_init()` when called with the `ICS_INIT_CONNECT_ALL` flag bit value, causes the calling CPU to attempt to connect and synchronize with all the other CPUs which are present in the CPU bitmask (as derived from the BSP). It will block until all the other CPUs have also called `ICS_cpu_init()`. If one or more of the other CPU fails to call `ICS_cpu_init()` then the operation will fail after a pre-defined `timeout` period.

For the valid set of `flags` bits please refer to the `ics_cpu_init()` function.

ICS_cpu_term — Terminate the ICS system on a CPU

Definition:

```
#include <ics.h>

void ICS_cpu_term (ICS_UINT flags)
```

Arguments:

`flags`	Various flag bits which affect behavior.

Returns: None.

Errors: None.

Context: Callable from task context only.

Description: `ICS_cpu_term()` will terminate the ICS system. It should be called when a CPU is being shutdown and no longer requires the ICS system. Calling it multiple times or when `ICS_cpu_init()` has not been previously called will have no effect.

This function will terminate all tasks and release all resources associated with the local ICS system. For example, open ports and allocated regions will be closed and removed.

Currently no `flags` bits are defined and this parameter must be set to zero.

ICS_debug_dump — Dump out the debug log

Definition:

```
#include <ics.h>

ICS_ERROR ICS_debug_dump (ICS_UINT cpuNum)
```

Arguments:

`cpuNum`	Logical CPU number.

Returns:

`ICS_SUCCESS`	Successfully dumped the log.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_SYSTEM_ERROR`	A system error occured.

Context: Callable from task context only.

Description: `ICS_debug_dump()` dumps out all the log messages for the logical CPU `cpuNum`. When linked against the debug ICS libraries each subsystem can be set to log messages by calling the `ics_debug_flags()` function. If the debug channel has also been set (using `ics_debug_chan()`) to enable logging to the cyclic buffer, then these messages will be displayed by calling this function.

ICS_dyn_load_file — Load a dynamic ELF module from a local file

Definition:

```
#include <ics.h>

ICS_ERROR ICS_dyn_load_file (ICS_UINT cpuNum,
                             ICS_CHAR *fname,
                             ICS_UINT flags,
                             ICS_DYN parent,
                             ICS_DYN *handlep)
```

Arguments:

`cpuNum`	Target CPU number.
`fname`	Local OS filename of the dynamic ELF module.
`flags`	Various flag bits which affect behavior.
`parent`	Dynamic object handle of parent module.
`handlep`	Returns dynamic module handle.

Returns:

`ICS_SUCCESS`	Successfully loaded the dynamic module.
`handlep`	Allocated dynamic module handle.

Errors:

`ICS_NAME_NOT_FOUND`	Filename not found.
`ICS_INVALID_ARGUMENT`	An invalid argument or ELF file was supplied.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	A communications operation timed out.

Context: Callable from task context only.

Description: `ICS_dyn_load_file()` is used to load relocatable ELF dynamic modules files on the calling CPU and relocate them so that they can be executed on the target CPU. On successful completion, a dynamic module handle is returned to the caller using the `handlep` argument.

The ICS dynamic module system is layered on top of the 'rl' library provided by the ST Micro Toolset. Much more detail can be found in the relevant Toolset manuals.

`cpuNum` is the logical target CPU of where the dynamic module should be loaded.

`fname` should be a local OS filename from where the ELF dynamic module image can be obtained.

Currently no `flags` bits are defined and this parameter must be set to zero.

`parent` is the `ICS_DYN` handle of the parent module of the one being loaded. If no parent exists, then it can be supplied as a zero handle. If a non-zero parent handle is supplied then the new module will be linked against that module in the target CPU, providing symbol inheritance from the parent module.

`handlep` should be a non-NULL pointer to an ICS_DYN sized handle, which is used to return the allocated dynamic module handle.

ICS_dyn_load_image Load a dynamic ELF module from a memory image

Definition:

```
#include <ics.h>

ICS_ERROR ICS_dyn_load_image (ICS_UINT cpuNum,
                ICS_CHAR *image,
                ICS_SIZE imageSize,
                ICS_UINT flags,
                ICS_DYN parent,
                ICS_DYN *handlep)
```

Arguments:

`cpuNum`	Target CPU number.
`image`	Base address of the dynamic ELF module.
`imageSize`	Size of the dynamic ELF module image.
`flags`	Various flag bits which affect behavior.
`parent`	Handle of parent dynamic module.
`handlep`	Returns dynamic module handle.

Returns:

`ICS_SUCCESS`	Successfully loaded the dynamic module.
`handlep`	Allocated dynamic module handle on success.

Errors:

`ICS_INVALID_ARGUMENT`	An invalid argument or ELF file was supplied.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	A communications operation timed out.

Context: Callable from task context only.

Description: `ICS_dyn_load_image()` is used to take a relocatable ELF dynamic modules image on the calling CPU and relocate it so that it can be executed on the target CPU. On successful completion a dynamic module handle is returned to the caller in the `handlep` argument.

The ICS dynamic module system is layered on top of the 'rl' library provided by the ST Micro Toolsets. Much more detail can be found in the relevant Toolset manuals.

As a dynamic module is loaded into the target CPU, the ICS system will automatically call a function named `module_init()` if one is present in the loaded module.

`cpuNum` is the logical target CPU number of where the dynamic module should be loaded.

image should be a local memory address where the ELF dynamic module image can be found.

`imageSize` should be size of the ELF dynamic module image in bytes.

Currently no `flags` bits are defined and this parameter must be set to zero.

`parent` is the `ICS_DYN` handle of the parent module of the one being loaded. If no parent exists, then it can be supplied as a zero handle. If a non-zero parent handle is supplied then the new module will be linked against that module in the target CPU, providing symbol inheritance from the parent module. For further details on symbol inheritance see the 'rl' toolkit manual.

`handlep` should be a non-`NULL` pointer to an `ICS_DYN` sized object, which is used to return the allocated dynamic module handle.

ICS_dyn_unload — Unload a previously loaded dynamic ELF module

Definition:

```
#include <ics.h>

ICS_ERROR ICS_dyn_unload (ICS_DYN handle)
```

Arguments:

`handle`	Handle of the dynamic module to be unloaded.

Returns:

`ICS_SUCCESS`	Successfully unloaded the dynamic module.

Errors:

`ICS_HANDLE_INVALID`	An invalid handle was supplied.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	A communications operation timed out.

Context: Callable from task context only.

Description: `ICS_dyn_unload()` unloads a dynamic ELF module which was previously loaded using `ICS_dyn_load_file()` or `ICS_dyn_load_image()`.

It will unload the dynamic module in the original target CPU making use of the 'rl' toolkit system.

As a dynamic module is unloaded from the target CPU, the ICS system will automatically call a function named `module_term()` if one is present in the loaded module.

`handle` should be a valid dynamic module handle.

ICS_msg_cancel — Cancel an asynchronous port receive

Definition:

```
#include <ics.h>

ICS_ERROR ICS_msg_cancel (ICS_MSG_EVENT event)
```

Arguments:

`event`	Message event handle to be cancelled.

Returns:

`ICS_SUCCESS`	Cancel completed successfully.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_HANDLE_INVALID`	An invalid event handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_msg_cancel()` cancels an asynchronous receive request as posted with `ICS_msg_post()`. On successful return the message event handle will have been released and cannot be used again.

`ICS_msg_cancel()` must not be called on message events which are currently being blocked on in a call to `ICS_msg_wait()`.

If the associated port is closed with `ICS_port_free()` whilst there are still outstanding posted receives then they will all be automatically released. Calling `ICS_msg_cancel()` on such handles will result in `ICS_HANDLE_INVALID` being returned.

`event` should be a valid message event handle.

ICS_MSG_DESC

Port message descriptor

Definition:

```
#include <ics.h>

typedef struct ics_msg_desc
{
  ICS_OFFSET        data;
  ICS_SIZE          size;
  ICS_MEM_FLAGS     mflags;
  ICS_UINT          srcCpu;
  ICS_CHAR          payload[ICS_MSG_INLINE_DATA];
} ICS_MSG_DESC;
```

Members:

`data`	Address of the message data buffer.
`size`	Size in bytes of the receive message.
`mflags`	Memory attribute flags of the data buffer.
`srcCpu`	CPU number of the source CPU.
`payload`	Area for `ICS_INLINE` data.

Description: This is the message descriptor posted and completed by `ICS_msg_recv()` and `ICS_msg_post()`.

On successful completion all the member fields will have been completed by the ICS system.

`data` will be a valid address of the corresponding data buffer. In the case of `ICS_INLINE`, `ICS_CACHED` and `ICS_UNCACHED` messages this will be a corresponding virtual address pointer. For `ICS_PHYSICAL` messages it will be a physical memory address.

`size` will be the size of the received message in bytes.

