STMicroelectronics

Multicom

User manual

7574220 Rev G

April 2008

www.st.com

BLANK

User manual

Multicom

A guide to STMicroelectronics' inter-processor communications software for multi-media applications.

Contents

Preface

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**.

Documentation identification and control

This book carries a unique identifier in the form:

ADCS *nnnnnnnx*

Where,

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

Whenever making comments on this document the complete identification ADCS *nnnnnnnx* should be quoted.

Comments on this or other manuals in this documentation suite should be made by contacting your local STMicroelectronics Limited Sales Office or distributor.

Acknowledgements

Linux® is a registered trademark of Linus Torvalds.

Microsoft®, Windows® and Windows NT® are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Solaris is a trademark of Sun Microsystems, Inc. in the US and other countries.

SPARC® is a registered trademark of SPARC International Inc. in the US and other countries.

Intel® is a registered trademark of Intel Corporation or its subsidiaries in the US and other countries.

1 Introduction

1.1 Overview

This document is a guide to STMicroelectronics' Multicom product. Multicom combines three distinct inter-processor communication technologies:

- **MME** - Multi-Media Engine API, a communications API specifically designed to control media transformations. MME is described in *Part 2 MME user guide on page 18*.
- **RPC** - Remote Procedure Call mechanism, a means to communicate between processors using normal function calls. RPC is described in *Part 3 RPC user guide on page 51*.
- **EMBX** - Extended mailbox communications system, a low-level communications API used by both MME and RPC to communicate with other processors. EMBX can also be used directly by applications with unusual requirements. EMBX is described in *Part 4 EMBX user guide on page 66*.

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

A C function running on one processor may call a function on another processor by using RPC. To achieve this, the function's arguments are **marshalled** into a communications buffer and transmitted to a different CPU where the arguments are **demarshalled** and the remote function called. The process is repeated in reverse when the remote function completes.

The code required to marshall and demarshall the arguments is automatically generated by the RPC tools from the application's C source code. The source code must be augmented by extra information called decorations where the meaning of the code is ambiguous. The C language together with decorations and other administrative information is called the Interface Definition Language (IDL); see *Chapter 6: Interface declarations on page 57* and *Chapter 7: Decorating types and functions on page 60*.

EMBX provides a carefully designed application programmer's interface to allow data buffers to be transmitted to other CPUs.

EMBX is designed to give the most efficient method of data transfer. The application programmer does not need to know exactly how the buffer will be made available to the other CPU (especially whether it will be copied or directly addressed). This allows the programmer to develop software that can be moved to any communication architecture supported by EMBX without having to revisit any optimizations peculiar to that platform.

1.2 Targeted platforms

Multicom contains operating-system-specific code to manage internal communications. Multicom is currently supported on the operating systems listed in *Table 1*.

Table 1. Supported operating systems

Toolset	CPU	Operating System
ST40 Micro Toolset	ST40	OS21, Linux User Mode, Linux Kernel Mode(1)
ST200 Micro Toolset	ST200 series	OS21

1. EMBX is supported in Linux Kernel Mode only.

EMBX provides an abstraction of the underlying hardware; thus MME and RPC have no ties to a specific platform since all hardware dependencies are handled by EMBX.

The most recent platforms, targeted by the EMBX implementation, and supported by Multicom are listed in *Table 2*. Each of these platforms is supported with examples.

Table 2. Targeted platforms

Platform	Board name	CPU(s)
STb7100-Mboard	mb411	STb7100
STb7109-Ref board	mb442	STb7109
STi7200-Mboard board	mb519 plus mb520	STi7200
STi7111-Mboard platform	mb618	STi7111

EMBX is designed to minimize the effort required to retarget other similar platforms. This makes customizing the software for your own boards very simple, see *Chapter 9: Transport specifics on page 79* and *Appendix A: Transport configurations on page 192*.

1.3 The MME/EMBX stack

MME's primary purpose is to allow application and driver software to dispatch data to transformers for encoding, decoding or some other transformation. It can be used in both uni-processor or multi-processor hardware designs and provides a consistent API in both cases.

MME is well suited to hardware designs that intend the host processor to run the application and the companion processors to be devoted purely to running media transforms.

Developers targeting MME/EMBX can ignore *Part 3 RPC user guide*.

1.4 The RPC/EMBX stack

RPC is designed to facilitate software development on multi-processor embedded systems. It is not designed for workstation-like flexibility but rather for simplicity, portability, small size and high performance. For this reason it has a different character to more traditional RPC systems.

The set of application programs running in each processor must be known at build time. There is no method for a CPU to discover remote processors or remote calls at run-time. This allows RPC marshalling and other glue code to be pre-compiled for each specific system reducing the need for on-target processing.

It is possible to have functionality that is dynamically available by initializing and deinitializing stubs, however, the nature of that functionality must be fixed at build time.

RPC provides a rapid and simple means to partition a system across multiple processors. Its flexibility allows designs to be quickly altered during or after initial development.

Developers targeting RPC/EMBX can ignore *Part 2 MME user guide*.

1.5 Using EMBX without RPC or MME

There are multi-processor systems for which neither RPC nor MME are ideally suited, perhaps because of existing software legacy or because the system does not fit easily into a host/companion design. Such systems may still require inter-processor communication. In this case it is possible to use the EMBX component to provide a well supported communications interface without using the other components of Multicom.

Developers targeting purely EMBX can ignore *Part 2 MME user guide* and *Part 3 RPC user guide* entirely and go directly to *Part 4 EMBX user guide*.

Part 1 Getting started

This getting started covers:

- *Setting up the distribution*
- *Building Multicom*
- *Examples*

2 Getting started

2.1 Setting up the distribution

Unpack the Multicom distribution on a suitable host machine which has the appropriate target toolsets installed. See *Table 1 on page 10* for a list of applicable toolsets.

Set the environment variable `RPC_ROOT` to the directory containing the distribution. Add one of the following directories, derived from `RPC_ROOT`, to the `PATH`:

- `%RPC_ROOT%\bin\winnt\ia32` for Windows NT hosts
- `$RPC_ROOT/bin/solaris/sparc` for Solaris hosts
- `$RPC_ROOT/bin/linux/ia32` for GNU/Linux hosts

The Multicom build scripts examine the current environment to detect which toolsets are installed. In particular that means that the toolsets must be included in your `PATH`.

Finally all makefiles supplied with Multicom use GNU **make** syntax. Thus 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.

`RPC_ROOT` contains a deep directory structure. The contents of these directories are overviewed in *Table 3*.

Table 3. The distribution directories

Directory	Contains
`bin/<os>/<arch>/`	Host tool executables.
`configs/`	Platform configuration files for ST40.
`docs/`	Product documentation.
`examples/embx/`	Some example programs demonstrating the raw use of the EMBX API.
`examples/mme/`	Some example programs demonstrating the fundamentals of MME.
`examples/rpc/`	Some example programs demonstrating the fundamentals of the RPC system.
`include/`	C header files used by every processor.
`lib/<os>/<arch>/`	Compiled libraries or kernel modules. For example, the EMBX shell and various EMBX transports.
`src/embx/`	Complete source code for the EMBX implementation together with test suites.
`src/mme/`	Complete source code for the MME implementation together with test suites.
`src/mkfiles`	A library of makefiles. These are used to build the distribution, test suites and examples.
`src/platform/`	Source code for any platform specific libraries.
`src/rpc/`	Source code for all target resident RPC code together with a simple test suite.
`src/tools/`	Source code for host based tools used to manage the test suites.

2.2 Building Multicom

With the exception of the Linux kernel modules, all software is supplied pre-built for all targeted platforms. Nevertheless all target resident code supplied with Multicom is provided in source form to facilitate debugging and porting.

From `RPC_ROOT` the entire base can be built by issuing the following command:

```
make install
```

The build system automatically detects which toolsets you have installed and build the libraries using every installed toolset.

All intermediate object files can be removed using the following command:

```
make clean
```

There is no need to remove the library files since these will be overwritten automatically when the software is next built.

Note: Throughout the entire Multicom tree a diving make system is used. Thus the above commands can actually be issued from within any directory in the distribution. This performs the supplied action on the current directory and all subdirectories.

By default, Multicom builds are optimized for speed and have no debug information available. Rebuilding Multicom provides the opportunity to include this debug information; this is achieved by issuing the following command (on a clean tree):

```
make install ENABLE_DEBUG=1
```

2.2.1 Building Linux kernel modules

The Multicom build system is integrated with Linux's kbuild allowing Multicom's kernel modules to be built in a similar way to most other Linux kernel modules. In order to compile Linux kernel modules your Linux kernel source tree must be configured using `make menuconfig` or similar.

From `RPC_ROOT` the kernel modules can be built by issuing the following command:

```
make \
        ARCH=sh CROSS_COMPILE=sh4-linux- \
        KERNELDIR=<path_to_configured_linux_kernel_sources> \
        modules
```

It is also possible to build Multicom's kernel modules when the Linux kernel source code and the configured built tree are in separate directories. For example:

```
make \
        ARCH=sh CROSS_COMPILE=sh4-linux- \
        KERNELDIR=<path_to_original_linux_kernel_sources> \
        O=<path_to_built_linux_kernel_sources> \
        modules
```

Having built the modules it is possible to automatically install them onto the target file system providing that it is also mounted on the host. For example, the following command installs the kernel modules into the default target file system in `/opt/STM/STLinux-2.3` (assuming that the user has permission to write to `/opt/STM/STLinux-2.3/devkit/sh4/target/lib`).

```
make \
        ARCH=sh CROSS_COMPILE=sh4-linux- \
        KERNELDIR=<path_to_configured_linux_kernel_sources> \
        INSTALL_MOD_PATH=/opt/STM/STLinux-2.3/devkit/sh4/target \
        modules_install
```

2.3 Examples

This section describes the `getstart` examples in detail. These instructions apply to all the examples provided because they share the same build system and are all loaded onto the target in the same way.

The MME and RPC `getstart` examples are found in the directories *RPC_ROOT*`/examples/mme/getstart` and *RPC_ROOT*`/examples/rpc/getstart` respectively.

Three variables must be supplied on the `make` command line to select the target platform and the operating systems running on it.

Table 4. Make variables

Make variable	Description
`PLATFORM`	The name of the platform being compiled for. The platform name is typically the name of the board in lowercase, however, the list of valid platforms is contained in the `README` file for the example harness found in the harness directory.
`OS_0`	The operating system to be run on CPU 0. Valid operating systems are `os21` and `linux`.
`OS_1`	The operating system to be run on CPU 1. Valid operating systems are `os21` and `linux`.

Note: *For multi-core devices with different processor cores the* `PLATFORM` *variable selects which processor is regarded as CPU 0. The* `README` *file in the example directory documents the number assigned to each processor.*

In addition to the `make` variables the target example must be used. For example, to build for STB7109 board with ST40/OS21 and ST231/OS21 the following command must be issued:

```
make example PLATFORM=mb442 OS_0=os21 OS_1=os21
```

Like the rest of the Multicom tree you can include debug information by adding *ENABLE_DEBUG=1* to this command line.

When building an example a commentary is emitted, prefixed with **+++**. This describes the purpose of each command. For example, the following commentary is emitted when building one of the RPC examples for the STB7109 board:

```
+++ Compile ST40/OS21 source [application.c]. +++
+++ C pre-process the interface definition for app. +++
+++ Generate the app stubs [app.stubs.c]. +++
+++ Compile ST40/OS21 source [app.stubs.c]. +++
+++ Link ST40/OS21 application for CPU 0 [obj/os21/st40/mb442/application.out]. +++
+++ Compile ST231/OS21 source [cdplayer.c]. +++
+++ C pre-process the interface definition for cd. +++
+++ Generate the cd stubs [cd.stubs.c]. +++
+++ Compile ST231/OS21 source [cd.stubs.c]. +++
+++ Link ST231/OS21 application for CPU 1 [obj/os21/st231/mb442/cdplayer.out]. +++
```

In this example two target executables are built, `obj/os21/st40/mb442/application.out` and `obj/os21/st231/m442/cdplayer.out`.

The precise file extensions and paths vary depending on the toolsets used to build the example. Read the build commentary to determine the paths needed. The examples do not include a mechanism to load output executables onto a target. The user must use their appropriate loader tools and environment.

Refer to *Appendix A: Transport configurations on page 192* for details of how to load code on your particular platform.

2.3.1 MME example

The MME `getstart` example represents a simple frame-based 'mixer', which mixes two input byte sequences to produce a single output sequence. The example executes first using a local transformer, and then a remote transformer.

A character-based bar graph output is displayed on the host console to represent the magnitudes of the input and output signals.

See the `README` and comments in the example code for further information.

2.3.2 Running examples on Linux

Note: These instructions assume you have already built and installed the Linux kernel modules as described in Section 2.2.1: Building Linux kernel modules on page 14.

The example programs, when run under OS21, include all the EMBX transport configuration required to run the example. Due to the way the components are separated, it is not possible to do this for GNU/Linux.

In order to run the examples on GNU/Linux the Multicom kernel modules must be correctly loaded before running the example. Perform the following steps.

1. Configure the modules by storing their parameters in the module loader configuration file, `/etc/modprobe.conf`. The harness example includes the file `$RPC_ROOT/examples/harness/modprobe.conf`. Copy this file to your target file system and remove the comment characters from the configuration parameters for your board.
2. Analyze the modules dependencies. This is done by executing the following command (on the target):
   ```
   root@stlinux:~# depmod -a
   ```
   This is done each time the modules are rebuilt.
3. Load the modules with the following sequence of commands after each reboot:
   ```
   root@stlinux:~# modprobe embxshm
   root@stlinux:~# modprobe mme_host
   ```

Note: When `mme_host` is loaded it will attempt to rendezvous with the other processor, thus `modprobe` will appear to hang if the other processor it not running.

Once the modules have been loaded the system is ready to run the example programs.

2.3.3 RPC example

The RPC examples directory contains a number of examples that demonstrate the use of various RPC techniques. Each example contains a file called `README` that describes the purpose of the example and highlights any particular points of interest.

Part 2 MME user guide

The MME user guide covers:

- *Using the MME API*
- *Writing an MME transformer*

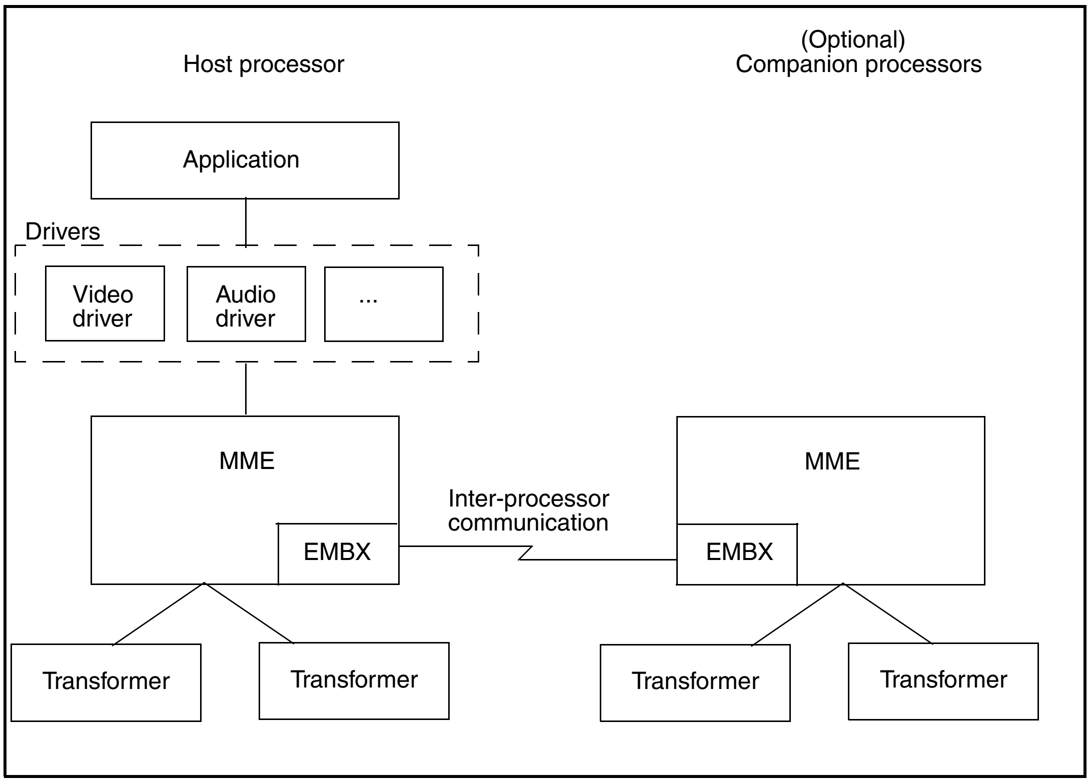
3 Using the MME API

3.1 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 EMBX communication interface. 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*.