`mflags` will be set according to the message data buffer memory attributes. Valid flags are `ICS_INLINE`, `ICS_CACHED`, `ICS_UNCACHED` and `ICS_PHYSICAL`.

`srcCpu` will be set to the ICS logical CPU number of the sending CPU.

`payload` will contain the inline data if `ICS_INLINE` is set in `mflags`.

ICS_msg_post — Post an asynchronous receive on a port

Definition:

```
#include <ics.h>

typedef struct ics_msg_desc
{
  ICS_OFFSET        data;
  ICS_SIZE          size;
  ICS_MEM_FLAGS     mflags;
  ICS_UINT          srcCpu;
  ICS_CHAR          payload[ICS_MSG_INLINE_DATA];
} ICS_MSG_DESC;

ICS_ERROR ICS_msg_post (ICS_PORT port,
              ICS_MSG_DESC *rdesc,
              ICS_MSG_EVENT *eventp)
```

Arguments:

`port`	Port handle.
`rdesc`	Receive descriptor pointer.
`eventp`	Pointer to an `ICS_MSG_EVENT` handle.

Returns:

`ICS_SUCCESS`	Successfully posted a receive.
`eventp`	The associated message event handle on success.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid port handle was supplied.
`ICS_PORT_CLOSED`	Port has been closed.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_msg_post()` posts an asynchronous (nonblocking) receive to the supplied `port` handle to match new message arrivals. Messages are received strictly in the order they were sent to the port and asynchronous receives are processed in order too.

On successful completion the supplied `rdesc` descriptor will have been posted against the requested port. This receive descriptor will be associated with the returned handle in `eventp`. This handle can then be used in calls to `ICS_msg_test()` and `ICS_msg_wait()` to test for or block for message arrival. Until `ICS_msg_wait()` has been called successfully on the returned `eventp` handle, the memory associated with the `rdesc` parameter must not be reused or freed.

For asynchronous posted receives, new incoming messages will always be matched to the posted receive descriptors in the order that the receives were issued. This also holds true if a blocking `ICS_msg_recv()` is subsequently issued against the same port.

If the port associated with the asynchronous receive is closed by `ICS_port_free()` then all outstanding posted receives will be signalled and completed with an `ICS_PORT_CLOSED` error.

`port` should be a valid local port as created with `ICS_port_alloc()`.

`rdesc` should be a pointer to a unique `ICS_MSG_DESC` sized region of memory.

`eventp` should be a pointer to an `ICS_MSG_EVENT` object.

ICS_msg_recv Blocking call to receive a message on an ICS port

Definition:

```
#include <ics.h>

typedef struct ics_msg_desc
{
  ICS_OFFSET        data;
  ICS_SIZE          size;
  ICS_MEM_FLAGS     mflags;
  ICS_UINT         srcCpu;
  ICS_CHAR          payload[ICS_MSG_INLINE_DATA];
} ICS_MSG_DESC;

ICS_ERROR ICS_msg_recv (ICS_PORT port,
              ICS_MSG_DESC *rdesc,
              ICS_LONG timeout)
```

Arguments:

`port`	Port handle.
`rdesc`	Receive descriptor pointer.
`timeout`	Time in milliseconds to block waiting for a message.

Returns:

`ICS_SUCCESS`	Successfully received a message.
`rdesc`	Updated message descriptor on success.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid port handle was supplied.
`ICS_PORT_CLOSED`	Port has been closed.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Receive timed out.

Context: Callable from task context only

Description: `ICS_msg_recv()` blocks on the supplied `port` handle awaiting a new message arrival. Messages are received in strictly the order they were sent to the port.

`port` should be a valid local open port as created with `ICS_port_alloc()`.

`rdesc` should be a pointer to a unique `ICS_MSG_DESC` sized region of memory.

`timeout` is the amount of time in milliseconds before the blocking wait should abort and return `ICS_SYSTEM_TIMEOUT`.

On successful completion the supplied `rdesc` descriptor will have been updated with all the new message information and any inline data. The `rdesc mflags` member indicates the data attributes. For `ICS_INLINE` messages the data will be available in the `payload` buffer and the `rdesc data` member will reflect this fact. For

`ICS_CACHED` and `ICS_UNCACHED` messages the corresponding virtual address mapping (that is, cached or uncached) will be supplied in the `data` member. For `ICS_PHYSICAL` messages the physical address of the sender's original data buffer will presented in the `rdesc data` member.

ICS_msg_send — Send a message buffer to a port

Definition:

```
#include <ics.h>

ICS_ERROR ICS_msg_send (ICS_PORT port,
                        ICS_VOID *buffer,
                        ICS_MEM_FLAGS mflags,
                        ICS_SIZE size,
                        ICS_UINT flags)
```

Arguments:

`port`	Port handle.
`buffer`	Source data buffer.
`mflags`	Destination data buffer memory attributes.
`size`	Source data buffer size.
`flags`	ICS message and send flag bits, see *Table 26*.

Returns:

`ICS_SUCCESS`	Successfully sent message.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid port handle was supplied.
`ICS_PORT_CLOSED`	Port has been closed.
`ICS_NOT_CONNECTED`	Target CPU is not connected.
`ICS_FULL`	An inter-CPU FIFO was full.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_msg_send()` sends a message to the target port handle. This port handle can either be a local port as created with `ICS_port_alloc()` or a remote port as discovered with `ICS_port_lookup()`.

Messages are always delivered to the target port in the same order they were sent. On successful return, it is guaranteed that the message will be delivered to the target port.

The message data should be presented as a virtual address in the `buffer` argument with its size in bytes being specified by the `size` parameter.

The `mflags` parameter allows the caller to control how the data is transferred and how it is presented to the target receiver. Setting `mflags` to `ICS_INLINE` causes the data to be copied on send into a system buffer area and subsequently copied to the target receiver as part of the `ICS_MSG_DESC` descriptor. Such inline message data cannot exceed `ICS_MSG_INLINE_DATA` bytes in size.

Setting the `mflags` parameters to either `ICS_CACHED` or `ICS_UNCACHED` will cause the data to be transferred using zero copy techniques; where the buffer address must have come from a memory region which has been previously registered with `ICS_region_add()`. In this case the target receiver will be presented with the appropriate virtual address mapping of that physical memory buffer (that is, cached or uncached).

Setting the flags parameters to `ICS_PHYSICAL` will cause the physical address of the supplied virtual data buffer to be presented at the target port. In this case the data buffer does not need be have been pre-registered with `ICS_region_add()`.

Using the zero-copy message passing technique effectively transfers ownership of the physical data region to the target CPU. ICS does not take any further control in this respect and therefore it is the programmer's responsibility to ensure the memory is correctly managed and released on the sender.

`ICS_msg_send()` is an asynchronous, nonblocking operation, and on return it cannot be assumed that the target CPU has received the message. This means that, except in the case of `ICS_INLINE` messages, the supplied data buffer cannot be reused until some form of acknowledgement has been received from the target CPU.

The valid `flags` bits are defined in *Table 26*

Table 26. ICS message send flag

Channel flag	Description
`ICS_MSG_CONNECT`	Make a connection to target CPU if necessary.

Setting the flag bit value `ICS_MSG_CONNECT` causes the ICS system to attempt to make a connection to the target CPU if one if not already in place. If the CPU has failed, then this operation blocks and the call eventually returns an error after a predefined timeout period.

Note: It is advised that the `ICS_MSG_CONNECT` flag should be used whenever a new connection is being potentially established with the target CPU, especially when the `ICS_INIT_CONNECT_ALL` flag was not passed to `ICS_cpu_init()` or `ics_cpu_init()`. However, once communication has begun with the target CPU, then `ICS_MSG_CONNECT` should not be used, otherwise large timeouts will occur in the case of a CPU failure.

ICS_msg_test Test an asynchronous message receive event for completion

Definition:

```
#include <ics.h>

ICS_ERROR ICS_msg_test (ICS_MSG_EVENT event,
                        ICS_BOOL *readyp)
```

Arguments:

`event`	Message event handle.
`readyp`	Pointer to a boolean for returning event status.

Returns:

`ICS_SUCCESS`	Call completed successfully.
`readyp`	Boolean of event completion state.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid event handle was supplied.
`ICS_PORT_CLOSED`	Port has been closed.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Wait timed out.

Context: Callable from task context only.

Description: `ICS_msg_test()` polls the state of a message event and sets the `readyp` parameter based on whether the event is ready or not. If the event is indicated as being ready, then a subsequent call to `ICS_msg_wait()` will always complete without blocking.

Note: *`ICS_msg_test()` does not free the message event handle.*

`event` should be a valid message event handle as returned by `ICS_msg_post()`.

`readyp` should be a pointer to an `ICS_BOOL` object. It will be set to either `ICS_TRUE` or `ICS_FALSE` on completion.