The MME API is delivered as two libraries. The host library contains the entire MME API and is used by the host processor. The companion library is used by zero or more companion processors and contains only a subset of the API. This subset contains only those functions required to register transformers and allow them to be accessed by the host processor.

Figure 1. Typical MME software stack



3.1.1 Transformers and transformer instances

A transformer is registered, typically when the system is initialized, in an abstract form (see *Section 3.2.3: Registering transformers on page 23*). 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.

3.1.2 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.

3.1.3 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.

3.1.4 Due time

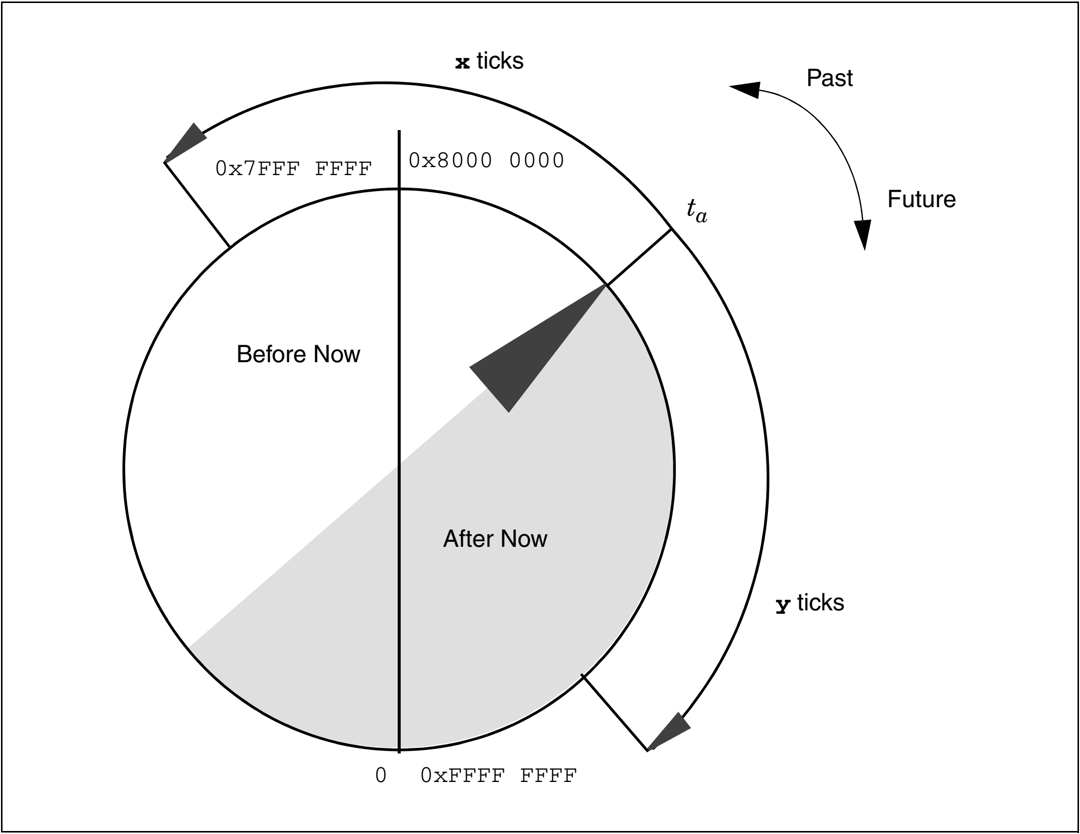
The due time is used by the MME implementation to determine in what order to process commands. The command queue for each transformer is maintained in due time order and when a command is dispatched all the queues are examined and the one with the lowest due time is selected for execution.

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 3.7: Types of commands on page 32.*

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

Figure 2. 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.

3.1.5 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.

3.1.6 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 10.3: MME constants, enums and types on page 163*.

3.2 Initialization

Initialization of a system is divided into three distinct stages. These are:

1. Initializing MME (which presupposes that the EMBX is initialized, see *Section 8.2 on page 68*).
2. Registering EMBX transports used to communicate with other processors.
3. Registering transformers.

Note: Steps 2. and 3. do not need to be performed in order.

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 single processor (host only) systems, there is no need to register EMBX transports because there are no other processors.
- For multi-processor OS21 systems, all steps are required for companion processors. Registering transformers is optional for host processors.
- For Linux **user space** applications, EMBX transports are registered when the MME module is loaded by module parameters. They cannot be registered from Linux user space.
- For Linux **kernel space** operation, MME is initialized automatically when the MME module is loaded. EMBX transports may optionally be registered by module parameters when the MME module is loaded. Registering local transformers is optional.

  Section 3.9.2: Linux on page 35 contains further details about loading Linux kernel modules.

Note: It is not possible to have a companion implemented under Linux.

3.2.1 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.*

3.2.2 Registering EMBX transports

EMBX transports can be registered, as soon as both EMBX and MME have been initialized. A transport is registered using the following function:

```
MME_ERROR MME_RegisterTransport(const char *transport_name)
```

When a transport is registered, MME immediately attempts to open the transport and establish communication. This function blocks only if the underlying transport is not ready to be opened and remains waiting for other processors to participate.

3.2.3 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 4: Writing an MME transformer*.

3.2.4 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

```
/* Do the CPU-specific EMBX init here - EMBX transports registered */
/* This is application-supplied on each CPU */

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

/* For each EMBX transport */
res = MME_RegisterTransport(transport);

/* 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);
res = MME_DeregisterTransport();
res = MME_Term();
```

Companion-side example

```
/* Do the CPU-specific EMBX init here - EMBX transports registered */
/* This is application-supplied on each CPU */

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

/* For each EMBX transport */
res = MME_RegisterTransport(transport);

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

/* This call returns when the host side calls MME_Term() */
res = MME_Run();

/* The following are needed in case MME_Run() terminates abnormally */
res = MME_DeregisterTransport(transport);
MME_Term();
```

3.3 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 3.5: Application and transformer specific data on page 29* and *Section 4.7: Parameter passing on page 45*.

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.

3.3.1 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.

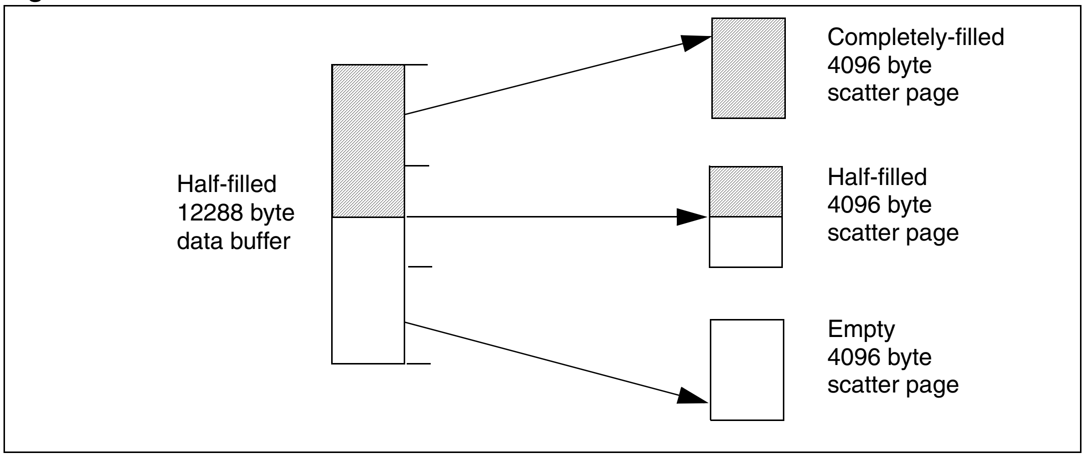
3.4 Buffer and cache management

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 struct `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 3. 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.

3.4.1 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)
```

A data buffer is allocated with an **affinity** to a particular transformer handle. This means that the memory returned will be allocated such that it is suited for optimal communication with a particular transformer instance. The data buffer can be used successfully with any other transformer instance though not always optimally. For example, a buffer allocated for a local transformer may not be able to make use of a zero copy acceleration when used with a remote transformer.

Note: When a chain of transformers is used, the cost of converting to an optimal buffer between links in the chain is likely to be larger than the cost of using a non-optimal buffer. Applications should therefore reuse buffers between links when this is possible. In most cases allocating a buffer with an affinity to the most compute intensive link in the chain, will yield optimal results.

It is possible to override pure affinity based allocations if the application requires specific properties of memory once outside the scope of MME. For example, if the transformed data is presented to a linear DMA engine it may be necessary to force allocation from a single scatter page. Similarly pre-scanning of a buffer by the host may require the data to be held in uncached memory so the host does not have to flush and invalidate its data cache prior to calling MME.

It may be useful in some situations to sub-divide memory allocated by MME into user-specified scatter pages. An example of this would be to use multiple scatter pages to divide a multiplexed audio and video stream without copying the original data. It is therefore quite legitimate to reuse the underlying memory locations outside of the original data buffer. It is not, however, possible to deallocate these data buffers individually. Only data buffers allocated by MME can be freed by MME. For this reason, the original data buffer must be retained in order to successfully free the buffer.

3.4.2 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 3.4.4.

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

This is a perfectly acceptable approach to application design but it is important that the application designer appreciate the care that may be required in order to achieve optimum efficiency.

In order to transfer data buffers, MME registers each scatter page as an EMBX distributed object[a], allowing the underlying EMBX transport to copy each page in the most efficient manner. If the page cannot easily be addressed (in a cache coherent manner) by a companion CPU, EMBX has to copy the data which is potentially time consuming. If the memory is cached it is also possible that EMBX has to make pessimistic assumptions regarding whether it is held in the CPU's cache.

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, forcing at least a partial copy.

3.4.3 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;
```

a. There are, in fact, a few fast paths through MME that avoid this registration but they are not visible to the application and therefore not mentioned, in order to simplify the explanation.

```
/* 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);
```

3.4.4 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.

3.4.5 Cache management

When a data buffer is held in cached memory, EMBX 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 3.4.1: Allocating data buffers on page 26*).

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 5 on page 29*, prior to being made available to the host.

Table 5. 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.

3.5 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 4.7: Parameter passing on page 45*.

3.6 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.

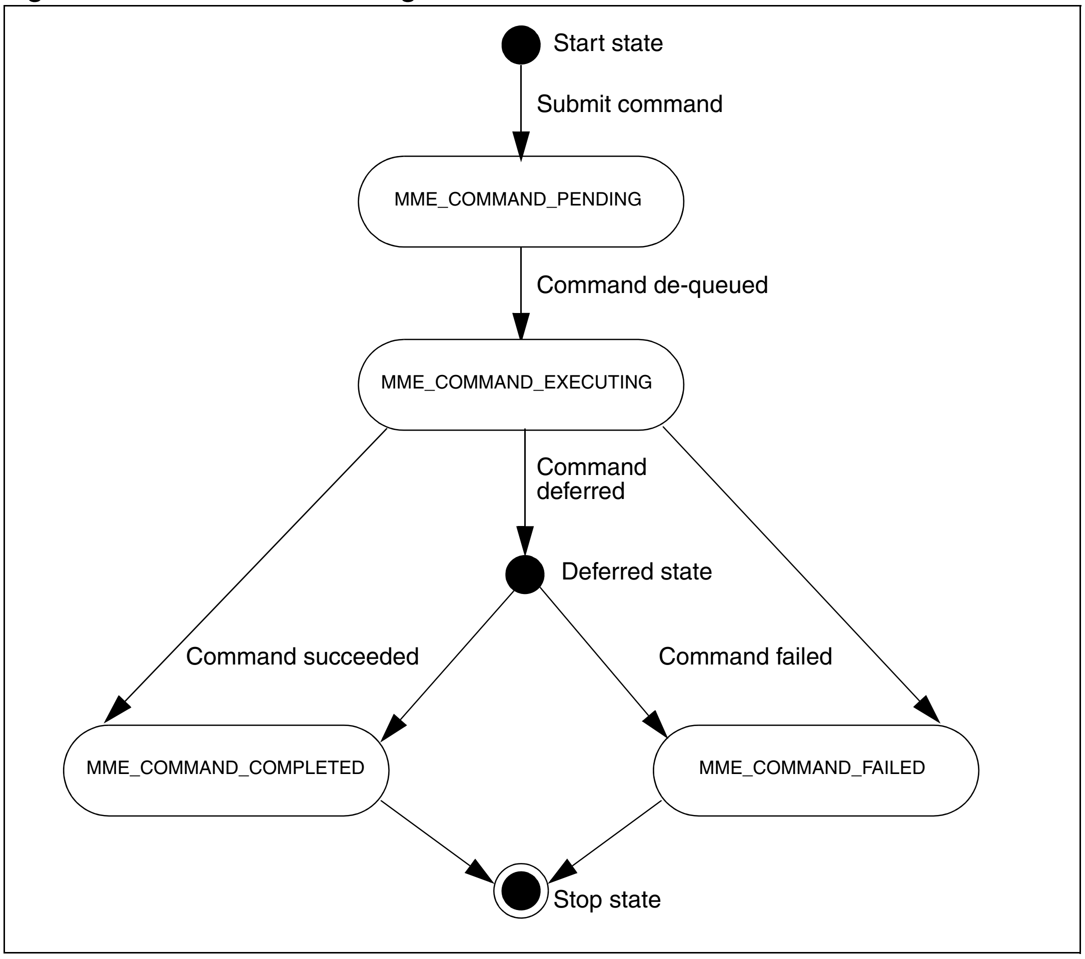
The application must fill in several fields of the `MME_Command_t` structure:

- `StructSize` - see *Section 3.1.6: Structure size on page 22*
- `CmdCode` - with the command to perform - see *Section 3.3: Managing transformer lifetimes on page 24*
- `CmdEnd` - to specify whether events such as a "command completion" cause the callback function to be called
- `Due time` - see *Section 3.1.4: Due time on page 20*

  (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 3.5: Application and transformer specific data on page 29* and *Section 4.7: Parameter passing on page 45*

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 4 on page 30*.

Figure 4. 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 3.8.2: Stream-based and hybrid operation on page 33*.

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.

3.6.1 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[b] 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.*

b. The technicalities of aborting a deferred command is discussed in detail in *Section 4.4.2: Deferred commands on page 41.*

3.7 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 3.7.1* through to *Section 3.7.3*.

3.7.1 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.

3.7.2 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 3.8.1* and *Section 3.8.2*. 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.

3.7.3 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.

3.8 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.

3.8.1 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.

3.8.2 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 4.4.3: Streaming and hybrid transformers on page 43.*

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.

3.9 Linking and loading

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

3.9.1 OS21

On OS21 the application must link against the EMBX shell and any applicable transport libraries. Additionally it must link against either the MME host library or the MME companion library.

Assuming the library search path is correctly set, add the following to the link line:

```
-lmme_host or -l mme_companion
```

Note: *The ST40 and ST200 toolsets perform linking in a single pass. For this reason the MME libraries must be included after the EMBX libraries or the application will fail to link.*

3.9.2 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.

The MME kernel module is called `mme_host.o`. This is loaded after all the EMBX modules (see *Section 9.3.2: Linking and loading on page 81* and *Appendix A: Transport configurations on page 192*). Comma-separated module parameters are used to specify previously registered EMBX transports to be used by MME. Up to four EMBX transports may be specified each with the following syntax:

```
transport<N>=name
```

The following example illustrates MME module parameters for two EMBX transports, `SHM_X` and `SHM_Y`:

```
insmod mme_host.o transport0=SHM_X,transport1=SHM_Y
```

If no transport parameters are supplied MME can only by used with local transformers.

For Linux user mode the application need only link against `libmme_host.a`[c], however, the resultant application will require the `mme_host.o` kernel module to be loaded into the kernel in order to operate correctly. Assuming the library search path is set correctly, add the following to the link line:

```
-lmme_host
```

It is meaningless for a Linux user application or a Linux kernel module to call `MME_RegisterTransport()` or `MME_DeregisterTransport()` because EMBX transports are registered when the module is loaded and deregistered when the module is unloaded. Furthermore, it is not valid for a Linux kernel module to call `MME_Init()` or `MME_Term()` because initialization and de-initialization occur automatically when the module is loaded and unloaded.