See also: *ICS_msg_wait*

ICS_msg_wait Block an asynchronous message receive event for completion

Definition:

```
#include <ics.h>

ICS_ERROR ICS_msg_wait (ICS_MSG_EVENT event,
                        ICS_LONG timeout)
```

Arguments:

`event`	Message event handle.
`timeout`	Time in milliseconds to block waiting for the event.

Returns:

`ICS_SUCCESS`	Call completed successfully.
`readyp`	Boolean of event completion state.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid event handle was supplied.
`ICS_PORT_CLOSED`	Port has been closed.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Wait timed out.

Context: Callable from task context only.

Description: `ICS_msg_wait()` blocks on the supplied message `event` handle until the associated event completes or the timeout period expires. In all cases, the message event is released on completion of this call.

`event` should be a valid message event handle as returned by `ICS_msg_post()`.

`timeout` is the amount of time in milliseconds to block waiting for a message event to complete.

See also: *ICS_msg_test*

ICS_nsrv_add — Add a named object with the name server

Definition:

```
#include <ics.h>

ICS_ERROR ICS_nsrv_add (const ICS_CHAR *name,
                        ICS_VOID *data,
                        ICS_SIZE size,
                        ICS_UINT flags,
                        ICS_NSRV_HANDLE handlep)
```

Arguments:

`name`	Object name to be added.
`data`	Object data to be associated with the name.
`size`	Size of object data.
`flags`	Various flag bits which affect behavior.
`handlep`	Pointer to an `ICS_NSRV_HANDLE` handle.

Returns:

`ICS_SUCCESS`	Successfully registered named object.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_NAME_IN_USE`	`name` is aready in use.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only.

Description: `ICS_nsrv_add()` adds the supplied object data to the global name server associating it with the supplied `name` string. Duplicate names are allowed and in this case the nameserver returns matching object lookups, in a round-robin order.

`name` should be an ASCII '`\0`' terminated string, of length not exceeding `ICS_NSRV_MAX_NAME` characters (not including the '`\0`').

`data` should be a pointer to the object data to be associated with the name string in the name server.

`size` should be the size in bytes of the object data. It must not exceed `ICS_NSRV_MAX_DATA` in size.

Currently no `flags` bits are defined and this parameter must be set to zero.

`handlep` should be a non-`NULL` pointer to an `ICS_NSRV_HANDLE` sized object, in which the allocated nameserver handle is returned.

ICS_nsrv_lookup — Lookup a named object in the name server

Definition:

```
#include <ics.h>

ICS_ERROR ICS_nsrv_lookup (const ICS_CHAR *name,
                           ICS_UINT flags,
                           ICS_LONG timeout,
                           ICS_VOID *data,
                           ICS_SIZE *sizep)
```

Arguments:

`name`	Name string (maximum length `ICS_NSRV_MAX_NAME`).
`flags`	Various flag bits which affect behavior.
`timeout`	Time in milliseconds to block waiting for a response.
`data`	Buffer for discovered object data.
`sizep`	Used to return the discovered object data size.

Returns:

`ICS_SUCCESS`	Successfully discovered an object in the name server.
`data`	Buffer where object data is copied on success.
`sizep`	Discovered object data size.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_NAME_NOT_FOUND`	`name` was not found in the name server.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only

Description: `ICS_nsrv_lookup()` attempts to discover the named object in the global name server. It supports both blocking and non-blocking modes of operation. In the non-blocking mode, if the supplied name is not present in the name server, then `ICS_NAME_NOT_FOUND` will be returned. In the blocking mode, the call will block for a specified amount of time waiting for the name to be registered. Once the `timeout` period has expired and no response has arrived from the name server, then `ICS_SYSTEM_TIMEOUT` will be returned. Duplicate object names are allowed in the nameserver, in which case matching lookups are supplied the referenced objects, in a round-robin order.

`name` should be an ASCII '`\0`' terminated string of length not exceeding `ICS_NSRV_MAX_NAME` characters (not including the '`\0`').

`flags` can be set to `ICS_BLOCK` to cause the function to block until the named object is registered in the name server.

`timeout` is the number of milliseconds an `ICS_BLOCK` call should wait for a response. Predefined values of timeout can be used;

– `ICS_TIMEOUT_IMMEDIATE` (return immediately)

– `ICS_TIMEOUT_INFINITE` (never return)

`data` should be a non-NULL pointer to an area of memory of at least `ICS_NSRV_MAX_DATA` bytes in size.

`sizep` should be a non-NULL pointer to an `ICS_SIZE` sized object in which the size of the discovered object data will be returned.

ICS_nsrv_remove — Remove an object from the name server

Definition:

```
#include <ics.h>

ICS_ERROR ICS_nsrv_remove (ICS_NSRV_HANDLE handle,
                           ICS_UINT flags);
```

Arguments:

`handle`	Handle of object to be de-registered.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully de-registered the named object.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_HANDLE_INVALID`	An invalid object `handle` was supplied.
`ICS_NAME_NOT_FOUND`	The object is not present in the name server.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only

Description: `ICS_nsrv_remove()` removes a previously registered object from the global ICS name server. If the supplied object is not present in the name server then `ICS_NAME_NOT_FOUND` is returned.

`handle` should be a valid name server object handle as returned by `ICS_nsrv_add()`.

Currently no `flags` bits are defined and this parameter must be set to zero.

ICS_port_alloc — Allocate an ICS port

Definition:

```
#include <ics.h>

typedef ICS_ERROR (*ICS_PORT_CALLBACK) (ICS_PORT port,
                ICS_VOID *param,
                ICS_MSG_DESC *rdesc)

ICS_ERROR ICS_port_alloc (const ICS_CHAR *portName,
                ICS_PORT_CALLBACK callback,
                ICS_VOID *param,
                ICS_UINT ndesc,
                ICS_UINT flags,
                ICS_PORT *portp)
```

Arguments:

`portName`	Port name to be allocated or NULL.
`callback`	Callback function to be associated with this port.
`param`	Parameter to be supplied to the callback function.
`ndesc`	Depth of message queue for this port.
`flags`	Various flag bits which affect behavior.
`portp`	Port handle pointer used to return allocated handle.

Returns:

`ICS_SUCCESS`	Successfully allocated port.
`portp`	Allocated port handle.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only

Description: `ICS_port_alloc()` allocates an ICS port on the local CPU. Ports can either be anonymous or named. All named ports are registered with the global name server from where they can be discovered and communicated with by all participating CPUs. If the supplied `portName` is already present in the name server, then multiple port handles with the same name will be registered and returned in a round-robin order on lookup.

`portName` is an optional ASCII `'\0'` terminated string of length not exceeding `ICS_PORT_MAX_NAME` characters (not including the `'\0'`).

`portName` can be set to NULL to indicate that this is a local anonymous port whose name does not need to be registered with the global name server.

`callback` is a pointer to function of type `ICS_PORT_CALLBACK` which will be invoked each time a new message arrives at the port. It will be invoked in interrupt context supplying the associated port handle, `param` and an `ICS_MSG_DESC` message descriptor pointer. If callbacks are not required then these parameters can be set to NULL.

`ndesc` determines the depth of the message queue associated with this port. It must be a power of 2 in size. Messages are stored on this queue in FIFO order until received by calling `ICS_msg_recv()` or `ICS_msg_post()`. A value of zero for this parameter is also allowed, in which case messages will be held in the inter-CPU FIFOs until a corresponding message receive is posted.

Note: Setting the port message queue depth to zero should be used with caution as it could block all other messages in the inter-CPU FIFO.

Currently no `flags` bits are defined and this parameter must be set to zero.

`portp` should be a pointer to an `ICS_PORT` object in which the allocated port handle will be returned on successful completion.

ICS_PORT_CALLBACK — Port callback function

Definition:

```
#include <ics.h>

typedef ICS_ERROR (*ICS_PORT_CALLBACK) (ICS_PORT port,
                ICS_VOID *param,
                ICS_MSG_DESC *rdesc)
```

Arguments:

`port`	The associated port handle.
`param`	`param` argument as supplied to the port create function.
`rdesc`	Receive descriptor pointer.

Returns:

`ICS_SUCCESS`	Successfully processed message.

Errors:

`ICS_FULL`	Failed to consume message.

Context: Called from interrupt context only

Description: This is the port callback function that is invoked for each message that arrives at the port. It will be called using the same `param` argument as supplied during `ICS_port_alloc()`.

The rdesc pointer will refer to a completed `ICS_MSG_DESC` descriptor. Normally the callback function should process this incoming rdesc and return `ICS_SUCCESS` which will cause the message to be consumed. If however, it cannot be consumed at this point in time, then `ICS_FULL` should be returned which will cause the message to be held on the per port message queues. It can then be later retrieved by calling `ICS_msg_recv()` or `ICS_msg_post()`.

In order to preserve message ordering, once the callback function has returned `ICS_FULL`, no further callbacks will be generated on that port. Callbacks will only be enabled once the associated port message queue has been emptied.

ICS_port_cpu — Return logical CPU number associated with port

Definition:

```
#include <ics.h>

ICS_EXPORT ICS_ERROR ICS_port_cpu (ICS_PORT port,
                ICS_UNIT *cpuNump)
```

Arguments:

`port`	The associated port handle.
`cpuNump`	Return logical CPU number.

Returns:

`ICS_SUCCESS`	Successfully determined port logical CPU number.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid port handle was supplied.

Context: Called from task and interrupt context

Description: `ICS_port_cpu()` can be used to return the logical CPU number associated with the supplied port handle. If the port handle is valid, then `ICS_SUCCESS` is returned and the associated logical CPU number is returned.

`port` should be a valid port handle.

`cpuNump` should be a pointer to an `ICS_UINT` sized object in which the associated CPU number is returned on success.

ICS_port_free — Free and close an ICS port

Definition:

```
#include <ics.h>

ICS_ERROR ICS_port_free (ICS_PORT port,
                         ICS_UINT flags)
```

Arguments:

`port`	Port handle.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully freed port.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_PORT_CLOSED`	Port is already closed.
`ICS_HANDLE_INVALID`	An invalid port handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only.

Description: `ICS_port_free()` will free off and close a port previously created with `ICS_port_alloc()`. Closing a port will cause any tasks blocked on the port to be awoken and returned with the `ICS_PORT_CLOSED` error status. For named ports they will also be de-registered from the global name server.

`port` should be a valid port handle

Currently no `flags` bits are defined and this parameter must be set to zero.

ICS_port_lookup — Lookup an ICS port handle

Definition:

```
#include <ics.h>

ICS_ERROR ICS_port_lookup (const ICS_CHAR *portName,
                           ICS_UINT flags,
                           ICS_LONG timeout,
                           ICS_PORT *portp);
```

Arguments:

`portName`	Port name string.
`flags`	Various flag bits which affect behavior.
`timeout`	Time in milliseconds to block waiting for a response
`portp`	Port handle pointer used to return discovered handle.

Returns:

`ICS_SUCCESS`	Successfully discovered port handle.
`portp`	Discovered port handle.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_NAME_NOT_FOUND`	Port name was not found in the name server.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only.

Description: `ICS_port_lookup()` attempts to discover the port named `portName` in the global name server. If found then `ICS_SUCCESS` is returned and the associated port handle is returned. If multiple port handles with the same name are present in the global name server then the handles are returned in a round-robin order.

`portName` should be an ASCII NUL (\0) terminated string of length not exceeding `ICS_PORT_MAX_NAME` characters (not including the NUL).

`flags` can be set to `ICS_BLOCK` to cause the function to block until the named port is registered in the name server. In this case, the function will block for the number of milliseconds specified in the `timeout` parameter. Predefined values of `timeout` can be used:

– `ICS_TIMEOUT_IMMEDIATE` (return immediately)
– `ICS_TIMEOUT_INFINITE` (never return)

`portp` should be a pointer to an `ICS_PORT` object in which the discovered port handle will be returned on successful completion.

ICS_region_add Add a region to the local and remote CPU region tables

Definition:

```
#include <ics.h>

ICS_ERROR ICS_region_add (ICS_VOID *map,
                          ICS_OFFSET paddr,
                          ICS_SIZE size,
                          ICS_MEM_FLAGS mflags,
                          ICS_ULONG cpuMask,
                          ICS_REGION *regionp)
```

Arguments:

`map`	Virtual address of region base.
`paddr`	Physical address of region base.
`size`	Size of the region in bytes.
`mflags`	Memory region attributes.
`cpuMask`	Bitmask of logical CPUs.
`regionp`	Region handle pointer.

Returns:

`ICS_SUCCESS`	Successfully added region.
`regionp`	Region handle allocated.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only

Description: `ICS_region_add()` maps a memory region in both the local CPU and all the remote CPUs as indicated by `cpuMask`. These regions are used to enable zero-copy message passing to be achieved, by allowing common regions of physical memory to be mapped by each CPU.

`map` is the optional virtual address of the base of the memory region being registered. If it is set to NULL, then the appropriate mapping will be created on the local CPU. The supplied `mflags` and `paddr` arguments must be consistent with those of the `map` virtual address.

`paddr` should be the physical address of the memory region being added. It should be aligned to an `ICS_PAGE_SIZE` boundary.

`size` is the size in bytes of the whole memory region. It should be a whole multiple of `ICS_PAGE_SIZE`.

`mflags` specifies the memory attributes of the region being added. Valid values are `ICS_CACHED` and `ICS_UNCACHED`. Memory mappings with the corresponding

memory attributes will be created on the remote CPUs and also on the local CPU if a NULL `map` parameter was supplied.

`cpuMask` should be a logical CPU bitmask of each CPU which is required to map the new region. Passing in `ICS_CPU_ALL` for this argument will cause it to be added to all CPUs present.

Note: The calling CPU is assumed to be present in this CPU bitmask and not setting the corresponding bit will have no effect.

`regionp` is used to return a region handle to the caller. This handle should be used in future `ICS_region_remove()` calls.

ICS_region_phys2virt — Translate a physical memory region into a local virtual address

Definition:

```
#include <ics.h>

ICS_ERROR ICS_region_phys2virt (ICS_OFFSET paddr,
                                ICS_SIZE size,
                                ICS_MEM_FLAGS mflags,
                                ICS_VOID **addressp)
```

Arguments:

`paddrp`	Physical address to be translated.
`size`	Size of memory region.
`mflags`	Memory attribute flags requested.
`addressp`	Returned virtual address pointer.

Returns:

`ICS_SUCCESS`	Successfully translated address.
`addressp`	Virtual address translation.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	No translation found.

Context: Callable from task and interrupt context.

Description: `ICS_region_phys2virt()` translates a physical address into a local CPU virtual address. It does this by making use of the region tables as created with `ICS_region_add()`.

If a matching region is found, then the corresponding virtual address mapping will be returned in `addressp`.

For a matching region to be found, the local region table must contain a region which covers the whole of the supplied memory region and that also has the same memory attributes as specified in the `mflags` argument. If no matching region is found, then `ICS_INVALID_ARGUMENT` is returned.

`paddr` should be a valid physical address.

`size` should be the size of the region being translated.

`mflags` should be a valid memory attribute flag such as `ICS_CACHED`, `ICS_UNCACHED` or `ICS_PHYSICAL`.

`addressp` should be a valid pointer to an `ICS_VOID` pointer.

ICS_region_remove — Remove a region from the local and remote CPU region tables

Definition:

```
#include <ics.h>

ICS_ERROR ICS_region_remove (ICS_REGION region,
                             ICS_UINT flags)
```

Arguments:

`region`	Region handle.
`flags`	Various flag bits which affect behavior.

Returns:

`ICS_SUCCESS`	Successfully removed region.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid region handle was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.
`ICS_SYSTEM_TIMEOUT`	Failed to communicate with the name server.

Context: Callable from task context only.

Description: `ICS_region_remove()` removes and un-maps a memory region on both the local CPU and all the remote CPUs as specified in the original `ICS_region_add()` call.

`region` should be a valid region handle as returned by `ICS_region_add()`.

No `flags` bits are currently defined so this parameter must be set to zero.

ICS_region_virt2phys Translate a local virtual address into a physical one

Definition:

```
#include <ics.h>

ICS_ERROR ICS_region_virt2phys (ICS_VOID *address,
                ICS_OFFSET *paddrp,
                ICS_MEM_FLAGS *mflagsp)
```

Arguments:

address	Virtual memory address.
paddrp	Physical address pointer to be returned.
mflagsp	Memory attribute flags to be returned.

Returns:

ICS_SUCCESS	Successfully translated address.
paddrp	Physical address translation.
mflagsp	Memory attribute flags of translated address.

Errors:

ICS_NOT_INITIALISED	ICS not initialized.
ICS_INVALID_ARGUMENT	An invalid argument was supplied.
ICS_HANDLE_INVALID	No translation found.
ICS_SYSTEM_ERROR	A system error occurred.

Context: Callable from task and interrupt context.

Description: ICS_region_virt2phys() translates a local CPU virtual address into a physical one. It does this by making use of the region tables as created with ICS_region_add().

If a matching region is found, then the corresponding physical address and memory attributes will be written to paddrp and mflagsp respectively.

If no matching region is found, then the call will fall back to the OS specific virtual to physical mapping system, and in this case mflagsp will be set to ICS_PHYSICAL.

address should be a local CPU virtual address.

paddrp should be a valid pointer to an ICS_OFFSET object.

mflagsp should be a valid pointer to an ICS_MEM_FLAGS object.