3.9.3 STLinux 2.3 and udev support

The STLinux 2.3 release enables dynamic device support using `udev`. This allows device drivers to dynamically create device files in `/dev`.

However, in order to use this facility Multicom has to make use of GPL only exported functions. This code path is not enabled by default and must be enabled by specifying `ENABLE_GPL=1` in the Multicom build options.

If you do not wish to build the Multicom modules under a GPL licence then you will need to statically create the `mme` device entry under `lib/udev/devices` in the target file system, or disable `udev` support when using the STLinux 2.3 release.

c. The user mode library `libmme_host.a` uses a device node `/dev/mme` to communicate with the MME kernel module. If this node does not exist then all API functions, including `MME_Init` will return `MME_DRIVER_NOT_INITIALIZED`.

The device node should be a character device with major number 231 and minor number 0. If it does not exist, then create it with the following command:

```
mknod /dev/mme c 231 0
```

For file systems such as NFS or HDD the device node will persist across system reboot, so this command need only be run once.

4 Writing an MME transformer

4.1 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 3: Using the MME API*.

The function to register transformers, `MME_RegisterTransformer`, is introduced in *Section 3.2.3 on page 23*. 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 6*. 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 6. 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 4.7 on page 45*.

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 4.4.2: Deferred commands on page 41*.

4.2 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)
```

4.2.1 Instantiation

`MME_InitTransformer_t` is called as a result of an application call to `MME_InitTransformer`, see *Section 3.3: Managing transformer lifetimes on page 24*.

`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

4.2.2 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.*

4.2.3 Termination

`MME_TermTransformer_t` is called as a result of an application call to `MME_TermTransformer`, see *Section 3.3: Managing transformer lifetimes on page 24*. `MME_TermTransformer_t` takes the `context` parameter described in *Section 4.2* 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.

4.3 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 3.3.1 on page 25*. 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 3.3.1: Querying the capabilities of a transformer on page 25*.

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 173 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 */
```

4.4 Processing a command

The transformer must implement a function that supports the processing any of the three types of commands introduces in *Section 3.7: Types of commands on page 32*. 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 3.6: Issuing commands on page 29*.

This function is supplied with the context pointer described in *Section 4.2: Managing transformer lifetimes on page 36*. 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 3.5: Application and transformer specific data on page 29*. 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 4.5: Aborting commands on page 44*.

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 4.4.3: Streaming and hybrid transformers on page 43*.

4.4.1 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 4.4.2: Deferred commands on page 41*) 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 43*) 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.*

4.4.2 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 154.*

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.*

4.4.3 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.

4.5 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 3.6.1: Aborting commands on page 31*.

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.

4.6 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 4.3 on page 38*) 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.*

4.7 Parameter passing

Many of the MME functions take transformer specific parameter blocks specified in MME structures as `MME_GenericParams_t`, see *Section 3.5: Application and transformer specific data on page 29*. 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.*

4.7.1 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 7 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 7. 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).

4.7.2 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);
```

4.7.3 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 8: Recommended postfixes for parameter array names* contains guidance for standard MME operations.

Table 8. 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`

4.7.4 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);
```

Part 3 RPC user guide

The RPC user guide covers:

- *Building RPC systems*
- *Interface declarations*
- *Decorating types and functions*

5 Building RPC systems

5.1 Overview

Remote Procedure Call (RPC) provides a means for a C language function implemented on one processor to transparently call a function implemented on a different processor, and obtain outputs and return values from the remotely-executed function. To achieve this, the function’s arguments are marshalled into a communications buffer and transmitted to a different CPU where the arguments are demarshalled and the remote function called. The process is repeated in reverse when the remote function completes.

The stub code required to marshall and demarshall the arguments is automatically generated by the RPC tools from the application’s C source code. Where the C language is insufficient to express the necessary marshalling and demarshalling, the C source code must be augmented by extra information called decorations, to allow the RPC tools to generate the correct stub code. Decorations may be embedded in the C source code or held in a separate file. Embedded decorations are removed automatically from the source code before it is presented to the C compiler by a process known as **stripping**.

The structure of the RPC system is N-way symmetric, that is, essentially the same software stack is run on all the CPUs involved in the RPC system. A single instance is shown in *Figure 5 on page 53*.

RPC uses an interface definition language as a scheme for describing, in a textual form, the C language functions which are implemented on one processor and are called from the other. The RPC interface definition language consists of normal ANSI C supplemented with extra information unique to RPC. These additions have a natural split between declarations and decorations.

RPC declarations provide systems level information such as the names of the arenas, the transport used to copy the arguments and the list of functions that are shared between arenas. RPC declarations are discussed in *Chapter 6: Interface declarations on page 57*.

RPC decorations provide extra type information about a specific function argument, structure member or type definition. The decorations are placed immediately before the normal type information. For example the following shows a function whose character pointer argument, `s`, has been decorated to ensure RPC copies `s` as a normal C string.

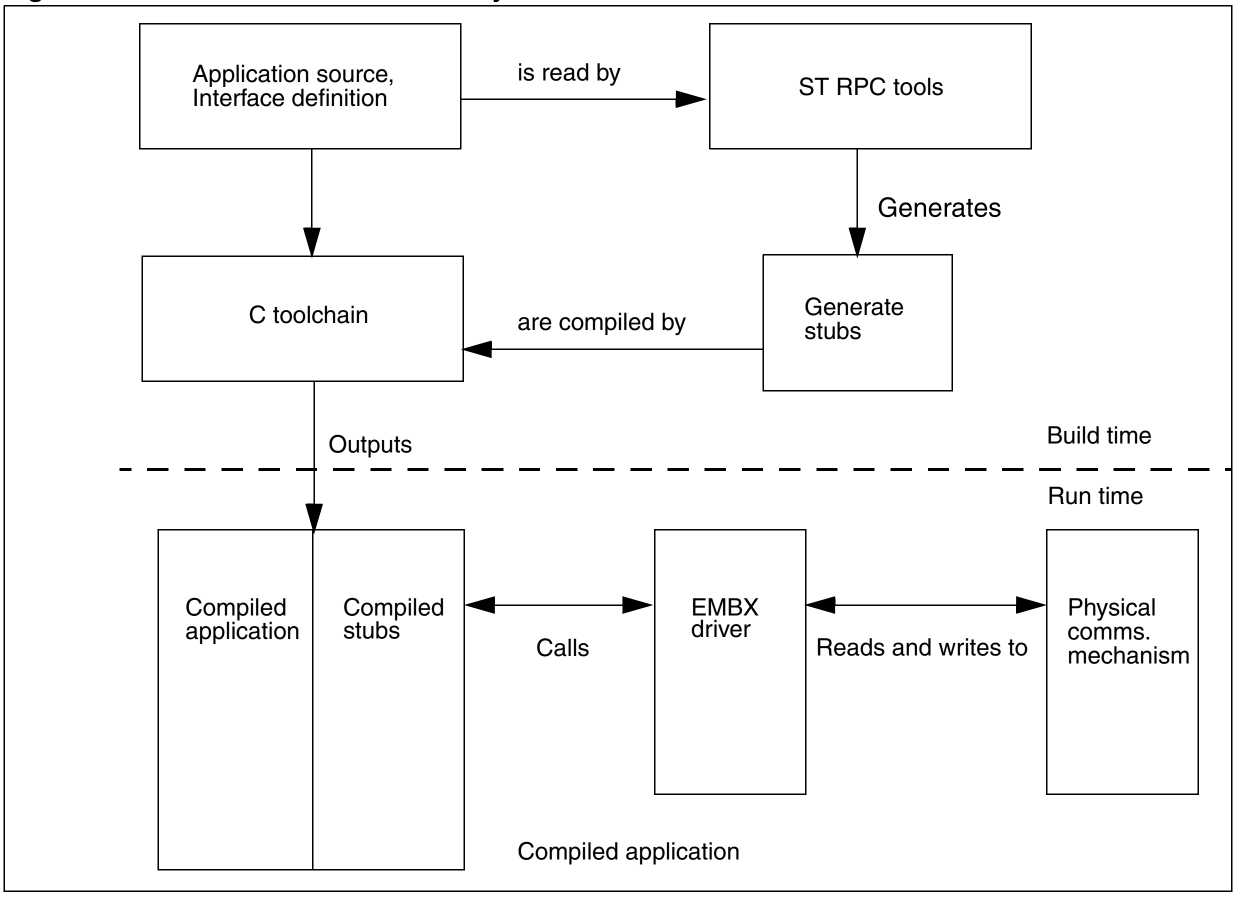
```
void putString(in.string(80) char *s);
```

Decorations extend the type system in places where C is not sufficiently expressive to describe how to copy a particular type. Decorations are described in depth in *Chapter 7: Decorating types and functions on page 60*.

The RPC generator tool reads the interface description and generates glue code called stubs which can be compiled and linked with the program in each **arena**, (see *Section 6.1: Terminology on page 57*). The generator tool is able to pass over C source code automatically extracting information. This means that changes to function arguments are automatically picked up by the generator, and descriptions of functions are not duplicated. The disadvantage of combining RPC information with the C code is that this code cannot be directly compiled. For this reason the RPC distribution contains a stripper tool to remove all RPC related information from the C source code so it can be compiled with a normal C compiler.

In some applications it may not be possible to integrate the stripper into the build system. In this case it is possible to collect all the RPC decorations and declarations into a supplementary file leaving the original C code untouched.

Figure 5. The structure of the RPC system



5.1.1 Structure of a typical system

RPC is a very flexible tool that permits the application designer to use only those parts that suit their application. This section describes one typical way to structure an RPC system. Most of the RPC examples follow this structure.

The RPC generator tools reads a single master header file that describes the structure of the system. The number of processors and the list of functions to import, is contained within this header. The header also includes a number of other headers where the functions to be imported are prototyped. The prototypes in these header files contain RPC decorations inserted into them. RPC decorations are discussed in detail in *Chapter 7: Decorating types and functions on page 60*, syntactically they appear in the same place as ANSI C type qualifiers such as `const` or `volatile`. The RPC generator uses the decorations and the systems level description in the master header to generate all the glue code required to perform RPC.

The header files containing RPC decorations are included by the other source files in the application. Although the source files contain no RPC information the header files they include do, this prevents the source files from being compiled directly. They must be stripped of RPC information after pre-processing but before they are compiled. This extra phase to compilation does not extract any RPC information, nor does it generate any code. Its sole purpose is to remove the RPC decorations from the header files so they can be processed by standard compilers.

The code output by the RPC generator is then compiled in the same way as any other code and linked into the application. The generated glue code contains implementations of the RPC functions used by the application. These functions use the EMBX communications system to call the real functions, for which they are acting as proxies.

5.2 Supplied tools

There are three tools supplied in the RPC distribution.

- The stripper, **strpcstrip**.
  This tool removes RPC information from the C source. It is usually run after the C pre-processor but before the actual C compiler. This makes it difficult to integrate with the build system and it is not normally called directly by the user.
- The generator, **strpcgen**.
  This tool parses a C pre-processed source file and uses the information to generate the stub functions.
- A specialized compiler driver to integrate **strpcstrip** into the build system, **rpccc**.
  This tool examines the C compiler's normal arguments and issues commands to run first the C pre-processor, then the stripper, and finally the C compiler.

5.3 Stripping with rpccc

Stripping with **rpccc** simply requires the build system to prefix the name of the C compiler with '`rpccc`'.

More explicitly the usage is typically one of the following:

```
rpccc sh4gcc [ <flags> ]...

rpccc sh4-linux-gcc [ <flags> ]...

rpccc st200cc [ <flags> ]...
```

Where `<flags>` are identical to those of the normal C compiler.

The tool will behave as if the C compiler has been invoked directly except that the RPC stripping tool will be invoked between the C pre-processor and the compiler. This allows RPC decorations to be removed before the code is presented to the compiler.

Note: **rpccc** *does not entirely replace the original compiler driver program. The tool only manages pre-processing and compilation; it cannot be used to invoke the linker thus using the compile only option (typically* **`-c`***) is mandatory.*

rpccc is the preferred way to ensure that all code is stripped before it is presented to the C compiler as it facilitates in place decorations eliminating redundant information from the interface definitions

All the examples (except the `posthoc` example) use **rpccc** to strip RPC information from the code. This provides many examples of typical usage.

5.4 Stripping with the C pre-processor

It is possible to eliminate the stripping stage completely from RPC build systems by carefully using macros that permit the C pre-processor to perform the stripping. This approach is slightly more complicated than using **rpccc** but will, in some cases, integrate more easily with the build system.

The basic idea is to define macros that contain the decorations when passed through the generator but that are empty when compiled normally.

For example:

```
#ifdef STRPC_GENERATOR
#define IN_STRING(x) in.string(x)
#define OUT_STRING(x) out.string(x)
#define INOUT_STRING(x) inout.string(x)
#else
#define IN_STRING(x)
#define OUT_STRING(x)
#define INOUT_STRING(x)
#endif /* STRPC_GENERATOR */

/* ... */
void printString(IN_STRING(256) char *s);
```

Typically the supporting definitions are packaged into a header file and all files that contain decorations must include that header.

5.5 Avoiding stripping

In some cases it is possible to avoid stripping entirely. The post-hoc addition of decorations discussed in *Section 7.10: Adding decorations post-hoc on page 65* permits RPC systems to be constructed without modifying any of the original source. Clearly if the original source is not modified there is no need to pass it through the stripper.

During the generation phase the information in the original source files can be supplemented with decorations located in a different file.

An example is provided that shows a system that can be compiled without using **rpccc**. This can be found in the `examples/rpc/posthoc` directory. *Section 2.3: Examples on page 15* explains how to build and run this example.

5.6 Generating RPC stubs

The generator takes input in the form of pre-processed ANSI C that has been augmented with additional decorations as described in *Chapter 6: Interface declarations on page 57*. It produces ANSI C output that when compiled and linked with an application, allows function calls between arenas.

The generator is called **strpcgen** and has the following command line:

```
strpcgen -i <input filename> -o <output filename> -a <arena tag>
```

The three parameters, `-i`, `-o` and `-a`, are mandatory. The `<arena tag>` is the arena for which stubs will be generated.

The input must already have been processed by the C pre-processor. Thus typical usage in a makefile would be:

```
disp.stubs.c : rpc_layout.h
   $(CPP) $(CPPFLAGS) rpc_layout.h -o disp.stubs.cpped
   strpcgen -i disp.stubs.cpped -o disp.stubs.c -a disp
```

The resultant C file, `disp.stubs.c`, can then be compiled as normal.

Note: The generated code will not contain any RPC decorations, however, it may include header files that do. If such header files are included the generated code must still be stripped using **`rpccc`** *or some other tool before it is compiled.*

It is possible for the generator to produce code that cannot be compiled. This is not always indicative of a problem with the generator. In general the generator will issue an error if it is unable to generate the stubs because of some problem with the RPC declarations or decorations. There are, however, expressions that pass through the generator without ever being fully parsed. These expressions are copied verbatim into the generated code; if these expressions contain errors this will result in stubs that cannot be compiled. Similarly, the headers declaration will cause the generated code to include arbitrary headers; therefore, if these headers contain errors or define macros that clash with identifiers used in the generated code the generated code will be uncompilable.

5.7 Linking, loading and configuring

The final stages of building an RPC system revolve around linking against the correct code and performing the required run-time initialization.

On OS21 the application must link against the EMBX shell and the appropriate EMBX transports. This is described in *Chapter 9: Transport specifics on page 79*.

In Linux user space there is no need to link against any extra libraries, the normal system libraries are quite sufficient. However, for the application to run correctly, the EMBX shell, the appropriate EMBX transports and the RPC micro-server (`rpc_userver.o`) must be loaded into the kernel as modules.

In Linux kernel space the application must be linked against the EMBX shell and the appropriate EMBX transport. However since linking is performed automatically by the module loader this can be achieved simply by loading those modules into the kernel.

Once the application has been linked or loaded as appropriate to the operating system, the EMBX transport and RPC stubs must be initialized before they can be used. Configuring the EMBX transports is described in *Chapter 9 on page 79*. The RPC stubs are initialized using the following function:

```
int rpcStubsInit(void *reserved);
```

The `reserved` pointer should be set to `NULL`.

The function returns 0 if it successfully initializes itself.

6 Interface declarations

6.1 Terminology

A fundamental concept of RPC is an **arena**, which describes the functions that are imported to or from the current namespace.

Each arena is identified with an **arena tag**, a four letter identifier used throughout the interface definition language (IDL). An arena also has attributes such as how it is connected to other arenas and the location of the headers where its functions are prototyped.

Typically there is one arena for each namespace in the communication system. On operating systems such as the OS21 there is only a single namespace for the entire application and therefore only one arena per processor. Linux however contains a namespace for each process and another for its kernel space. On Linux therefore, it is common for there to be more than one arena per processor.

6.2 Arena declarations

Arena declarations associate an arena tag with a particular operating system and CPU.

There may be any number of arena declarations throughout the code and each arena declaration may describe zero or more arenas. In practice however there is typically only one arena declaration describing at least two arenas.

The standard form is:

```
arenas {
   { arena-tag, os-specifier, cpu-specifier },
   ...
};
```

Each arena tag must be unique, subject to the same rules as C identifiers and no more than four characters long.

The OS specifier must be selected from: `OS_OS21`, `OS_LINUX_USER`, `OS_LINUX_KERNEL`.

The CPU specifier must be selected from: `CPU_ST40`, `CPU_ST200`.

For example:

```
arenas {
   { st40, OS_LINUX_USER, CPU_ST40 }
};
```

6.3 Transport declarations

Transport declarations associate two arena tags with the transport mechanism that these arenas use to communicate.

There may be any number of transport declarations throughout the code and each transport declaration may describe zero or more associations.

The standard form is:

```
transport {
   { arena-tag, arena-tag, transport-mechanism[, transport-name] },
   ...
};
```

Currently there is only one supported transport mechanism, `TRANS_EMBX`.

The transport name is an optional string used to select a particular EMBX transport. This is only useful if there is more than one transport registered with the EMBX driver. If the transport name is omitted then RPC will use the first (or only) registered transport.

For example:

```
transport {
    /* mstr and slv1 use the first registered transport */
   { mstr, slv1, TRANS_EMBX },
    /* mstr and slv2 use the named transport shm */
   { mstr, slv2, TRANS_EMBX, "shm" }
};
```

6.4 Import declarations

Import declarations specify which functions are imported into an arena and also note the arena in which particular functions are implemented.

There may be any number of import declarations throughout the code and each import declaration may contain zero or more imports. The standard form is:

```
import {
   { function-name, importing-arena, source-arena },
   ...
};
```

For example:

```
import {
   { remoteFunc, imp, src },
   { otherFunc, imp, src }
};
```

6.5 Header declarations

Header declarations specify which C header files should be included with the generated stubs. These are required so that type definitions, function prototypes and C pre-processor macros are defined before they are used within the generated stubs.

There may be any number of header declarations throughout the code and each header declaration may contain zero or more header lists. The standard form is:

```
headers {
   { arena-tag, { "header-filename" }, ... },
   ...
};
```

The header file name can be expressed in three forms. Each form is shown in the example below:

```
headers {
   /* #include ″foo.h″ and in alternative form, #include “bar.h″ */
   { st40, { ″foo.h″, ″\″bar.h\″″ } }
};
```

7 Decorating types and functions

7.1 Default behavior

The default behavior is applied to all undecorated data types. Additionally other decorations typically augment the default behavior rather than replacing it. This makes it particularly important to understand the default behaviors.

All undecorated data types will be treated as input parameters. Specifically this means that their values will be copied when the function is called but will not be copied back on function return. For structures and primitive types this is wholly intuitive since this exactly matches the behavior of the C language. However for array and pointer types the lack of a copy back differs significantly from C's normal behavior. In particular, programs that use pointers to simulate pass-by-reference will require a decoration if the function alters the target of the pointer.

Note: *Values returned from a function deviate from this rule, all return values are treated as output parameters.*

The mechanism used to copy a data type is dependant of its type. *Table 9* describes how each type will be treated.

Table 9. Default copying behavior for specific data types.

Type	Example	Technique for copying
Primitive type	`float`	Copied as is.
Normal pointer	`short *`	Assumed to point to a single object, it is dereferenced and the default strategy is applied again to the dereferenced object (for example, `int **` is dereferenced twice and then copied as a single primitive).
Void pointer	`void *`	It is impossible for RPC to pass an undecorated void pointer thus the generator issues an error.
Arbitrary sized array	`int a[]`	Assumed to be an array of length one. As such it is treated identically to a pointer.
Fixed size array	`int a[10]`	Each element is copied using the default strategy.
Structure	`struct foo`	The default strategy is applied to each structure member in turn.
Union	`union bar`	It is impossible for RPC to pass an undecorated union thus the generator issues an error.
Enumerated type	`enum tee`	In ANSI/ISO C an enumerated type is a form of integer, enumerations are therefore copied as primitive types.

7.2 Direction information

Direction information tells the RPC system when to copy a data type. Direction specifiers are always the first component of a decoration. In fact all decorations must start with a direction, some decorations will add optional information after the decoration.

All RPC decorations should appear immediately before the type they are decorating. For example, the following function contains a parameter `pipe` decorated with the `inout` specifier:

```
void exportedFunction(inout long *pipe);
```

Table 10 shows the direction specifiers used by RPC.

Table 10. RPC direction specifiers

Specifier	Meaning
`in`	Copy before the function is called.
`out`	Copy after the function is called.
`inout`	Copy before and after the function is called.
`transient`	Never copy this parameter.

The directional information is inherited by all elements and members of the current data type. For example, if a structure is specified **`out`** then all its members are copied after the function is called (using the default strategy). This can be overridden by decorating the structures members with alternative decorations.

For example:

```
struct outStruct {
   int mode;
   out char data[256];
};
void useOutStruct(inout struct outStruct *os);
```

Note: `transient` *differs from the other directions. Because a transient data type and its members or elements are not copied, a structure member cannot override its direction by providing an alternative.*

7.3 Strings

The default behavior for unsized character arrays and character pointers is to assume they point to a single data element (that is an array of length one). In fact in C, character pointers usually point to a string, that is an array delimited by the nil character. Strings are very common in C and for this reason a decoration specifically for strings is provided.

The standard form is:

```
direction-specifier.string(max)
```

`max` is a constant expression that evaluates to the maximum length of the string.

For example:

```
void printString(in.string(256) char *s);
```

Note: *If a string is an* `inout` *parameter, the output string cannot be longer than the input.*

7.4 Known length arrays

The default behavior for both unsized arrays and pointers is to assume they point to a single data element (that is, an array of length one). This is clearly not always the case, so RPC decorations are provided which allow the user to specify the number of elements in an array.

The standard form is:

```
direction-specifier.array([max,] len)
```

`max` is an optional constant expression that evaluates to the maximum length of the array.

`len` is an integer expression that evalulates to the actual length of the array. If `max` is ommited `len` must be a constant expression. If `max` is supplied then the expression is evaluated at run-time, allowing the quantity of data copied to be calculated dynamically.

For example:

```
/* simple fixed length array */
void getFixedData(out.array(16) int *data);


/* run-time sized dynamic array */
void printCharacters(int len, in.array(256, len) char *chars);
```

It is also possible for the `len` expression to contain the special pseudo-identifier `__struct__`. This pseudo-identifier refers by value to the current structure (if there is one). This allows one structure member to be dynamically sized by the value of a different structure member.

For example, in the following structure the size of data, pointed to by `sample` is based on the value of `length`:

```
typedef struct pcm {
   int length;
   in.array(48000, __struct__.length) short *sample;
} pcm_t;
```

7.5 Delimiter terminated arrays

There are some types of array for which it is not possible (without making a function call) to construct an expression that evaluates to the length of an array. In these cases it may be possible to construct a boolean expression that identifies the final element in an array. A C string is an example of such an array although as we have seen RPC provides a special case decoration for C strings.

The standard form for such a decoration is:

```
direction-specifier.termarray(max, expr)
```

Like the `array` strategy `max` is a constant expression that evaluates to the maximum length of the array. In this case however it is not optional.

`expr` is a boolean expression that becomes `true` for the last element of the array. Like the `array` strategy the expression can make use certain psuedo-identifiers. `__element__` is replaced by the current element in the array (by value), and `__count__` is replaced by the index of the current element.

For example:

```
/* an alternative (and usually slower) way to pass a C string */
void printString(in.termarray(256, __element__ == '\0') char *s);


/* an alternative (and slower) way to pass an array */
void printCharacters(
      int len,
      in.termarray(256, __count__ >= len) char *chars);


/* a more realistic example */
struct playlist_t {
   int length;
   in.string(256) char trackName[256];
   in.string(256) char bandName[256];
}

void playPlaylist(
      in.termarray(64, __element__.length == 0) playlist_t *pl);
```

7.6 Opaque pointers

An opaque pointer is assumed to be a valid pointer on the originating machine but one that the receiving machine will never dereference. Such a pointer will only have meaning if the recipient passes it back to the originating machine as a function argument. Opaque pointers are most often used when providing parameters for asynchronous callbacks that are passed back to the originator.

The standard form is:

```
direction-specifier.opaque
```

Note: Like the `transient` *direction specifier the opaque strategy cannot be overridden by decorations with structure members; since an opaque pointer is never dereferenced these structure members are not copied.*

For example, the following function will rearrange an array of **`codec`** requests into priority order and uses an opaque pointer to avoid having to copy the data itself:

```
struct codecRequest {
   int command;
   int streamID;
   inout.opaque short *data;
} codecRequest_t;
void prioritizeTransforms(int n, in.array(64, n) codecRequest_t
*r);
```

7.7 Pointers to shared memory

On platforms that possess shared memory it is possible to pass pointers to shared memory through the RPC system. Such pointers are altered such that they point to the same part of memory even if the pointer must point to a different address on each processor.

Note: Using shared memory is a means to trade portability for performance. Since the contents of shared pointers are not copied they can be efficiently transferred between processors but *this requires the system to assume that the RPC/EMBX architecture is based on the CPUs sharing memory.*

The standard form is:

```
direction-specifier.shared
```

For example:

```
void playPCM(int len, in.shared short *pcm);
```

For the `shared` strategy to be useful it requires the user to have some means to allocate shared memory. `EMBX_Alloc` is typically used to achieve this.

Note: The RPC system will not make any attempts to maintain cache coherency when pointers are shared. If it is possible for the shared data to be held in either CPU's cache then this must be flushed before it is read (or written) by the other CPU.

7.8 Type definitions

RPC decorations can be applied to type definitions as well as to function arguments and structure members.

When a particular type is used by a large number of functions it is simpler and more maintainable to apply the appropriate decorations to the type rather than to the functions themselves.

For example:

```
typedef inout.string(4096) char *string_t;
```

7.9 Function pointers and callbacks

Pointers to functions are specially treated by RPC. As the pointers are transferred through the RPC system their values are altered in a particular manner such that values remain sensible for imported functions. Functions that are not imported to or from a particular arena will be converted to `NULL`.

Specifically, assume a function is imported to a particular arena and a pointer to its stubs function is passed using RPC. The pointer is automatically converted to a pointer to the implementation of that function.

Similarly if a function is exported from a particular arena then a pointer to its implementation will be converted into a pointer to the stub function when it is passed.

Function pointers are handled automatically without any decorations although when implementing callback style interfaces the opaque strategy is particularly useful.

For example, the following system allows the arena `cb` to register `callback` using `registerCallback`, thus permitting arena `reg` to issue a callback at some later point:

```
void registerCallback(void (*cb)(void *), in.opaque void *d);
void callback(in.opaque void *d);


import {
   { registerCallback, cb, reg },
   { callback, reg, cb }
};
```

7.10 Adding decorations post-hoc

In some source bases it is thought inappropriate to modify the original source code by adding decorations. If the original source code is supplied by a third party or forms part of the system headers it would be difficult to add decorations without affecting other programs. Similarly limitations in the build system make it difficult to strip the decorations from the source code.

In order to cope with such source bases, RPC allows decorations to be added to declarations that have already been presented to the generator.

The standard form is:

```
rpc_info {
   declaration-or-function-prototype
};
```

Declarations and function prototypes contained within the `rpc_info` block should be identical to the original declarations, with the exception that these declarations may contain RPC decorations.

For example, the declaration below shows how to add decorations to two functions from the standard C libraries.

Note: That this is purely an example, in general trying to export standard C functions will result in symbol clashes at link time.

```
#include <stdio.h>


rpc_info {
   int puts(in.string(256) const char *s);

   /* we assume stdout, stderr etc. will be initialized using opaque
    * pointers before fputs is called.
    */
   int fputs(in.string(256) const char *s, in.opaque FILE *stream);
};
```

Part 4 EMBX user guide

The EMBX user guide covers:

- *Using the EMBX API*
- *Transport specifics*

8 Using the EMBX API

8.1 Overview

The EMBX API provides an interface through which all communication is accomplished, regardless of the underlying hardware mechanism being used to transfer data.

EMBX is supplied with a number of **transports** which control in detail the method of communication. Generally, a transport can manage communication between any number of CPUs(a) and there can be any number of transports active at one time.

The application can create named **ports** within a transport. A port is used to receive communication events and follows a multi-drop model. Specifically a port provides for single direction communication and can be connected to any number of data sources. Two way communication is achieved by logically pairing ports together at the application level. In general there are no fixed limits on the number of ports within a transport(a).

Before an application can send a message to a port it must make a **connection** to that port.

Communication events can take three forms. The first is a message event; messages are contained in buffers allocated by the application from a particular transport's memory pool. The other forms relate to distributed objects. A distributed object is a block of memory allocated by the application and registered with a transport. An object update event is used when changes to the contents of that object need to be propagated to other processors. One form of object update informs the process that the object has been updated, the other does not.

Although EMBX allows multiple transports to be active at the same time it does not support cross transport communication directly. Applications that require such communication must perform this manually.

8.1.1 The EMBX shell

The EMBX API is implemented as a library or Linux kernel module called the EMBX shell. This library provides high level error checking, handle management and operating system abstraction services.

Each different transport type is provided in its own library or kernel module. An application for a single binary environment such as OS21 picks which transport libraries it needs to link against. In the Linux kernel environment, transport modules can be loaded and unloaded dynamically.

The shell defines an internal interface between itself and a transport implementation, whose definition is outside the scope of this document. New transports can be produced, without modification of the EMBX shell or the external API, by developing code to implement this internal interface.

a. A specific transport may have limits on the number of CPUs or ports associated with it.

8.2 Initialization

Initialization is divided into three distinct phases. These are:

- specifying which transports should be created and their configuration
- initializing EMBX
- opening a transport

Due to the variety of system environments supported by EMBX the first two steps are not strictly ordered. That is, it is possible to initialize EMBX before specifying any transport configurations.

Typically an OS21 based application will register transports before initializing EMBX simply because this allows a more natural split between system configuration code and generic application code.

In a Linux environment it is not possible to register transports before EMBX is initialized. Transports can only be registered once the kernel module implementing them has been loaded into the kernel and transports cannot be loaded into the kernel until the EMBX shell has been loaded. The EMBX shell automatically initializes itself as it loads, thus ordering is imposed by the inter-dependencies of the kernel modules.

8.2.1 Registering transport factories

Transports are created using transport factories, which consist of a factory function and a configuration structure. A particular transport implementation will provide a public header file, which contains prototypes for the factory functions and defines the configuration structures it supports. A single transport implementation may export any number of factory functions in order to support different variations of the transport or the underlying hardware platforms.

A transport factory is registered with the EMBX driver using:

```
EMBX_ERROR EMBX_RegisterTransport(
   EMBX_TransportFactory_fn *fn,
   EMBX_VOID                *arg,
   EMBX_UINT                 arg_size,
   EMBX_FACTORY             *handle);
```

This call makes a copy of the configuration argument, which leaves the application free to destroy the original configuration data after the call. The call returns a factory handle that can be used to unregister the factory at a later time. Within the Linux kernel environment, registering a transport factory increases the module reference count on the EMBX API module ensuring the module cannot be unloaded until all transport factories have been unregistered.

Note: *A transport factory function attempts to create a transport based on the configuration argument provided. The factory checks, as far as it is possible, that the configuration requested is correct for the system the code is running on. Should this check fail then the transport is not be created.*

However, if the transport is not created, the call to `EMBX_RegisterTransport` *still succeeds as the transport has been successfully registered. This is because the initialization of a registered transport is actually deferred until* `EMBX_Init` *is called.*

The transport query calls provide a means to detect failure to create a registered transport (see Section 8.3.1 on page 69).