ICS_watchdog_add — Install a CPU watchdog handler

Definition:

```
#include <ics.h>

typedef void (*ICS_WATCHDOG_CALLBACK)(ICS_WATCHDOG handle,
                ICS_VOID *param,
                ICS_UINT cpuNum)

ICS_ERROR ICS_watchdog_add (ICS_WATCHDOG_CALLBACK callback,
                ICS_VOID *param,
                ICS_ULONG cpuMask,
                ICS_UINT flags,
                ICS_WATCHDOG *handlep)
```

Arguments:

`callback`	Callback function for this watchdog.
`param`	Parameter to be supplied to the callback function.
`cpuMask`	Bitmask of all logical CPUs to be monitored.
`flags`	Various flag bits which affect behavior.
`handlep`	Handle pointer used to return allocated handle.

Returns:

`ICS_SUCCESS`	Successfully installed the watchdog.
`handlep`	Contains allocated watchdog handle.

Errors:

`ICS_NOT_INITIALISED`	ICS not initialized.
`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_ENOMEM`	Failed memory/resource allocation.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_watchdog_add()` installs a new CPU watchdog callback. The CPU watchdog system will continuously monitor the CPUs as indicated by the `cpuMask` bitmask. If any one of these CPUs crashes or is reset then the callback function will be called. Multiple watchdog callbacks can be installed, and it doesn't matter if the set of CPUs they are monitoring overlaps. When a CPU failure occurs, all watchdog callbacks monitoring the failed CPU will be called.

Once the callback has been triggered for a particular CPU, it will not trigger again until `ICS_watchdog_reprime()` has been called for that particular CPU. Using this technique means that if a CPU crashes and is rebooted, multiple watchdog callbacks are not generated. However, the callback handler may still be called if other CPUs in the monitored `cpuMask` bitmask fail.

`callback` must be a non-NULL `ICS_WATCHDOG_CALLBACK` function pointer.

`param` is an optional parameter pointer to be supplied to the callback function. It may be NULL.

`cpuMask` is a bitmask of each logical CPU to be monitored by this watchdog callback. Currently no `flag` bits are defined and this parameter must be set to zero.

`handlep` should be a valid pointer to an `ICS_WATCHDOG` object.

ICS_WATCHDOG_CALLBACK — Watchdog callback function

Definition:

```
#include <ics.h>

typedef void (*ICS_WATCHDOG_CALLBACK)(ICS_WATCHDOG handle,
                ICS_VOID *param,
                ICS_UINT cpuNum);
```

Arguments:

`handle`	Watchdog handle associated with this callback.
`param`	`param` argument as supplied with the install function.
`cpuNum`	Logical CPU number of failed CPU.

Returns: None

Errors: None

Context: Called from task context only.

Description: This is the watchdog callback function that is invoked for each watchdog trigger that occurs for the monitored CPUs. It will be called using the same `param` argument as supplied during `ICS_watchdog_add()`.

It is also supplied with the allocated watchdog `handle`, and the failed `cpuNum`.

Note: As this callback is called from task context, it is allowed to call any of the ICS task context functions.

ICS_watchdog_remove Remove a previously installed watchdog callback.

Definition:

```
#include <ics.h>

ICS_ERROR ICS_watchdog_remove (ICS_WATCHDOG handle)
```

Arguments:

`handle`	Handle of watchdog to be removed.

Returns:

`ICS_SUCCESS`	Successfully removed the watchdog callback.

Errors:

`ICS_HANDLE_INVALID`	An invalid handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_watchdog_remove()` removes a previously installed watchdog callback handler. All further watchdog triggers of the monitored CPUs will be ignored.

`handle` should be a valid watchdog handle as allocated by `ICS_watchdog_add()`.

ICS_watchdog_reprime — Reprime a triggered watchdog callback

Definition:

```
#include <ics.h>

ICS_ERROR ICS_watchdog_reprime (ICS_WATCHDOG handle,
                                ICS_UINT cpuNum)
```

Arguments:

`handle`	Handle of watchdog to be re-primed.
`cpuNum`	Logical CPU number of the CPU to monitor.

Returns:

`ICS_SUCCESS`	Successfully re-primed the watchdog.

Errors:

`ICS_INVALID_ARGUMENT`	An invalid argument was supplied.
`ICS_HANDLE_INVALID`	An invalid handle was supplied.
`ICS_SYSTEM_ERROR`	A system error occurred.

Context: Callable from task context only.

Description: `ICS_watchdog_reprime()` reprimes a watchdog handler that has previously triggered. Once a watchdog for a particular CPU has been triggered, the callback handler for that CPU will not be issued again until it has been reprimed by calling this function.

`handle` should be a valid watchdog handle as allocated by `ICS_watchdog_add()`.

`cpuNum` should be the logical number of the CPU for which watchdog monitoring is to be resumed.

6.3 Macro definitions

This section provides detailed descriptions of the ICS macros.

These macros can be used by the programmer to determine at compile time which version of the ICS system is being used.

ICS_VERSION C language macro describing the ICS version

Definition:

```
#include <ics.h>

#define ICS_VERSION(MAJOR,MINOR,PATCH)
```

Description: `ICS_VERSION()` is a macro that takes a major, minor and patch level as arguments and generates a version code that can be compared against `ICS_VERSION_CODE`.

For example:

```
#if ( ICS_VERSION_CODE >= ICS_VERSION(1, 1, 0) )
          /* Call an API added in V1.1.0 */
#endif
```

ICS_VERSION_CODE C language macro describing the ICS version

Definition:

```
#include <ics.h>

#define ICS_VERSION_CODE
```

Description: `ICS_VERSION_CODE` is the current ICS version code.

Appendix A ICS board support package

The ICS BSP is used to configure SoC specific information such as Mailbox addresses and reset registers in order for ICS to function. It also contains the CPU mapping table that is used to map CPU names onto their logical CPU numbers.

Currently ICS is supplied with BSPs for a limited set of SoCs. However, it is fairly simple to create a BSP for other SoCs. The BSP code can be found in `source/src/bsp`.

Taking the STi7200 as an example, we have:

```
stx7200/cores.c
stx7200/st40/mbox.c
stx7200/st40/name.c
stx7200/st40/reset.c
stx7200/st40/sti7200reg.h

stx7200/st231/audio0/mbox.c
stx7200/st231/audio0/name.c

stx7200/st231/audio1/mbox.c
stx7200/st231/audio1/name.c

stx7200/st231/video0/mbox.c
stx7200/st231/video0/name.c

stx7200/st231/video1/mbox.c
stx7200/st231/video1/name.c
```

The format and content of these BSP files is straight forward and the supplied BSPs can be used as a template for creating new ones. The SoC specific header file `sti7200reg.h` can be copied directly from the ST40 Micro Toolset, example directory.

In order for the OS21 BSP to be built by Multicom then the new SoC and core names must be added to the relevant makefiles in the BSP directory, for example:

```
makebspst40.inc
makebspst231.inc
```

For the Linux kernel module build, the new BSP files need to be added to:

```
source/src/ics/Makefile
```

The Multicom 4 library can also be built using the external BSP source directory specified by the `BSP_SRCDIR` **make** environment variable. This directory should have exactly the same directory hierarchy and file locations as the one in the Multicom 4 source tree. The included files `makebspst40.inc` and `makebspst231.inc` can then be modified to specify the new SoC BSPs.

The `BSP_INCDIR` environment variable can also be used to supply an external include directory, during compilation of the external BSP.

Under the Linux kernel the original Multicom 4 `source/src/ics/Makefile` directory still needs to be modified to specify the new SoC. All the BSP files can be located in the external BSP directory.

Note: The `BSP_SRCDIR` and `BSP_INCDIR` paths supplied during the building of the ICS Linux kernel module must be relative to the Multicom 4 source tree `source/src/ics` directory.

A.1 BSP data structures

This section describes each of the BSP data structures for CPU, mailbox, reset and boot addresses.

Note: The data structures for the reset and boot addresses are defined for each CPU. It is important therefore that the order of the entries defined for the table of CPUs (described in Section A.1.1) are synchronized with the order of entries for the reset and boot address structures (defined in `reset.c`) and listed in Section A.1.3.

A.1.1 CPU table

This data structure defines an entry in the CPU name table. The CPU table is used to map CPU core names onto their logical CPU numbers.

```
struct bsp_cpu_info
{
  char *name;
  char *type;
  int   num;
  unsigned int flags;
};
```

Table 27. CPU name table structure

Structure member name	Description
`name`	CPU core name, for example, “`estb`” or “`audio`”
`type`	CPU architecture, for example, “`st40` or “`st231`”
`num`	Logical CPU number for this core (`-ve` to disable)
`flags`	Various flag bits

Note: The order of the CPU definitions in this table must match that used in the CPU boot and reset tables in `reset.c`.