8.2.2 Initializing EMBX

The EMBX driver is initialized using the following function:

```
EMBX_ERROR EMBX_Init(void)
```

The EMBX driver must be loaded and initialized on each processor that wishes to participate in the system. With the exception of `EMBX_RegisterTransport` and `EMBX_UnregisterTransport`, no API can be called until the driver is initialized.

In a system where multiple processes wish to use the driver, it is permissible for each process to call `EMBX_Init`. The first call performs initialization, returning `EMBX_SUCCESS` if no error occurs. Any subsequent calls simply return `EMBX_ALREADY_INITIALIZED`. That is, a second call to `EMBX_Init` does not re-initialize an already initialized driver.

Note: Calls to `EMBX_Init` *are not counted. Thus particular care must be taken de-initializing the driver when sharing the EMBX between multiple processes.*

On Linux, the EMBX is automatically initialized when the EMBX shell module is loaded and it is automatically de-initialized then the module is unloaded. It is therefore unnecessary for other modules using the API to call either `EMBX_Init` or `EMBX_Deinit`.

`EMBX_Init` causes all registered transport factories to be called with their configuration argument. Those transports that are successfully created are available for use once the call completes.

Note: It is possible for no transport factories to be registered at this point, or for all the factories to fail; in which case no transports will be available.

8.3 Transports

8.3.1 Querying transports

In most cases, an application will know all of the transports it needs to use. However, in order to allow greater flexibility in an application's architecture, two API mechanisms can be used to query available transports.

```
EMBX_ERROR EMBX_FindTransport(EMBX_CHAR *name, EMBX_TPINFO *tpinfo)
EMBX_ERROR EMBX_GetFirstTansport(EMBX_TPINFO *tpinfo)
EMBX_ERROR EMBX_GetNextTransport(EMBX_TPINFO *tpinfo)
```

`EMBX_FindTransport` is used to lookup a transport whose name is already known. This is useful to check that a registered transport was correctly initialized.

The other two functions are used to iterate through the list of available transports

All three functions populate a transport information structure which contains the elements described in *Table 11 on page 70*.

Table 11. **Transport information structure elements**

Name	Type	Description
`name`	`EMBX_CHAR *`	ASCII string name of the transport, limited to `EMBX_MAX_TRANSPORT_NAME` characters in length.
`isInitialized`	`EMBX_BOOL`	Is this transport in an initialized state?
`usesZeroCopy`	`EMBX_BOOL`	Flag indicating the transport's copy semantics for buffer sends.
`allowsPointer Translation`	`EMBX_BOOL`	Flag indicating if buffer pointers belonging to this transport can be translated to and from opaque values and be safely transmitted as part of a message.
`allowsMultiple Connections`	`EMBX_BOOL`	Flag indicating if this transport allows ports to be created that accept multiple connections.
`maxPorts`	`EMBX_UINT`	The maximum number of ports the transport supports, zero indicates no upper limit, given the available resources in the machine.
`nrOpenHandles`	`EMBX_UINT`	The number of currently open transport handles to this transport.
`nrPortsInUse`	`EMBX_UINT`	The number of ports currently active on a transport.
`memStart`	`EMBX_VOID *`	Beginning of the transport's memory pool.
`memEnd`	`EMBX_VOID *`	End of the transport's memory pool.

Note: The grayed items in Table 11 are valid only once a transport is initialized by a successful call to `EMBX_OpenTransport` *(see Section 8.3.2); this is indicated by the* `isInitialized` *flag in the structure.*

It is also possible to extract the transport information structure for an open transport using the following function:

```
EMBX_ERROR EMBX_GetTransportInfo(EMBX_TRANSPORT tp, EMBX_TPINFO *tpinfo)
```

8.3.2 Transport open and close

Transport handles can be opened and closed using the following functions:

```
EMBX_ERROR EMBX_OpenTransport(EMBX_CHAR *name, EMBX_TRANSPORT *tp)
EMBX_ERROR EMBX_CloseTransport(EMBX_TRANSPORT tp)
```

Application code requires a transport handle to call any of the communication or allocation functions. In a system containing multiple processes, each process wanting to use the transport should open a transport handle. If successful `EMBX_OpenTransport` will supply a transport handle which is passed in other API calls to identify which transport is to be used.

If this is the first time a transport handle is opened the call initializes the transport. This includes connection to whatever underlying mechanism is being used to implement the transport. This may involve performing a handshake with partner devices. Thus there is no guarantee that this call will return control to the controlling application; if the partner devices are incorrectly configured it is possible for this call to block forever waiting for a communication that cannot happen. Any initialization time-outs must be managed by the application from another thread.

A transport can be closed only when there are no open port handles associated with it; applications must explicitly close all of their open port handles before closing a transport handle. At this point the driver releases the resources associated with the handle. Using it after the call completes results in undefined behavior.

Note: When a buffer is allocated it implicitly stores a copy of the transport handle within the buffer. For this reason all buffers associated with a transport should be deallocated before that transport is closed.

8.4 Buffer management

The EMBX API provides an efficient **mailbox** communication system. This is a style of communication rather like writing a letter, the application gets a message buffer, fills it with information and then posts it to a destination port. Once the buffer has been sent the sending application no longer owns nor can use the buffer.

Each transport allocates memory of a type that is appropriate for that transport mechanism. Depending on the transport type, the memory used for buffers may have to be:

- in contiguous physical memory (not part of a virtual address space)
- in a part of the address map that can be accessed by multiple processors in the system
- uncached

Some transports choose to implement their own memory management mechanism others just use the system allocator. Transports requiring memory with specific properties are likely to use their own allocator; such a transport's configuration usually specifies the amount of memory available and possibly even the physical address range that can be used by the transport.

The lifetime of buffers is managed by the transport, allowing for safe semantics when closing ports with outstanding messages and avoiding memory leaks.

8.4.1 Buffer allocation and release

The following functions allow buffers to be allocated or deallocated from the buffer pool belonging to a transport:

```
EMBX_ERROR EMBX_Alloc(EMBX_TRANSPORT tp, EMBX_UINT size, EMBX_VOID **buffer)
EMBX_ERROR EMBX_Free(EMBX_VOID *buffer)
```

Allocation operates in a similar manner to standard operating system memory allocation mechanisms.

`EMBX_Free` does not require the transport handle since this is stored internally. However, just as in conventional memory management schemes, passing garbage or an already released pointer is likely to corrupt the system.

8.4.2 Querying buffer size

The following interface allows the applications to determine the underlying size of a buffer:

```
EMBX_ERROR EMBX_GetBufferSize(EMBX_VOID *buffer, EMBX_UINT *size)
```

This can be used on buffers returned from `EMBX_Receive` (see *Section 8.7.1: Receiving message and object events on page 75*) to find the actual size of a buffer rather than just the number of valid bytes of data.

Note: Some transports will actually allocate a buffer larger than that requested through `EMBX_Alloc`*; hence the returned size may not be identical to that passed to the call that allocated the buffer.*

8.5 Distributed objects

The mailbox style of communication is well suited for command and control type applications; but there are situations where it is not appropriate, including:

- communicating data contained in memory that cannot be managed by EMBX
- communicating a small region of data from a larger contiguous buffer
- specifying the exact location, on the destination, where data is to be sent to

The first two situations can be solved by first copying the data into an EMBX allocated message buffer. However, since unnecessary copying has an impact on performance this is not ideal. The last situation is also driven by wanting to avoid an additional data copy on the destination, usually coupled with the fact that the destination memory is being managed by another component in the system.

In a shared memory system this could be achieved by manipulating pointers and bypassing EMBX completely for the data path. This would, however, be likely to result in difficult to port applications and, unless very carefully implemented, subtle and hard to find errors.

Fortunately the EMBX provides **distributed objects**, a means to efficiently cope with the problems outlined above. A distributed object is a fixed size region of memory, managed by an application on a specific CPU, which is registered with an EMBX transport and identified by an object handle. The handle, while unique within the transport, is common to all processors participating in the transport. This allows distributed objects to be passed in message buffers, RPC arguments as well as through object update events. This provides flexibility to the application programmer allowing them to select the data flow that best suites their purposes.

8.5.1 Distributed object registration

The following functions are used to register or deregister distributed objects:

```
EMBX_ERROR EMBX_RegisterObject(
         EMBX_TRANSPORT tp,
         EMBX_VOID     *object,
         EMBX_UINT      size,
         EMBX_HANDLE   *handle)
EMBX_ERROR EMBX_DeregisterObject(
         EMBX_TRANSPORT tp,
         EMBX_HANDLE    handle)
```

Any area of memory that can be seen by the application may be registered with the transport. EMBX does not know about, nor can it manage the lifetime of the registered memory, thus it is the registering application's responsibility to ensure that this memory continues to be valid until that object is deregistered.

It is sometimes useful to register memory that has been allocated using `EMBX_Alloc`. Such memory is carefully placed in memory in order to optimize memory transfers (typically by facilitating zero copy updates). Distributed objects are designed to use the fastest copying technique available but this can be enhanced through careful memory allocation at the application level.

Note: *If memory returned from `EMBX_Alloc` is registered in whole or in part as a distributed object it must not be used as a normal message buffer; when a buffer is transferred in that manner the ownership is transferred potentially freeing local copies.*

If the registered memory is not addressable by other processors within the transport then the transport allocates one or more shadow objects. These shadows contain undefined values and must be updated before they are read from.

The same piece of memory or any portion of it can be registered multiple times with the same transport, as well as being registered with different transports at the same time. Each registration produces a new object handle and possibly a new set of shadow objects.

When an object is deregistered all shadow objects are released and the handle becomes invalid. Use of handles after deregistration has an undefined effect and must be avoided.

8.5.2 Querying distributed object properties

This function can, given a valid distributed object's handle, query its location:

```
EMBX_ERROR EMBX_GetObject(EMBX_TRANSPORT tp,
    EMBX_HANDLE    handle,
    EMBX_VOID    **object,
    EMBX_UINT     *size)
```

Depending on the underlying transport implementation this supplies a pointer to either the original registered pointer or to a locally valid shadow copy. Therefore this call makes no guarantees about the validity of the contents of the memory, if the shadow object has never been updated its contents are undefined.

Note: *If the handle refers to an object that does not yet have a physical representation on the calling CPU then appropriate memory is allocated; this will fail if insufficient memory is available.*

8.6 Ports

8.6.1 Obtaining port handles

A port is an endpoint for communication; messages are sent to and read from ports. For this to happen both the sender and receiver must have a reference to the same port, called a port handle, of type `EMBX_PORT`. The following function can be used to create a port handle capable of receiving:

```
EMBX_ERROR EMBX_CreatePort(
         EMBX_TRANSPORT tp,
         EMBX_CHAR *name,
         EMBX_PORT *port)
```

This handle cannot be used to send data to the port and any attempt to do so will result in `EMBX_INVALID_PORT` being returned. The created port is bound to the given name, which must be unique within a particular transport, so that it can be a target for a remote connection.

In order to send data to a port, a connection must be made to a bound port name; this returns a port handle that can be used to send data to the port. This is done using one of the following calls:

```
EMBX_ERROR EMBX_Connect(
         EMBX_TRANSPORT tp,
         EMBX_CHAR *portName,
         EMBX_PORT *port)

EMBX_ERROR EMBX_ConnectBlock(
         EMBX_TRANSPORT tp,
         EMBX_CHAR *portName,
         EMBX_PORT *port)
```

`EMBX_Connect` returns an error if the port name is not found while `EMBX_ConnectBlock` blocks until a port becomes bound to the name. Both functions return immediately if the port has already been bound. This provides a synchronization mechanism between the two participants and removes any requirement for one to start before the other.

The port handle returned from both connect calls cannot be used to receive data from the port and any attempt to do so results in `EMBX_INVALID_PORT` being returned.

Note: *Some transports may only permit one connection to a port to be open at one time. Thus both the above calls can fail if the port already has a connection and the port was created in a transport that only permits single connections.*

An application can connect to any port in any transport that it can see, including ports that it created. This allows communications from the local processor to be handled in an identical manner to remote processors.

8.6.2 Closing ports

Closing a port handle, returned from one of the connect calls, logically breaks the connection between sender and receiver. If the destination port was created in a transport that allows only single connections, then this port becomes unconnected and a future call to `EMBX_Connect` or `EMBX_ConnectBlock` will be able to successfully connect to it. Any messages that have been successfully sent to the destination, but not been received using `EMBX_Receive` or `EMBX_ReceiveBlock`, are not affected by the closure of this side of the communication.

Closing a port handle, returned from `EMBX_CreatePort`, destroys the port and unbinds the port's name. All connections to the port are invalidated but not closed; any attempt to communicate through an invalid port handle returns `EMBX_INVALID_PORT`. If the application receives this return value it should close the port handle in order to free the local resources allocated to the connection. Once the connections have been invalidated, any blocking receivers are awoken, causing them to return an error, any pending messages on the port are returned, unread, to the free pool and the port's local resources are released.

Both types of port handle are closed using the following function:

```
EMBX_ERROR EMBX_ClosePort(EMBX_PORT port)
```

As with transport handles, once closed, using a port handle results in undefined behavior.

`EMBX_ClosePort` alone does not allow the safe cleaning up of local threads that do blocking reads on the port. If the port were closed immediately before a thread makes its next call to `EMBX_ReceiveBlock` this would result in undefined behavior since the handle would have been closed. The solution to this is to invalidate the handle without closing it. For receive ports this can be acheived using the following function.

```
EMBX_ERROR EMBX_InvalidatePort(EMBX_PORT port)
```

This signals to remote connections that the port is invalid in the same way as `EMBX_ClosePort`. It also interrupts tasks waiting on `EMBX_ReceiveBlock`, causing them to return an error indicating that the port handle is now invalid. Finally the port handle is marked invalid such that all future calls using the handle, with the exception of `EMBX_ClosePort`, return `EMBX_INVALID_PORT`. The application should synchronize the termination of all the threads, once they have detected the port has become invalid, then finally call `EMBX_ClosePort` to clean up the port's resources.

8.7 Send and receive

8.7.1 Receiving message and object events

Sending a message or an object update to a connection port posts an event on the receive port's queue. An event is received from a port by an application using one of the following:

```
EMBX_ERROR EMBX_Receive(EMBX_PORT port,
                        EMBX_RECEIVE_EVENT *event)
EMBX_ERROR EMBX_ReceiveBlock(EMBX_PORT port,
                             EMBX_RECEIVE_EVENT *event)
```

The first call does not block if no event is available, instead it returns an error. The second call blocks until an event arrives or until it is interrupted, for instance by the port closing down. When an event is available its details are placed in the event structure supplied by the caller. This structure contains the following information:

- the event type, currently either `EMBX_REC_MESSAGE` or `EMBX_REC_OBJECT`,
- the handle of the object for an object event,
- a pointer to the beginning of the message buffer or object's memory,
- the offset from the above pointer to the first valid byte of data for this event;
- the number of valid bytes (size) of data for this event.

For a message event the offset is always zero, for an object update event the offset and size specify the region of the object that is known to contain valid information. The event structure does not contains the actual size of the underlying message buffer or distributed object. These can be queried using `EMBX_GetBufferSize` or `EMBX_GetObject` respectively.

When a message buffer is received, ownership of the buffer is transferred with it. Thus the receiving application is responsible for deallocating the buffer, either explicitly, by freeing it or implicitly, by sending the buffer to another port. Put in different terms this means that the application must treat any message buffer received from these calls as though it has been obtained using `EMBX_Alloc`.

Note: *The ownership of a distributed object is never transferred when the object is updated; it can only be deregistered on the processor it was registered. The application is responsible for ensuring it is not used after deregistration.*

8.7.2 Sending messages

The following function is used to send a message buffer:

```
EMBX_ERROR EMBX_SendMessage(EMBX_PORT  port,
                           EMBX_VOID *buffer,
                           EMBX_UINT  size)
```

This call delivers the given buffer to the destination port represented by the port handle. Only message buffers that have been allocated from the same transport as the port handle can be sent. This means that passing a message buffer received from one transport cannot be directly sent to a different transport; the application must use distributed objects, which can be registered with multiple transports, or simply copy the message if inter-transport routing is required.

Send is an asynchronous operation; it may return before the message has been received by the destination port. The send operation is reliable, that is any checking or retry mechanism is handled transparently by the transport and ordering is preserved.