The BSP declaration of this table are normally found in the top level `cores.c` file so that they are shared across all the CPU cores.

The following declarations need to be made in the BSP (normally found in `cores.c`);

```
extern unsigned int          bsp_cpu_count;
extern struct bsp_cpu_info   bsp_cpus[];
```

Where `bsp_cpus` is the array of CPU definitions and `bsp_cpu_count` is set to the number of entries in that array.

A.1.2 Mailbox table

This structure defines an individual Mailbox entry. Each mailbox entry defines a set of 4 x 32-bit mailbox registers. For local mailboxes, an interrupt request number (or address on OS21) should be provided so that interrupts can be accepted on that set of mailboxes. For non-local mailboxes (that is, mailboxes owned by other CPUs) then this should be set to 0.

Note: Each Mailbox IP on the SoC actually provides two independent sets of 4 x 32-bit mailbox registers. The second set is at offset 0x100 from the base address of the Mailbox IP.

```
struct bsp_mbox_regs
{
  void *base;
  unsigned int   interrupt;
  unsigned int   flags;
};
```

Table 28. Mailbox table structure

Structure member name	Description
`base`	Physical base address of the mailbox set
`interrupt`	Interrupt request number (or address on OS21) associated with this mailbox, if local. Set to 0 for remote mailboxes
`flags`	Various flag bits

The following declarations need to be made in the BSP (normally found in `mbox.c`);

```
extern unsigned int              bsp_mailbox_count;
extern struct bsp_mbox_regs      bsp_mailboxes[];
```

Where `bsp_mailboxes` is the array of mailbox definitions and `bsp_mailbox_count` is set to the number of entries in that array.

A.1.3 Reset and boot addresses

These data structures are used to define, for each CPU, the core reset and boot information so that ICS can load and start each CPU. The entries in these tables must be synchronized with entries in the CPU table, described in *Section A.1.1 on page 196*.

Boot address

```
/* Boot address info */
struct bsp_boot_address_reg
{
  volatile unsigned int *address;
  int   leftshift;
  unsigned int  mask;
};
```

Table 29. Boot address structure

Structure member name	Description
address	For each CPU, SYSCONF boot register address
leftshift	Left-shift to be applied to boot address
mask	Mask to be applied to boot address

Reset address

```
struct bsp_reg_mask
{
  volatile unsigned int *address;
  unsigned int  mask;
  unsigned int  value;
};
```

Table 30. Register address structure

Structure member name	Description
address	Register SYSCONF address
mask	Mask to be applied to register
value	Data value to be applied to the register

The following declarations need to be made in the BSP (normally found in reset.c);

```
extern struct bsp_boot_address_reg bsp_sys_boot_registers[];
extern struct bsp_reg_mask         bsp_sys_reset_bypass;
extern unsigned int                bsp_sys_reset_bypass_count;
extern struct bsp_reg_mask         bsp_sys_boot_enable[];
extern struct bsp_reg_mask         bsp_sys_reset_registers[];
```

Where bsp_boot_address_registers is an array of boot address definitions for each CPU core in the SoC.

The bsp_sys_reset_bypass array defines all the SYSCONF boot reset bypass registers and bit patterns necessary to allow individual CPUs to be reset. The bsp_sys_reset_bypass_count variable should be set to the number of entries in this array.

The bsp_sys_boot_enable table defines for each CPU, the SYSCONF boot enable registers and bit pattern to cause each CPU to boot.

The bsp_sys_reset_registers table defines for each CPU, the SYSCONF reset registers and bit patterns to cause each CPU to be reset. By setting an array entry value member to 0, the reset logic applies mask followed by ~mask to the register.

Normally these BSP declarations are only supplied in the ST40/sh4 BSPs as they are usually responsible for the reset and loading of the other CPU cores.

Note: These tables and the associated macro header file are normally copied directly from the ST40 Micro Toolset examples directory.

A.1.4 CPU core name

```
extern const char *   bsp_cpu_name;
```

This is a CPU core declaration and is usually found in the `name.c` BSP file for each CPU core. It should be set to the ASCII CPU core name of the corresponding CPU and must match one of the CPU names found in the `bsp_cpus[]` table.

A.2 Example BSP template

The following example BSP template is for the MB628/STx7141 based platform/SoC.

Note: Only the templates for the `estb` *and* `audio` *CPUs are included.*

A.2.1 CPU table

```
/* Filename: stx7141/cores.c */

struct bsp_cpu_info bsp_cpus[] =
{
  {
    .name = "estb",/* eSTB */
    .type = "st40",
    .num= 0,   /* MASTER */
    .flags= 0,
  },

  {
    .name = "ecm",/* eCM */
    .type = "st40",
    .num= 3,
    .flags= 0,
  },

  {
    .name = "audio",/* audio */
    .type = "st231",
    .num= 2,
    .flags= 0,
  },

  {
    .name = "video",/* video */
    .type = "st231",
    .num= 1,
    .flags= 0,
  },

};

unsigned int bsp_cpu_count = sizeof(bsp_cpus)/sizeof(bsp_cpus[0]);
```

A.2.2 Mailbox tables

```
/* Filename stx7141/st40/estb/mbox.c */

#define MBOX0_ADDR 0xfe211000
#define MBOX1_ADDR 0xfe212000
#define MBOX2_ADDR 0xfe211800
#define MBOX3_ADDR 0xfe212800

#ifdef __os21__

#include <os21/st40/sti7141.h>

#define VIDEO_MBOX_IRQ((unsigned int) &OS21_INTERRUPT_MBOX0_SH4_IRQ)/* MBOX0 SET2 */
#define ECM_MBOX_IRQ  ((unsigned int) &OS21_INTERRUPT_MBOX2_SH4_IRQ)/* MBOX2 SET2 */
#define AUDIO_MBOX_IRQ((unsigned int) &OS21_INTERRUPT_MBOX1_SH4_IRQ)/* MBOX1 SET2 */
#define ESTB_MBOX_IRQ((unsigned int) &OS21_INTERRUPT_MBOX3_LX_DH_IRQ) /* MBOX3 SET1 */

#endif

#ifdef __KERNEL__

/* Based on a ILC3 base of 65 + offsets in ADCS 8071978 (Table 10) */

#define VIDEO_MBOX_IRQ(65 + 9)/* MBOX0 SET2 */
#define ECM_MBOX_IRQ  (65 + 13)/* MBOX2 SET2 */
#define AUDIO_MBOX_IRQ(65 + 11)/* MBOX1 SET2 */
#define ESTB_MBOX_IRQ (65 + 14)/* MBOX3 SET1 */

#endif

struct bsp_mbox_regs bsp_mailboxes[] =
{
  {
    .base= (void *) (MBOX0_ADDR), /* Video LX Mailbox (MBOX0.1) */
    .interrupt= 0,/* Video owns SET1 */
    .flags      = 0
  },
  {
    .base= (void *) (MBOX0_ADDR+0x100), /* Video LX Mailbox (MBOX0.2) */
    .interrupt= VIDEO_MBOX_IRQ,/* *WE* own SET2 */
    .flags      = 0
  },


  {
    .base= (void *) (MBOX2_ADDR), /* ST40 ECM Mailbox (MBOX2.1) */
    .interrupt= 0,/* ECM owns SET1 */
    .flags      = 0
  },
  {
    .base= (void *) (MBOX2_ADDR+0x100),/* ST40 ECM Mailbox (MBOX2.2) */
    .interrupt= ECM_MBOX_IRQ,/* *WE* own SET2 */
    .flags      = 0
  },


  {
    .base= (void *) (MBOX1_ADDR),/* Audio LX Mailbox (MBOX1.1) */
    .interrupt  = 0,/* Audio owns SET1 */
    .flags      = 0
  },
```

```
{
    .base= (void *) (MBOX1_ADDR+0x100),/* Audio LX Mailbox (MBOX1.2) */
    .interrupt  = AUDIO_MBOX_IRQ,/* *WE* own SET2 */
    .flags      = 0
  },


  {
    .base= (void *) (MBOX3_ADDR),/* ST40 ESTB Mailbox (MBOX3.1) */
    .interrupt  = ESTB_MBOX_IRQ,/* *WE* own SET1 */
    .flags      = 0
  },
  {
    .base= (void *) (MBOX3_ADDR+0x100),/* ST40 ESTB Mailbox (MBOX3.2) */
    .interrupt  = 0,/* ECM owns SET2 */
    .flags      = 0
  },

};

unsigned int bsp_mailbox_count = sizeof(bsp_mailboxes)/sizeof(bsp_mailboxes[0]);

/* Filename: stx7141/st231/audio/mbox.c

#include <bsp/_bsp.h>

#define MBOX0_ADDR 0xfe211000
#define MBOX1_ADDR 0xfe212000
#define MBOX2_ADDR 0xfe211800
#define MBOX3_ADDR 0xfe212800

#include <os21/st200/sti7141.h>

struct bsp_mbox_regs bsp_mailboxes[] =
{
  {
    .base= (void *) (MBOX0_ADDR), /* Video LX Mailbox (MBOX0.1) */
    .interrupt= 0,
    .flags      = 0
  },
  {
    .base= (void *) (MBOX0_ADDR+0x100), /* Video LX Mailbox (MBOX0.2) */
    .interrupt= 0,
    .flags      = 0
  },


  {
    .base= (void *) (MBOX2_ADDR), /* ST40 ECM Mailbox (MBOX2.1) */
    .interrupt= 0,
    .flags      = 0
  },
  {
    .base= (void *) (MBOX2_ADDR+0x100), /* ST40 ECM Mailbox (MBOX2.2) */
    .interrupt= 0,
    .flags      = 0
  },


  { .base= (void *) (MBOX1_ADDR),/* Audio LX Mailbox (MBOX1.1) */
    .interrupt  = (unsigned int) &OS21_INTERRUPT_MAILBOX,/* *WE* own this one */
    .flags      = 0
  },
```

```
  { .base= (void *) (MBOX1_ADDR+0x100),/* Audio LX Mailbox (MBOX1.2) */
    .interrupt  = 0,
    .flags      = 0
  },


  { .base= (void *) (MBOX3_ADDR),/* ST40 ESTB Mailbox (MBOX3.1) */
    .interrupt  = 0,
    .flags      = 0
  },
  { .base= (void *) (MBOX3_ADDR+0x100),/* ST40 ESTB Mailbox (MBOX3.1) */
    .interrupt  = 0,
    .flags      = 0
  },

};

unsigned int bsp_mailbox_count = sizeof(bsp_mailboxes)/sizeof(bsp_mailboxes[0]);
```

A.2.3 Reset and boot addresses

```
/* Filename: stx7141/st40/reset.c /*

#include <bsp/_bsp.h>

#include "sti7141reg.h"

/* Reset bypass mask must not unset masked eCM reset as reset will be asserted. */

struct bsp_reg_mask bsp_sys_reset_bypass[] = {
  {STI7141_SYSCONF_SYS_CFG08, ~(1 << 3), (1 << 4) | (1 << 3)},
  {STI7141_SYSCONF_SYS_CFG09, ~((1 << 28) | (1 << 27)), (1 << 27)}
};

/* Size of the above array */
unsigned int bsp_sys_reset_bypass_count =
sizeof(bsp_sys_reset_bypass)/sizeof(bsp_sys_reset_bypass[0]);

struct bsp_boot_address_reg bsp_sys_boot_registers[] = {
  {STI7141_SYSCONF_SYS_CFG08, 3, 0xFFFF0000},   /* eSTB */
  {STI7141_SYSCONF_SYS_CFG04, 3, 0xFFFF8000},   /* eCM */
  {STI7141_SYSCONF_SYS_CFG26, 0, 0xFFFFFF00},   /* audio */
  {STI7141_SYSCONF_SYS_CFG28, 0, 0xFFFFFF00}    /* video */
};

struct bsp_reg_mask bsp_sys_boot_enable[] = {
  {STI7141_SYSCONF_SYS_CFG08, ~1, 1},/* eSTB */
  {STI7141_SYSCONF_SYS_CFG08, ~2, 2},/* eCM */
  {STI7141_SYSCONF_SYS_CFG26, ~1, 1},/* audio */
  {STI7141_SYSCONF_SYS_CFG28, ~1, 1}/* video */
};

struct bsp_reg_mask bsp_sys_reset_registers[] = {
  {STI7141_SYSCONF_SYS_CFG08, ~(1 << 6), 1 << 6},/* eSTB */
  {STI7141_SYSCONF_SYS_CFG08, ~(1 << 7), 1 << 7},/* eCM */
  {STI7141_SYSCONF_SYS_CFG27, ~1, 1}, /* audio */
  {STI7141_SYSCONF_SYS_CFG29, ~1, 1} /* video */
};
```

A.2.4 CPU core name

```
/* Filename: stx7141/st40/estb/name.c /*
#include <bsp/_bsp.h>

const char * bsp_cpu_name = "estb";

Filename: stx7141/st231/audio/name.c
#include <bsp/_bsp.h>

const char * bsp_cpu_name = "audio";
```

Appendix B MME supplement

This appendix provides supplementary information to *Chapter 2: Using the MME API*.

B.1 Parameter encoding

This section describes a simply interface for a simple MPEG video decoder showing how the configuration parameters can be customized for each transformer.

B.1.1 Samples definitions

lists the MPEG video decoders specific definitions.

Table 31. MPEG video decoders specific definitions

Type	Description
MPGV_PictureType_t	Defines the different picture types an MPEG video transformation can handle.
MPGV_GlobalParams_t	Parameters to be used for the next transformation an MPEG transformer has to process.
MPGV_DecodeParams_t	Parameters used by an MPEG transformer to decode an MPEG picture.

MPGV_PictureType_t

Definition:

```
typedef enum
{
     MPGV_PICTURE_TYPE_I,
     MPGV_PICTURE_TYPE_P,
     MPGV_PICTURE_TYPE_B,
} MPGV_PictureType_t;
```

Description: Defines the different picture types an MPEG video transformation can handle.

Fields:

MPGV_PICTURE_TYPE_I	Picture type is I.
MPGV_PICTURE_TYPE_P	Picture type is P.
MPGV_PICTURE_TYPE_B	Picture type is B.

See also: *MPGV_DecodeParams_t*

MPGV_GlobalParams_t

Definition:

```
enum MPGV_GlobalParamsIdx {
MME_OFFSET_MPGVGlobal_horizontal_size_value,
MME_OFFSET_MPGVGlobal_vertical_size_value,
MME_OFFSET_MPGVGlobal_intra_quantiser_matrix,
MME_OFFSET_MPGVGlobal_non_intra_quantiser_matrix =
   MME_OFFSET_MPGVGlobal_intra_quantiser_matrix + 64,

MME_LENGTH_MPGVGlobal =
   MME_OFFSET_MPGVGlobal_non_intra_quantiser_matrix + 64

#define MME_TYPE_MPGVGlobal_horizontal_size_value        U32
#define MME_TYPE_MPGVGlobal_vertial_size_value           U32
#define MME_TYPE_MPGVGlobal_intra_quantiser_matrix       U8
#define MME_TYPE_MPGVGlobal_non_intra_quantiser_matrix   U8
};
typedef MME_GenericParams_t MPGV_GlobalParams_t[MME_LENGTH(MPGVGlobal)];
```

Description: Parameters to be used for the next transformation an MPEG transformer has to process.

The following code provides a simplified example:

```
MPGV_GlobalParams_t params;

MME_PARAM(params, Length) = MME_LENGTH(MPGVGlobal);
MME_PARAM(params, MPGVGlobal_horizontal_size_value) = hsv;
MME_PARAM(params, MPGVGlobal_vertical_size_value) = vsv;
for (i=0; i<64; i++) {
MME_INDEXED_PARAM(params, MPGVGlobal_intra_quantiser_matrix, i) =
            iqm[i];
    /* ... */
}
```

or to declare parameters statically:

```
MPGV_GlobalParams_t params = {
10,
10,

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,

/* ... */
}
```

See also: *MME_AllocationFlags_t*

MME_SendCommand

MME_PARAM

MME_INDEXED_PARAM

MPGV_DecodeParams_t

Definition:

```
enum MPGVIdx {
MME_OFFSET_MPGV_picture_type,
MME_OFFSET_MPGV_full_pel_forward_vector,
MME_OFFSET_MPGV_forward_f_code,
MME_OFFSET_MPGV_full_pel_backward_vector,
MME_OFFSET_MPGV_backward_f_code,
MME_OFFSET_MPGV_forward_horizontal,
MME_OFFSET_MPGV_forward_vertical,
/* ... */

MME_LENGTH_MPGV

#define MME_TYPE_MPGV_picture_type                  MPGV_Picture_t
#define MME_TYPE_MPGV_full_pel_forward_vector       U32
#define MME_TYPE_MPGV_forward_f_code                U32
#define MME_TYPE_MPGV_full_pel_backward_vector      U32
#define MME_TYPE_MPGV_backward_f_code               U32
#define MME_TYPE_MPGV_forward_horizontal            U32
#define MME_TYPE_MPGV_forward_vertical              U32
/* ... */
};
typedef MME_GenericParams_t MPGV_DecodeParams_t[MME_LENGTH(MPGV)];
```

Description: Parameters to be used by a MPEG transformer to be decode an MPEG picture.