The transport only transfers the first `size` bytes of the message buffer in order to make most efficient use of the transmission medium. Nevertheless the size of the underlying buffer is maintained irrespective of the value of size; therefore providing the original buffer is suitably large, it is quite possible for replies to be larger than the original message.

When the send call completes, the sender no longer has ownership of the message buffer and must not access the buffer memory again; it is as if `EMBX_Free` had been called. However, in transports that support zero-copy messages the buffer will simply have changed ownership.

8.7.3 Sending and updating distributed objects

Changes to the contents of an object are made visible to other applications in the system by explicitly sending an update to ports belonging to those applications. The following functions provide the means to do this:

```
EMBX_ERROR EMBX_SendObject(
         EMBX_PORT   port,
         EMBX_HANDLE handle,
         EMBX_UINT   offset,
         EMBX_UINT   size)
```

```
EMBX_ERROR EMBX_UpdateObject(
         EMBX_PORT   port,
         EMBX_HANDLE handle,
         EMBX_UINT   offset,
         EMBX_UINT   size)
```

The difference between the two is that `EMBX_SendObject` posts an `EMBX_REC_OBJECT` event to the specified port, whilst `EMBX_UpdateObject` does not. Both calls ensure that the specified portion of the object is updated such that it can be read by the owner of the receive port.

Note: If the distributed object's memory is directly addressable by the sender then no copying takes place which in fact means that `EMBX_UpdateObject` *does no work at all. However, the calls should still be made to ensure correct operation on copying transports.*

The calls specify the region of the distributed object that will be updated, with an offset from the beginning of the object's memory and the size of the region. This allows very tight control of the granularity of data copies if this is required.

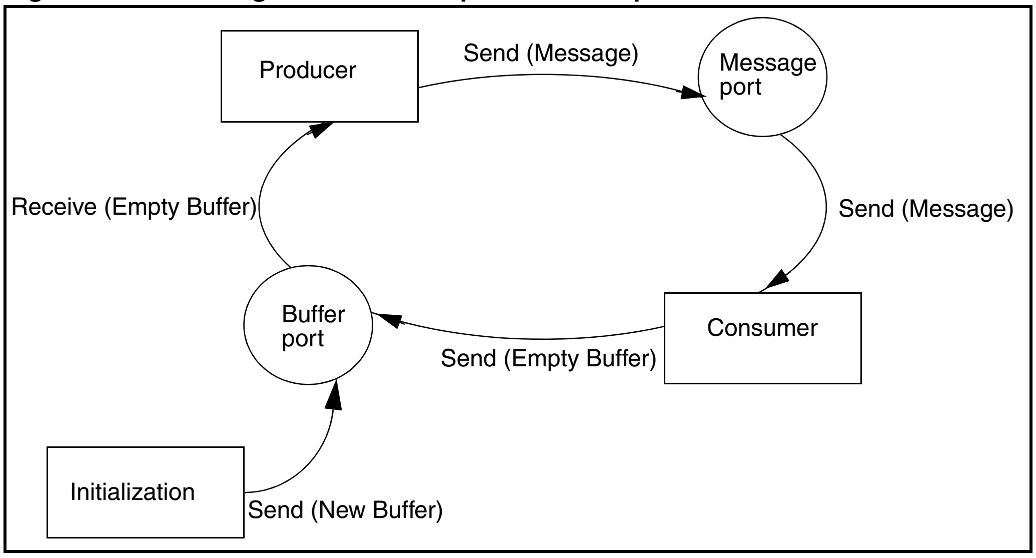
A region that is zero sized results in shadow storage being allocated on the destination if required. This mechanism allows `EMBX_SendObject` to be used as a means for passing write permission to an object between components without requiring the use of a control message or copying useless data.

The intended use for `EMBX_UpdateObject` is to allow multiple objects to be updated, without having a task on the receiving side having to deal with any events. Typically the updates would be followed by a single control message or RPC operation that specifies what to do with the now updated objects.

8.7.4 Usage example: buffer pool

An application could use the behavior of `EMBX_SendMessage` and `EMBX_Receive` to efficiently implement buffer reuse management from a fixed size pool, see *Figure 6*.

Figure 6. Reusing buffers from a pre-allocated pool



Here we see a producer/consumer model, communicating with a finite number of fixed size message buffers. Two communication ports exist: the buffer pool port, which is the source of free buffers for the producer and the message port which is destination for messages sent to the consumer. The buffer pool port has two active connections to it; hence, the transport must allow multiple connections. The initialization creates a number of buffers and uses `EMBX_SendMessage` to send them all to the buffer pool port. For the producer to send a message to the consumer it must first receive an empty buffer from the buffer pool. When the consumer has finished dealing with a message, it sends the message buffer back to the buffer pool port, using a data size of zero so that no physical copy will happen on a copy based transport. The rate of production of new messages is naturally regulated by the number of buffers circulating around the system and the relative speeds of the producer and consumer. The producer blocks if it is too fast, waiting for an empty buffer to arrive and the consumer blocks if it is too fast waiting for a message to process.

8.8 Transport and EMBX shutdown

EMBX is shutdown by calling:

```
EMBX_ERROR EMBX_Deinit(void)
```

This:

- invalidates all open port and transport handles (see *Section 8.3: Transports on page 69* and *Section 8.6: Ports on page 73*)
- interrupts any tasks waiting on `EMBX_OpenTransport`, `EMBX_ConnectBlock` or `EMBX_ReceiveBlock`
- waits for all transport handles to be closed by the application
- shuts down all currently active transports and resets any driver structures

At this point the driver can be re-initialized with another call to `EMBX_Init`. In the Linux kernel implementation this is called when a request to unload the API module has been made by the user. If an application has been written such that `EMBX_Deinit` can be called, while tasks are still using active transports, it must test the return values from API calls in order to determine that the system has become invalid and is waiting to shutdown. When this is detected it should release message buffers, close ports and finally close all of its transport handles. If an application fails to close open ports and transport handles then the call to `EMBX_Deinit` blocks indefinitely.

If multiple processes call `EMBX_Deinit` before it completes then all of them block until it does complete or fail; in this case all of the blocked calls return the same result, whatever that might be.

Shutting down EMBX with `EMBX_Deinit` does not un-register transport factories. Therefore if `EMBX_Init` is called again after the driver has been shutdown the factories are called once more and transports created. In order to clean up the copy of the factory configuration argument, factories should be unregistered after the driver has closed using:

```
EMBX_ERROR EMBX_UnregisterTransport(EMBX_FACTORY handle)
```

Note: In a Linux kernel environment this decrements the module usage count for the API module. Failing to un-register transport factories prevents the API module from being unloaded from the kernel, as this can only happen if the usage count is zero.

`EMBX_UnregisterTransport` can also be called while EMBX is still active. The behavior in this case depends on whether or not a transport was successfully created from the given factory handle. If no such transport exists then the record of the factory is simply removed; however if a transport does exist it is closed down, using the same sequence of events used in `EMBX_Deinit`. Hence, like `EMBX_Deinit`, `EMBX_UnregisterTransport` can block indefinitely waiting for the transport to be closed.

Note: In a Linux kernel environment, blocked kernel operations may get woken up due to a signal being delivered to an associated user process. This occurs when a user process issues a system call, to another kernel module, that then calls EMBX. In this case, if the `EMBX_Deinit` *or* `EMBX_UnregisterTransport` *operation has not successfully completed it is postponed and the call returns* `EMBX_SYSTEM_INTERRUPT`*.*

9 Transport specifics

9.1 Introduction

Each transport within the EMBX must provide one or more factory functions to create an instantiation of that transport. A factory function is bound to a name using `EMBX_RegisterTransport` (see *Section 8.2.1: Registering transport factories on page 68*). Also bound is a factory specific data structure used to configure the transport.

This chapter provides an overview of the unique features of each transport together with a detailed description of how each transport is configured.

9.2 EMBXMailbox

`EMBXMailbox` is not actually a transport in its own right. In fact `EMBXMailbox` is simply a utility library used by other transports with the EMBX system.

`EMBXMailbox` is responsible for managing the use of the hardware mailbox resources. The hardware mailbox is a peripheral present on STMicroelectronics' shared memory platforms; the mailbox is used for interrupt generation and to manage mutual exclusion in the absence of bus-locked read-modify-write operations. `EMBXMailbox` provides a means to allocate the resources of the hardware mailbox as well as simple functions to manipulate resources once they are allocated.

Whilst `EMBXMailbox` provides a complete API to manipulate the hardware mailboxes, most of this API is only expected to be used by the other EMBX transports. The API is described in *Chapter 10: Function descriptions on page 87* and the main functions intended for general use are discussed in this chapter.

The `EMBXMailbox` functions of general interest are as follows.

```
EMBX_ERROR EMBX_Mailbox_Init(void)
EMBX_ERROR EMBX_Mailbox_Register(
       void *pMailbox,
       int intNumber,
       int intLevel,
       EMBX_Mailbox_Flags_t flags)
EMBX_VOID EMBX_Mailbox_Deregister(void *pMailbox)
```

`EMBX_Mailbox_Init` is used to initialize the library and should be called before any other library call. It takes no arguments and will only fail if there is insufficient system memory.

`EMBX_Mailbox_Register` is used to register a hardware mailbox with the library. Once a mailbox has been registered, transports that make use of the hardware mailboxes will be able to allocate resources. The mailboxes must be registered before such a transport is initialized by either `EMBX_Init` or `EMBX_RegisterTransport`. See *Section 8.2: Initialization on page 68* for further information on when transports are initialized.

Example mailbox registrations for targeted platforms can be found in *Appendix A: Transport configurations on page 192*.

9.2.1 EMBXMailbox as a Linux kernel module

When `EMBXMailbox` is loaded as a kernel module then mailboxes can be registered using module parameters. Up to four mailboxes can be registered using the parameters `mailbox0`, `mailbox1`, `mailbox2` and `mailbox3`.

The standard form for each of the parameters is:

`mailbox-addr``:`*`interrupt-number`*`:`*`flags`*`...`

Flags can contain (in lower case) the last word of each flag listed in *Table 15: EMBX_Mailbox_Flags_t flags on page 119*.

Example:

```
insmod embxmailbox.o mailbox0=0x66150000:112:set2:activate
```

9.3 EMBXSHM

`EMBXSHM` is a shared memory transport suited to platforms with high bandwidth memory access.

`EMBXSHM` requires the following hardware support from its underlying platform:

- an area of shared memory that can be directly addressed by all participants in the platform, this memory need not appear at the same address in each address map
- a mechanism for interrupting every other participant within the transport

`EMBXSHM` supports zero-copy data transfers both for messages, and for distributed objects held in memory addressable by all participants.

`EMBXSHM` requires exactly one master processor and can support any number of slave processors. The maximum number of slave processors is configured at compilation time and the supplied binaries are configured to support no more than three slave processors.

By convention the processor that managed the boot process, for example by configuring the memory interfaces, will be the master processor although there is nothing in the transport that mandates this.

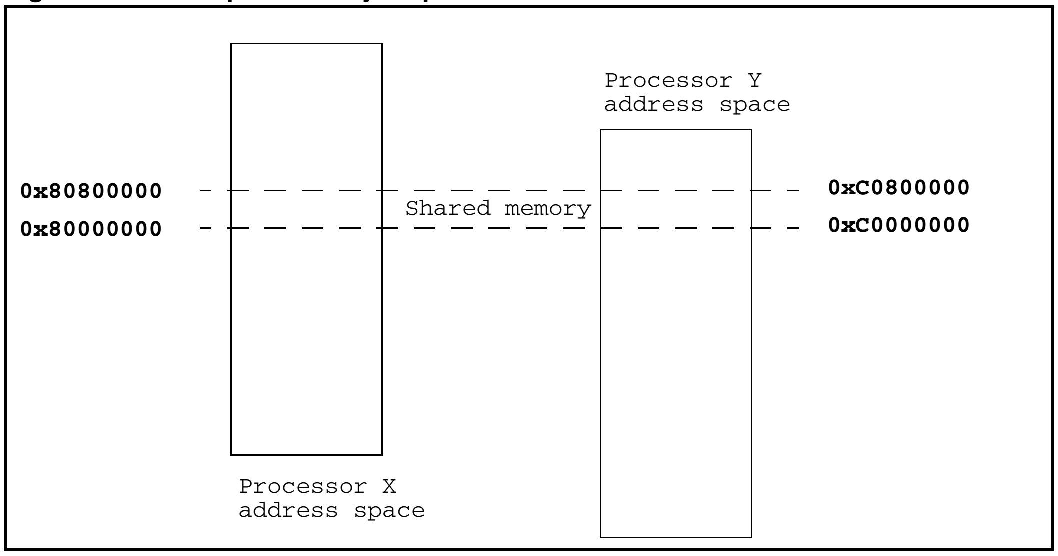
The master processor is responsible for initializing the shared memory and for communicating the address of that memory to the slave processors. The master processor may also be responsible for automatically allocating memory to share if the transport configuration has requested this.

9.3.1 Address modes and pointer warping

While `EMBXSHM`'s shared memory pool must be contiguous in the processors' memory map there is no requirement for the pool to map to the same address on all processors because `EMBXSHM` contains an in-built means to warp pointers before passing them to other processors.

Consider for example two processors, X and Y, that possess the memory maps shown in *Figure 7: Example memory maps on page 81*.

Figure 7. Example memory maps



There are 8 Mbytes of shared memory mapped at different addresses on each processor. The address of the base of the shared memory on a particular processor is called the local base address. In this case that is 0x80000000 for processor X and 0xC0000000 for processor Y.

In order to make it easy to calculate the values by which the pointers will be warped for translation we introduce the concept of a bus base address. The bus base address is a selected common value representing the memory bus view of the base of shared memory. For multi-core devices the shared memory is typically on the system bus thus, the bus address is typically the physical address of the shared memory. For other devices the choice is less clear since there are multiple memory buses, in this situation the bus base address is usually considered to be zero. In the above example we will assume that the bus base address is zero.

The pointer warp is added to the local address to form the bus address. Thus the pointer warp can be calculated for each processor as the local base address subtracted from the bus base address.

Note: The pointer warp is always considered to be positive, to achieve negative conversions a positive number is used such that the address calculation will overflow.

Thus for processor X the pointer warp is:

```
0x0 - 0x80000000 = -0x80000000 = 0x80000000
```

and for processor Y the pointer warp is:

```
0x0 - 0xC0000000 = -0xC0000000 = 0x40000000
```

9.3.2 Linking and loading

On OS21 the application must link against the EMBX shell, the mailbox library and the shared memory transport.

Assuming the library search path is correctly set, this requires the following to be added to the link line:

```
-lembxshell -lembxmailbox -lembxshm
```

Note: The ST40 and ST200 toolsets perform linking in a single pass. For this reason it is important to preserve the above ordering or the application will fail to link. Similarly it is vital that the object files appear in the link line **after** *the above options.*

In Linux kernel space the application must be linked against the same components, however, since linking is performed automatically by the module loader this can be achieved by loading the following modules, in order, into the kernel:

```
embxshell.o embxmailbox.o embxshm.o
```

9.3.3 The mailbox factory function

The mailbox factory function, `EMBXSHM_mailbox_factory` is used to manufacture transports that use the `EMBXMailbox` library to interrupt other participants but have no other specialist requirements.

Devices that should use this factory function are typically multi-core devices that share a single address space; on these devices there is no need to expose memory to slave devices.

Processors such as the ST40 and the ST231 running OS21 offer multiple views of memory. It is important that pointers supplied to the factory function point to the correct view of memory. For EMBXSHM the restrictions are as follows:

Table 12. Pointer types required for EMBXSHM configuration

Parameter	ST40	ST231
Local address (used to calculate `pointerWarp`)	P2 uncached virtual address	Physical address
`sharedAddr`	P2 uncached virtual address	Uncached virtual address
`warpRangeAddr`	P2 uncached virtual address	Uncached virtual address

The transport configuration structure, `EMBXSHM_MailboxConfig_t`, is described in *Table 13 on page 83*.