Fields:

MPGV_picture_type	Type of the picture that has to be decoded
MPGV_full_pel_forward_vector	As described in ISO/IEC 13818-2.
MPGV_forward_f_code	As described in ISO/IEC 13818-2.
MPGV_full_pel_backward_vector	As described in ISO/IEC 13818-2.
MPGV_backward_f_code	As described in ISO/IEC 13818-2.
MPGV_forward_horizontal	As described in ISO/IEC 13818-2.
MPGV_forward_vertical	As described in ISO/IEC 13818-2.

...

The following code provides an example:

```
MME_Command_t command;
MME_CommandStatus_t status;
MME_DataBuffer_t buffers[4];
MPGV_DecodeParams_t params;

command.StructSize = sizeof(command);
command.CmdEnd = MME_COMMAND_END_RETURN_NOTIFY;
command.DueTime = now + (20 * MS);
command.CmdStatus_p = &status;
command.NumInputBuffers = 3;
command.NumOutputBuffers = 1;
command.Buffers_p = &buffers;
command.Param_p = &params;

status.StructSize = sizeof(status);
```

```
/* Setup the input buffers, compressed data, backward reference picture and
forward reference picture */
/* Setup the output buffer, the decompressed picture */

/* Setup the transformer specific parameter structure */
MME_PARAM(params, MPGV_picture_type) = pt;
MME_PARAM(params, MPGV_full_pel_forward_vector = 0;
/* ... */
err = MME_SendCommand(handle, MME_TRANSFORM, &command);
```

See also:

MPGV_GlobalParams_t

MME_AllocationFlags_t

MME_SendCommand

MME_PARAM

Appendix C Advanced build options

This appendix describes how environment or make command line variables can be used to tailor the Multicom build process. See *Chapter 1: Building Multicom on page 8* for details on how to build the software.

Note: Many of the options described below alter they way the software is built. For such changes it is important that the tree is cleaned before building to prevent **make***'s build avoidance techniques from interfering with the changes.*

C.1 Debugging assertions and logging

All target resident source code is supplied with the Multicom distribution, this allows application developers to enable the in built assertion checking and/or run-time logging to help them identify problems.

Note: Debug assertions are runtime tests compiled into the application that, to a limited extent, verify correct operation of the program. This is supplementary to normal debugging which requires only debugging information. Compiling with debugging information is discussed in Chapter 1: Building Multicom on page 8.

All these facilities are controlled at build time by a single environment or make variable `DEBUG_CFLAGS`. The contents of this variable are placed on the command line for every compiler invocation allowing `DEBUG_CFLAGS` to be used to defined C pre-processor macros the alter the build.

Table 32. Pre-processor macros that enable diagnostic code

Pre-processor macro	Purpose
`ICS_DEBUG`	Enable all debug assertions within the ICS tree. Also causes the tracing code to be compiled in. When used alone this macro does not cause any tracing to be output since each module must have tracing separately enabled.
`ICS_DEBUG_FLAGS=<mask>`	Enable individual ICS subsystem library tracing.
`ICS_DEBUG_MEM=1`	Enable extra checking and tracing of all memory allocations from the ICS subsystem.
`MME_DEBUG`	Enable all debug assertions within the MME tree. Also causes the tracing code to be compiled in. When used alone this macro does not cause any tracing to be output since each module must have tracing separately enabled.
`MME_DEBUG_FLAGS=<mask>`	Enable individual MME subsystem library tracing,

To build an Multicom tree with the maximum possible amount of diagnostic code the following command line could be used.

```
make DEBUG_CFLAGS="-DICS_DEBUG -DICS_DEBUG_MEM=1
     -DICS_DEBUG_FLAGS=-1 -DMME_DEBUG -DMME_DEBUG_FLAGS=-1"
```

`DEBUG_CFLAGS` does not have to be specified on the make command line; it can also be set as an environment variable.

When running the host under Linux, debug logging can be enabled in the same way, that is by compiling the Multicom 4 kernel drivers, having set `DEBUG_CFLAGS` appropriately. The debug logging level and output channel can then be controlled by using the kernel module parameters supplied to both the ICS and MME kernel modules. For example:

```
insmod ics.ko debug_flags=1 debug_chan=5
insmod mme.ko debug_flags=1
```

C.2 Tuneable parameters

MME allows parameters such as thread priority to be tuned without recompiling Multicom components. The function to modify tuneable parameters is:

```
MME_ERROR MME_ModifyTuneable(MME_Tuneable_t key, MME_UINT value)
```

Each call to this function allows a single tuneable value to be updated. See *MME_ModifyTuneable on page 61* for further details.

Appendix D ICS Linux module parameters

This appendix describes the installation parameters that can be supplied to the ICS kernel module.

D.1 Support for declaring ICS regions on the module load

The parameters used for this purpose have the following syntax:

```
regions = region[,region]
region = <bpa2name>|<phys_base>:<size>
```

Where <*bpa2name*> is the ASCII name of the `bpa2` partition, for example, `LMI_VID` or `LMI_SYS`. Otherwise the physical base address and size of a region can be specified directly.

When configured these regions are mapped into all CPUs as cached and uncached translations.

D.2 Support for declaring ICS companion firmware on the module load

The parameters used for this purpose have the following syntax:

```
firmware = companion[,companion]
companion = <cpu>:<filename>
```

Where <*cpu*> can either be the logical CPU number (integer) or the CPU core name, for example: `audio0` or `video0`. <`filename`> must be an ASCII filename of an ELF file located in `/lib/firmware`.

D.3 Support for contiguous allocations from a named BPA2 memory partition

The parameter used for this purpose have the following syntax:

`bpa2_part` = “*part_name*”

The default for this value is “`bigphysarea`”.

Revision history

Table 33. Document revision history

Date	Revision	Changes
24-Jan-2011	C	Minor rewordings not listed. Updated *Chapter 1: Building Multicom on page 8* and *Section 1.1.2: Compiler recommendations on page 9* to add STLinux releases and footnote. Updated *Section 1.4: Compiling and linking against the Multicom 4 libraries on page 10* to remove inappropriate note. Corrected typo in *Section 1.7: Running the tests under OS21 on page 12*. Updated *Section 2.5.2: Manually managing data buffers on page 23* to correctly refer to the function `MME_RegisterMemory` rather than `MME_RegisterBuffer`. Updated *Section 2.11.1: OS21 on page 31* to clarify. Documented the `BSP_INCDIR` environment variable in *Appendix A: ICS board support package on page 195*.

Table 33. Document revision history (continued)

Date	Revision	Changes
29-Jun-2010	B	Supports the Multicom 4 R4.0.2. Removed all references to `ics_load_entry` and `ics_load_free`. Added details about Linux userspace to *Chapter 1: Building Multicom on page 8*, throughout. Updated *Table 1 on page 8*. Updated *Section 1.1.2: Compiler recommendations on page 9*. Updated the Notes in *Section 1.4: Compiling and linking against the Multicom 4 libraries on page 10*. Updated *Section 1.6: Debugging support on page 11* and added *Section 1.6.1: Debug logging on page 12*. Updated *Section 1.8: Running the tests under Linux on page 12*. Updated *Section 2.1.1: Transformers and transformer instances on page 14* and added *Figure 2*. Moved multi-hosting material to *Section 2.1.2: Multi-hosting support on page 15* and added *Figure 3*. Added *Section 2.8: Fault detection and recovery on page 28*. Added *MME_PingTransformer on page 66*. Updated the description of *MME_WaitCommand on page 72*. Added `MME_COMMAND_TIMEOUT` to *MME_ERROR on page 88*. Added `MME_TRANSFORMER_TIMEOUT` to *MME_Event_t on page 90*. Updated *Table 9 on page 109* and associated text. Updated *Section 5.2.3: ICS initialization and termination on page 110*. Added Note to *Section 5.8: CPU watchdog support on page 118*. Updated the description of *ics_cpu_init on page 120*, adding *Table 21.* Updated the definition and error of *ics_cpu_self on page 126*. Updated the description of *ics_cpu_start on page 127*. Updated the definition and description of *ics_load_elf_file on page 141* and *ics_load_elf_image on page 142*. Updated “Context” for *ICS_channel_send on page 151*. Updated the description of *ICS_cpu_disconnect on page 154*. Updated the description of *ICS_cpu_init on page 156*. Updated the description of *ICS_msg_send on page 170*. Introduction to *Appendix A: ICS board support package on page 195* rewritten and example updated. Updated `num` in *Table 27: CPU name table structure on page 196*. Updated *Reset address on page 198*. Updated *Section A.2.1: CPU table on page 199*. Updated *Section A.2.2: Mailbox tables on page 200*. Updated *Section A.2.3: Reset and boot addresses on page 202*. Updated *Section C.1: Debugging assertions and logging on page 208*. Added *Appendix D: ICS Linux module parameters on page 210*.
19-Oct-2009	A	Initial release.

Index

Symbols

B

C

D

E

F

H

I

L

M

N

O

R

S

T

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com