Table 13. EMBXSHM_MailboxConfig_t structure

Name	Type	Description
name	EMBX_CHAR []	The name of this transport.
cpuID	EMBX_UINT	ID of the local processor within the transport. The master must have a CPU identity of 0.
participants	EMBX_UINT [8]	Map of processors participating in this transport. An array element should be set to 0 if the corresponding CPU identity does not exist, otherwise the element should be set to 1.
pointerWarp	EMBX_UINT	Correction factor used to convert a **local** address into a **bus** address.
maxPorts	EMBX_UINT	The maximum number of receive ports that can be open at one time. Set the value to zero to permit an unlimited number of ports.
maxObjects	EMBX_UINT	The maximum number of distributed objects that can be registered at one time.
freeListSize	EMBX_UINT	The number of pre-allocated free nodes per port. This is the maximum number of unreceived communication events that a single port can support.
sharedAddr	EMBX_VOID *	Base address of the memory pool to be managed by EMBXSHM. For automatic allocation of the memory pool sharedAddr should be NULL.(1)
sharedSize	EMBX_UINT	Size of the shared memory pool in bytes.
warpRangeAddr	EMBX_VOID *	The primary warp range base address. Used to specify the memory range base address for which zero-copy memory transfers between the CPUs are safe. See *warpRangeAddr and warpRangeSize on page 84*.
warpRangeSize	EMBX_UINT	Size of the primary warp range for which zero-copy data transfer is safe. The primary warp range starts at warpRangeRange and ends at warpRangeAddr + warpRangeSize.
warpRangeAddr2	EMBX_VOID *	The secondary warp range base address. Used to specify the memory range base address for which zero-copy memory transfers between the CPUs are safe. See *warpRangeAddr2 and warpRangeSize2 on page 85*.
warpRangeSize2	EMBX_UINT	Size of the secondary warp range for which zero-copy data transfer is safe. The primary warp range starts at warpRangeRange2 and ends at warpRangeAddr2 + warpRangeSize2.

1. If a Linux master processor is configured to automatically allocate memory for the shared heap it attempts to allocate the memory from the bigphysarea. For such a configuration to operate correctly the following must appear in the Linux kernel arguments:

```
bigphysarea=pages
```

where *pages* is the number of operating system pages to be allocated (each page is 4096 bytes in size on the ST40/SH4). This value should be selected to ensure there is enough memory to allocate the shared heap.

When configuring CPU 0 (the master processor) all members of the configuration structure must be used. However, for the slave processor all members that are grayed out in *Table 13*, should be omitted, their values will be automatically derived from the master processor configuration.

The following code shows how to register a transport using this factory function.

```
EMBX_MailboxConfig_t config = {
   /* ... */
};
EMBX_FACTORY hFactory;

err = EMBX_RegisterTransport(
         EMBXSHM_mailbox_factory, &config, sizeof(config),
         &hFactory);
```

See *Appendix A: Transport configurations on page 192* for example configuration structures for all targeted platforms supported by the this factory function. This also shows how to configure the transport using Linux kernel module arguments.

warpRangeAddr and warpRangeSize

If `warpRangeAddr` and `warpRangeSize` are set to zero their values are automatically determined from the values used for `sharedAddr` and `sharedSize`.

By setting `warpRangeAddr` and `warpRangeSize` the range of addresses can be extended, for which EMBX does not duplicate data when transferring it from one processor to another (zero-copy). This is only possible when both processors are capable of directly addressing data held in shared memory.

Note: *Multi-core devices typically share a single STBus, the effect of this is that all memory is shared. This allows very large warp ranges to be used.*

By setting these members, an application is asserting not only that it is possible to zero-copy data that falls within that address range but also that it is safe. It is therefore the application's responsibility to ensure that it really is safe to zero-copy pointers in that range. The following steps are required to be safe:

1. The extended warp range must entirely enclose the shared memory pool. If `sharedAddr` is NULL then the shared memory pool is allocated automatically. Thus for OS21 based systems the warp range must cover the host's entire C library heap, and for Linux based systems it must cover the entire `bigphysarea`.
2. The `pointerWarp` value must be correct for the entire span of the extended warp range. This typically requires the address space described to be physically contiguous on both the host and any companion processors.
3. Additionally on ST231 processors it is the application/bootstrap writer's responsibility to ensure speculation is enabled to all RAM addresses within the range described on both host and companion processors.

Host 32-bit space enhanced mode support

The R3.2 release of Multicom is the first to support the ST40 host running in 32-bit space enhanced (SE) mode. In this mode the ST40 no longer has a simple mapping from local addresses to physical (bus) ones (that is the P1/P2 mappings in 29-bit mode).

Hence it is not possible to use the EMBXSHM pointer warping mechanism in the same way. To address this issue, Multicom 3.2 and later have modified the EMBXSHM Warp range specification to use physical (bus) addresses instead of local ones. This change also means that the `pointerWarp` calculation is no longer necessary and that parameter can be set to `0`.

warpRangeAddr2 and warpRangeSize2

On platforms where there are two disjoint physical memory regions (LMI banks) it may not be possible to specify a single warp range which spans both. Hence the EMBXSHM configuration has been extended to include a **secondary** warp range specification, with the original warp range now being referred to as the **primary** one.

This secondary warp range behaves like the primary one in that memory between `warpRangeAddr2` and `warpRangeAddr2 + warpRangeSize2` is assumed to be safe for zero-copy data transfers.

In the situation where a secondary warp range is required, a primary warp range must always be specified and it must still span the shared memory pool as before. It does not matter if the secondary address `warpRangeAddr2` is lower than that of the primary address `warpRangeAddr`.

If no secondary warp range is required then set the `warpRangeSize2` parameter to 0.

9.4 EMBXSHMC

EMBXSHMC is a variant of the EMBXSHM transport which allows its main memory pool to be cached. Its configuration and setup is identical to that of EMBXSHM, see *Section 9.3: EMBXSHM on page 80*.

EMBXSHMC performs extensive cache flushes during its operation, for this reason it takes much longer to transfer data which significantly affects latency. However, in some cases the use of the cache improves communication bandwidth since the application's read and write operations to data buffers, often run significantly faster.

The choice of EMBXSHM versus EMBXSHMC should be made on an application by application basis and largely depends on how intensively the application uses allocated memory.

Note: EMBXSHM and EMBXSHMC are wire-compatible making it possible for some processors within the system to run EMBXSHM and some EMBXSHMC. For example some MME configurations are well suited to running EMBXSHM on the host processor and EMBXSHMC on each companion processor.

9.4.1 The mailbox factory function

Processors such as the ST40 and the ST231 running OS21 offer multiple views of memory. It is important that pointers supplied to the factory function point to the correct view of memory. For EMBXSHMC the restrictions are given in *Table 14*.

Table 14. Pointer types required for EMBXSHMC configuration

Parameter	ST40	ST231
Local address (used to calculate `pointerWarp`)	P1 cached virtual address	Physical address
`sharedAddr`	P1 cached virtual address	Cached virtual address
`warpRangeAddr`	P1 cached virtual address	Cached virtual address

Part 5 Functions, types and macros

Functions, types and macros covers:

- *EMBX functions*
- *MME functions and macros*
- *MME constants, enums and types*

10 Function descriptions

10.1 EMBX functions

EMBX_Address

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_Address(EMBX_TRANSPORT  tp,
                        EMBX_INT        offset,
                        EMBX_VOID     **address)
```

Arguments:

`tp`	Specifies the transport that should be used for translating the value.
`offset`	Specifies the value to be translated.
`address`	Specifies a pointer to the pointer that will initialized with the translated address.

Returns:

`EMBX_INVALID_TRANSPORT`	The given transport handle was invalid or the transport does not support the operation.
`EMBX_INVALID_ARGUMENT`	The offset could not be translated in the context of the given transport.
`EMBX_INCOHERENT_MEMORY`	The offset was successfully translated and maps to an incoherent memory address.
`EMBX_SUCCESS`	The call was successful.

Description: Convert an opaque value, representing a memory address within the given transport's memory pool, into a memory pointer suitable for the caller's CPU and OS environment.

This call may not be supported on all transport types, which is indicated by the `allowsPointerTranslation` field in the transport information structure.

Comments: See also: *EMBX_Offset*

EMBX_Alloc

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_Alloc(EMBX_TRANSPORT tp,
                      EMBX_UINT      size,
                      EMBX_VOID     **buffer)
```

Arguments:

`tp`	Specifies the transport that is allocated from..
`size`	Specifies the number of bytes to allocate.
`buffer`	Specifies a pointer to the pointer to be initialized with the allocated memory address.

Returns:

`EMBX_INVALID_TRANSPORT`	The given transport handle was invalid.
`EMBX_INVALID_ARGUMENT`	The buffer pointer invalid.
`EMBX_NOMEM`	Insufficient memory was available.
`EMBX_SUCCESS`	The call was successful.

Description: Allocate a memory buffer, large enough to hold the given size in bytes, from the pool for a particular transport.

Comments: See also: *EMBX_Free*

EMBX_GetBufferSize

EMBX_ClosePort

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_ClosePort(EMBX_PORT port)
```

Arguments:

`port`	Specifies the port handle to close.

Returns:

`EMBX_INVALID_PORT`	The given port handle was invalid.
`EMBX_SUCCESS`	The call was successful.

Description: Close a handle to a communication port that has been obtained from one of `EMBX_CreatePort`, `EMBX_Connect` or `EMBX_ConnectBlock`.

Closing a port handle, obtained from `EMBX_CreatePort`:

– unbinds a name from the port, if one is currently bound

– signals any remote connections to the port that it is closing and wait until they signal back that they have invalidated their references to the port

– signals any tasks waiting on this port using `EMBX_ReceiveBlock` that it is closing and waits for the interrupted calls to signal back that they have finished with the port

– releases all pending message buffers on the port

– releases any internal data structures related to the port and port handle

Note: Although the call will interrupt tasks still blocked on the port, this is an indication of a poorly written application. If an application has such tasks, which need to be cleanly shutdown so that they do not use the port handle after it has closed, it should first call `EMBX_InvalidatePort`. *It can then call* `EMBX_ClosePort` *once all of the tasks have reached a point where it is guaranteed they will not use the port handle again.*

– closing a port handle obtained from one of the “Connect” calls

– releases any internal data structures related to the port handle

– reduces the connection count on the target port by one

Comments: See also: *EMBX_Connect*

EMBX_ConnectBlock

EMBX_CreatePort

EMBX_InvalidatePort

EMBX_ReceiveBlock

EMBX_CloseTransport

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_CloseTransport(EMBX_TRANSPORT tp)
```

Arguments:

`tp`	Specifies the transport handle to close.

Returns:

`EMBX_INVALID_TRANSPORT`	The given transport handle was invalid.
`EMBX_PORTS_STILL_OPEN`	The given transport handle still has open ports.
`EMBX_SUCCESS`	The call was successful.

Description: Closes a handle to a transport. This will only succeed if there are no open port handles left that have been created, using the handle to be closed. If any other tasks are waiting on the transport handle with `EMBX_ConnectBlock` then they will be signaled that the handle is closing and the resources associated with the handle will be released. At this point any further use of the handle will result in undefined behavior.

Comments: See also: *EMBX_ClosePort*
EMBX_ConnectBlock
EMBX_OpenTransport

EMBX_Connect

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_Connect(EMBX_TRANSPORT tp,
                        EMBX_CHAR      *portName,
                        EMBX_PORT      *port)
```

Arguments:

`tp`	Specifies the transport that is the target for the connection.
`portName`	Specifies the name of the port to connect to.
`port`	Specifies a pointer to the port handle to be initialized.

Returns:

`EMBX_INVALID_TRANSPORT`	The given transport handle was invalid.
`EMBX_INVALID_ARGUMENT`	The port name or port pointer was invalid or the port name exceeded `EMBX_MAX_PORT_NAME` characters.
`EMBX_PORT_NOT_BIND`	The given port name does not exist.
`EMBX_CONNECTION_REFUSED`	The port refused the connection request.
`EMBX_NOMEM`	Insufficient memory was available for internal structures.
`EMBX_SUCCESS`	The call was successful.

Description: This call attempts to make a connection to a port bound to the provided name in the specified transport. If the port does not exist then the call fails, returning `EMBX_PORT_NOT_BIND`. If the port has an existing connection and was created, allowing only single connections, then this call will fail and return `EMBX_CONNECTION_REFUSED`. The returned port handle can be used to send messages or objects to the destination port using subsequent calls to `EMBX_SendMessage`, `EMBX_SendObject` and `EMBX_UpdateObject`.

Comments: See also: *EMBX_ClosePort*

EMBX_CreatePort

EMBX_OpenTransport

EMBX_SendMessage

EMBX_SendObject

EMBX_UpdateObject

EMBX_ConnectBlock

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_ConnectBlock(EMBX_TRANSPORT  tp,
                             EMBX_CHAR      *portName,
                             EMBX_PORT      *port)
```

Arguments:

`tp`	Specifies the transport that is the target for the connection.
`portName`	Specifies the name of the port to connect to.
`port`	Specifies a pointer to the port handle to be initialized.

Returns:

`EMBX_INVALID_TRANSPORT`	The given transport handle was invalid
`EMBX_INVALID_ARGUMENT`	The port name or port pointer was invalid or the port name exceeded `EMBX_MAX_PORT_NAME` characters.
`EMBX_TRANSPORT_CLOSED`	The transport handle was closed while the call was blocked.
`EMBX_TRANSPORT_INVALIDATED`	The transport handle was invalidated while the call was blocked.
`EMBX_CONNECTION_REFUSED`	The port refused the connection request.
`EMBX_NOMEM`	Insufficient memory was available for internal structures.
`EMBX_SYSTEM_INTERRUPT`	The call was interrupted by the OS before a connection was made.
`EMBX_SUCCESS`	The call was successful.

Description: This call attempts to make a connection to a port bound to the provided name in the specified transport. If the port does not exist then the call blocks until the name has been bound; hence, this call may never return. If the port already exists and has an existing connection, but only supports a single connection model, then this call will fail and return `EMBX_CONNECTION_REFUSED`. If more than one process is blocked on the same port name and the transport only supports single connection ports, then when the port with the given name is created the following happens:

1. one connection request is picked, a connection is made and the blocked task is woken up
2. the tasks blocked on the remaining connection requests are woken up and will return `EMBX_CONNECTION_REFUSED`

If the transport handle is closed by another task calling `EMBX_CloseTransport`, while the call is blocked, the blocked task will be woken up and the call will return `EMBX_TRANSPORT_CLOSED`. If the driver invalidates the transport handle, due to a call to `EMBX_Deinit` or some transport specific event while the call is blocked, the task will be woken up and the call will return `EMBX_TRANSPORT_INVALIDATED`.

In the later case the application is still required to call `EMBX_CloseTransport` to clean up the transport's resources.

In a Linux kernel environment if a user process associated with this operation (through a system call) receives a signal while it is blocked then the call will return `EMBX_SYSTEM_INTERRUPT`.

The returned port handle can be used to send messages or objects to the destination port using subsequent calls to `EMBX_SendMessage`, `EMBX_SendObject` and `EMBX_UpdateObject`.

Comments: See also: *EMBX_ClosePort*

EMBX_CloseTransport

EMBX_CreatePort

EMBX_Deinit

EMBX_OpenTransport

EMBX_SendMessage

EMBX_SendObject

EMBX_UpdateObject

EMBX_CreatePort

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_CreatePort(EMBX_TRANSPORT tp,
     EMBX_CHAR      *name,
     EMBX_PORT      *port)
```

Arguments:

`tp`	Specifies the transport to create the port on.
name	Specifies the name of the port.
`port`	Specifies a pointer to the port handle to be initialized.

Returns:

`EMBX_INVALID_TRANSPORT`	The transport is invalid.
`EMBX_INVALID_ARGUMENT`	The port name or port pointer is invalid or the port name is too long.
`EMBX_ALREADY_BIND`	The port name is already in use in this transport.
`EMBX_NOMEM`	There were insufficient resources available to create the new port.
`EMBX_SUCCESS`	The call was successful.

Description: Creates a new named port on the given transport, returning a port handle for use in subsequent API calls to receive message and object events. The port name can be up to `EMBX_MAX_PORT_NAME` characters in length. The given port name must not already be in use in the specified transport, if it is the call will fail and return `EMBX_ALREADY_BIND`.

This call may fail if the maximum number of ports on the transport has already been reached.

Comments: See also: *EMBX_ClosePort*

EMBX_Receive

EMBX_ReceiveBlock

EMBX_Deinit

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_Deinit(void)
```

Arguments: None.

Returns:

`EMBX_DRIVER_NOT_INITIALIZED`	The driver is not initialized.
`EMBX_SYSTEM_ERROR`	An unexpected system error prevented the driver closing down.
`EMBX_SUCCESS`	The call was successful.

Description: This call de-initializes the driver by doing the following actions:

– invalidate all port and transport handles;

– interrupt any tasks waiting on `EMBX_OpenTransport`, `EMBX_ConnectBlock` or `EMBX_ReceiveBlock`

– wait for all transport handles to be closed by the application

– close down the currently active transports and reset any driver structures

Once shutdown, the driver will be in a state where it is possible to unload it from the system or restart the software with another call to `EMBX_Init`. This call does not unregister transport factories registered using `EMBX_RegisterTransport`.

While this call causes all of the driver's transports to shutdown, which will have varying effects on other processors in the system dependent on the transport implementations, it does **NOT** cause the EMBX driver running on those processors to shutdown as well.

Comments: See also: *EMBX_CloseTransport*

EMBX_Init

EMBX_DeregisterObject

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_DeregisterObject(EMBX_TRANSPORT tp,
     EMBX_HANDLE    handle)
```

Arguments:

`tp`	Specifies the transport the object handle is registered in.
`handle`	Specifies the object handle to deregister.

Returns:

`EMBX_INVALID_TRANSPORT`	The transport handle is invalid.
`EMBX_INVALID_ARGUMENT`	The object handle is invalid or was not registered on this CPU.
`EMBX_SUCCESS`	The call was successful.

Description: Deregister the given object handle from the specified transport; this can only be called successfully on the CPU that registered the object in the first place. If any other CPUs connected to the transport have local copies of the object, these will be freed.

Deregistering an object handle has no physical effect on the object's memory on the owning CPU, nor does it affect any other registered handles for the same memory in either the transport specified or in other transports in the system.

Note: If the object handle to be deregistered has previously been sent to a port using `EMBX_SendObject` *but that event has not yet been received, the driver does* ***not*** *attempt to stop a future receive from succeeding. Accessing a handle from* `EMBX_REC_OBJECT` *event when that handle has been deregistered is undefined. It is the responsibility of all applications in the system to co-operate and correctly manage the lifetime of and access to objects.*

Comments: See also: *EMBX_RegisterObject*

EMBX_FindTransport

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_FindTransport(EMBX_CHAR   *name,
     EMBX_TPINFO  *tpinfo)
```

Arguments:

`name`	Specifies the name of the transport required.
`tpinfo`	Specifies a pointer to the structure to be initialized.

Returns:

`EMBX_DRIVER_NOT_INITIALIZED`	The driver is not initialized.
`EMBX_INVALID_TRANSPORT`	The transport name was not found.
`EMBX_INVALID_ARGUMENT`	An argument was an invalid pointer.
`EMBX_SUCCESS`	The call was successful.

Description: Return a transport information structure for the named transport, if it exists, see *Section 8.3.1: Querying transports on page 69*.

Comments: See also: *EMBX_GetFirstTransport*

EMBX_GetNextTransport

EMBX_Free

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_Free(EMBX_VOID *buffer)
```

Arguments:

`buffer`	Specifies the memory buffer to be released.

Returns:

`EMBX_INVALID_ARGUMENT`	The buffer pointer is invalid.
`EMBX_SUCCESS`	The call was successful.

Description: Free a buffer that has been obtained either from a call to `EMBX_Alloc`, `EMBX_Receive` or `EMBX_ReceiveBlock`. The transport and hence the memory pool, that the buffer belongs to, is deduced from the given pointer. The call will fail if the pointer cannot be associated with a valid buffer and transport.

Comments: See also: *EMBX_Alloc*

EMBX_Receive

EMBX_ReceiveBlock

EMBX_GetBufferSize

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_GetBufferSize(EMBX_VOID *buffer,
     EMBX_UINT *size)
```

Arguments:

`buffer`	Specifies a pointer to the buffer whose size is required.
`size`	Specifies a pointer to the size variable to be initialized.

Returns:

`EMBX_INVALID_ARGUMENT`	The buffer or size pointer is not valid.
`EMBX_SUCCESS`	The call was successful.

Description: Return the actual size of the given buffer pointer, which must have been obtained from `EMBX_Alloc`, `EMBX_Receive` or `EMBX_ReceiveBlock`.

Note: The actual size of the buffer may not be exactly the size requested from `EMBX_Alloc` *since the transport may have internally rounded up the requested size.*

Comments: See also: *EMBX_Alloc*

EMBX_Receive

EMBX_ReceiveBlock

EMBX_GetFirstTransport

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_GetFirstTransport(EMBX_TPINFO *tpinfo)
```

Arguments:

`tpinfo`	Specifies a pointer to the structure to be initialized.

Returns:

`EMBX_DRIVER_NOT_INITIALIZED`	The driver is not initialized.
`EMBX_INVALID_STATUS`	No transports are available.
`EMBX_INVALID_ARGUMENT`	The info pointer is invalid.
`EMBX_SUCCESS`	The call was successful.

Description: Return a transport information structure for the first transport in the driver's internal list. The rest of the transports can be retrieved by successive calls to `EMBX_GetNextTransport`, see *Section 8.3.1: Querying transports on page 69*.

Comments: See also: *EMBX_FindTransport*

EMBX_GetNextTransport

EMBX_GetNextTransport

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_GetNextTransport(EMBX_TPINFO *tpinfo)
```

Arguments:

`tpinfo`	Specifies a pointer to the last transport entry and to the structure to be initialized.

Returns:

`EMBX_DRIVER_NOT_INITIALIZED`	The driver is not initialized.
`EMBX_INVALID_ARGUMENT`	The pointed to structure does not contain a valid transport descriptor.
`EMBX_INVALID_STATUS`	There are no more transports in the list.
`EMBX_SUCCESS`	The call was successful.

Description: Return a transport information structure for the next transport in the driver's internal list. The transport information structure pointed to by `tpinfo` must contain the transport description as returned by either `EMBX_GetFirstTransport` or a previous call to `EMBX_GetNextTransport`, see *Section 8.3.1: Querying transports on page 69*.

Comments: See also: *EMBX_FindTransport*

EMBX_GetFirstTransport

EMBX_GetObject

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_GetObject(EMBX_TRANSPORT tp,
     EMBX_HANDLE    handle,
     EMBX_VOID    **object,
     EMBX_UINT     *size)
```

Arguments:

`tp`	Specifies the transport which the handle is registered in.
`handle`	Specifies the object handle to query.
`object`	Specifies the object memory pointer variable to be initialized.
`size`	Specifies the object size variable to be initialized.

Returns:

`EMBX_INVALID_TRANSPORT`	The transport handle is invalid.
`EMBX_INVALID_ARGUMENT`	The object handle or return value pointers are invalid.
`EMBX_NOMEM`	Unable to allocate new storage for the object on this CPU.
`EMBX_SUCCESS`	The call was successful.

Description: Return a valid machine pointer and size, for the representation on the calling CPU, of the object identified by the given handle. If there is currently no physical representation of this object on this CPU then storage will be allocated, which may fail with `EMBX_NOMEM`.

Comments: See also: *EMBX_RegisterObject*

EMBX_GetTransportInfo

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_GetTransportInfo(EMBX_TRANSPORT tp,
     EMBX_TPINFO *tpinfo)
```

Arguments:

`tp`	Specifies the transport whose information structure is required.
`tpinfo`	Specifies a pointer to the structure to be initialized.

Returns:

`EMBX_INVALID_TRANSPORT`	The transport handle was invalid.
`EMBX_INVALID_ARGUMENT`	The transport info pointer was invalid.
`EMBX_SUCCESS`	The call was successful.

Description: Return a transport information structure for the transport referenced by the given handle, see *Section 8.3.1: Querying transports on page 69*.

Comments: See also: *EMBX_OpenTransport*

EMBX_Init

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_Init()
```

Arguments: None.

Returns:

`EMBX_ALREADY_INITIALIZED`	The driver is already initialized.
`EMBX_SYSTEM_ERROR`	An unrecoverable error occurred while initializing the driver.
`EMBX_SUCCESS`	The call was successful.

Description: If the EMBX driver is not initialized this call will:

– create resources global to the EMBX environment, if not already done

– call each currently registered transport factory function, with its associated argument, to attempt to create a transport instantiation

It is valid for no transports to be registered and therefore for no transports to be created by any registered factories.

Comments: See also: *EMBX_Deinit*

EMBX_RegisterTransport

EMBX_InvalidatePort

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_InvalidatePort(EMBX_PORT port)
```

Arguments:

`port`	Specifies the port handle to invalidate.

Returns:

`EMBX_INVALID_PORT`	The given port handle was invalid.
`EMBX_SUCCESS`	The call was successful.

Description: Invalidate a communication port, using the handle that was returned from `EMBX_CreatePort`. Attempting to invalidate a port handle returned from `EMBX_Connect` or `EMBX_ConnectBlock` will fail returning `EMBX_INVALID_PORT`.

This call will:

– unbind the name from the port, if one exists

– signal any remote connections to the port that it is being invalidated and wait until they signal back that they have invalidated their references to the port

– signal any tasks waiting on this port using `EMBX_ReceiveBlock` that it is invalidated and waits for the interrupted calls to signal back that they have finished with the port

The port handle must still be closed at some later point by the application, using `EMBX_ClosePort`, before the transport can be closed or the driver shut down. Once the port has been invalidated any use of the port handle, except for `EMBX_ClosePort`, will result in `EMBX_INVALID_PORT` being returned.

Comments: See also: *EMBX_ClosePort*

EMBX_CreatePort

EMBX_ReceiveBlock

EMBX_Mailbox_Alloc

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_Alloc(
  void            (*handler)(void *),
  void             *param,
  EMBX_Mailbox_t **pMailbox)
```

Arguments:

`handler`	Specifies the function that will be called back when an interrupt is generated for the allocated mailbox.
`param`	Contains an arbitrary data pointer that will be passed to the callback handler.
`pMailbox`	Points to the mailbox handle where the allocated mailbox will be stored.

Returns:

`EMBX_NOMEM`	The were no free status/enable pairs.
`EMBX_SUCCESS`	The call was successful.

Description: Allocate a matched status/enable pair from the hardware mailbox. This pair will be suitable for generating interrupts on the local processor and will have the supplied interrupt handler attached to it. After allocation both the status and the enable will be set to zero.

Note: This function makes callbacks from interrupt context. The supplied handler must comply with the operating system dependant restrictions on calls running from an interrupt handler.

Comments: See also: *EMBX_Mailbox_Free*

EMBX_Mailbox_AllocLock

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_AllocLock(EMBX_MailboxLock_t **pLock)
```

Arguments:

`pLock`	Points to the lock handle where the allocated lock will be stored.

Returns:

`EMBX_NOMEM`	The were no free status/enable pairs.
`EMBX_SUCCESS`	The call was successful.

Description: Allocate a LOCK register from the hardware mailbox. This lock can be used to implement mutual exclusion between every processor that can address the mailbox. After allocation the lock will be unset.

Note: For a shared lock only one processor should allocate the lock. All other processors should be passed a description of the lock using a shared handle obtained through `EMBX_Mailbox_GetSharedHandle`*.*

Comments: See also: *EMBX_Mailbox_FreeLock*

EMBX_Mailbox_Deregister

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_Deregister(void *pMailbox)
```

Arguments:

`pMailbox`	Pointer to the base address of the hardware mailbox being deregistered.

Returns: None.

Description: Deregister a hardware mailbox. This function will reclaim all local allocated resources associated with a mailbox. It will immediately invalidate all resources allocated from this mailbox, the application must release these before calling this function.

Comments: See also: *EMBX_Mailbox_Free*

EMBX_Mailbox_Free

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_Alloc(EMBX_Mailbox_t *mailbox)
```

Arguments:

`mailbox`	The mailbox handle to be deallocated.

Returns: None.

Description: Deallocates a local mailbox pair, allocated by `EMBX_Mailbox_Alloc`, or remote mailbox pair obtained from `EMBX_Mailbox_Synchronize`. This will free any associated memory and, for local mailboxes, will desensitize the mailbox to interrupts.

Comments: See also: *EMBX_Mailbox_Alloc*

EMBX_Mailbox_FreeLock

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_AllocLock(EMBX_MailboxLock_t *lock)
```

Arguments:

`lock`	The lock handle to be deallocated.

Returns: None.

Description: Deallocate a LOCK register, allocated by `EMBX_Mailbox_AllocLock`. This will free any associated memory.

Note: Do not free locks obtained by `EMBX_Mailbox_GetLockFromHandle`*. These locks should be freed on the processor that allocated them.*

Comments: See also: *EMBX_Mailbox_AllocLock*

EMBX_Mailbox_GetLockFromHandle

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_GetLockFromHandle(
   EMBX_UINT            handle,
   EMBX_MailboxLock_t **pLock)
```

Arguments:

`handle`	The opaque handle.
`pLock`	Points to the lock handle where the allocated lock will be stored.

Returns:

`EMBX_INVALID_ARGUMENT`	The lock handle is not valid.
`EMBX_SUCCESS`	The call was successful.

Description: Convert an opaque handle, obtained from `EMBX_Mailbox_GetSharedHandle`, into a locally available lock handle. This is used by a remote processor to identify a shared lock.

Comments: See also: *EMBX_Mailbox_GetSharedHandle*

EMBX_Mailbox_GetSharedHandle

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_GetSharedHandle(
   EMBX_MailboxLock_t  *lock,
   EMBX_UINT            pHandle)
```

Arguments:

`lock`	The handle of the lock to make opaque.
`pHandle`	Points to the location where the shared handle will be stored.

Returns:

`EMBX_INVALID_ARGUMENT`	The lock handle is not valid.
`EMBX_SUCCESS`	The call was successful.

Description: Convert an locally allocated lock handle into an opaque handle that can be passed to other processors.

This function is used to inform other processor which LOCK register a processor has allocated to provide mutual exclusion protection. The opaque handle can be passed between all processors in the system and uniquely identifies a specific lock register. This opaque handle can be converted back into a local handle using `EMBX_Mailbox_GetLockFromHandle`.

Comments: See also: *EMBX_Mailbox_GetLockFromHandle*

EMBX_Mailbox_Init

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_Init(void)
```

Arguments: None.

Returns:

`EMBX_ALREADY_INITIALIZED`	The hardware mailbox support has already been initialized.
`EMBX_NOMEM`	There was not enough free memory to complete this operation.
`EMBX_SUCCESS`	The call was successful.

Description: Initialize the hardware mailbox support library. This will allocate any software resources required to manage the hardware.

Note: This function is not thread-safe since it is responsible for allocating thread-safety resources.

EMBX_Mailbox_InterruptClear

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_InterruptClear(
   EMBX_Mailbox_t *mailbox,
   EMBX_UINT       bit)
```

Arguments:

`mailbox`	Handle for the enable/status pair the be cleared.
`bit`	Bit to be cleared (0-31).

Returns: None.

Description: Clear a particular bit within the mailbox pairs STATUS register clearing any interrupt associated with that bit.

Comments: See also: *EMBX_Mailbox_InterruptDisable*

EMBX_Mailbox_InterruptEnable

EMBX_Mailbox_InterruptRaise

EMBX_Mailbox_StatusGet

EMBX_Mailbox_StatusMask

EMBX_Mailbox_StatusSet

EMBX_Mailbox_InterruptDisable

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_InterruptDiable(
   EMBX_Mailbox_t *mailbox,
   EMBX_UINT       bit)
```

Arguments:

`mailbox`	Handle for the enable/status pair the be cleared.
`bit`	Bit to be cleared (0-31).

Returns: None.

Description: Clear a particular bit within the mailbox pairs ENABLE register disabling any interrupt associated with that bit.

Comments: See also: *EMBX_Mailbox_InterruptClear*

EMBX_Mailbox_InterruptEnable

EMBX_Mailbox_InterruptRaise

EMBX_Mailbox_StatusGet

EMBX_Mailbox_StatusMask

EMBX_Mailbox_StatusSet

EMBX_Mailbox_InterruptEnable

Definition:

```
 #include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_InterruptEnable(
  EMBX_Mailbox_t *mailbox,
  EMBX_UINT       bit)
```

Arguments:

`mailbox`	Handle for the enable/status pair the be set.
`bit`	Bit to be set (0-31).

Returns: None.

Description: Set a particular bit within the mailbox pairs ENABLE register enabling any interrupt associated with that bit.

Comments: See also: *EMBX_Mailbox_InterruptClear*

EMBX_Mailbox_InterruptDisable

EMBX_Mailbox_InterruptRaise

EMBX_Mailbox_StatusGet

EMBX_Mailbox_StatusMask

EMBX_Mailbox_StatusSet

EMBX_Mailbox_InterruptRaise

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_InterruptRaise(
   EMBX_Mailbox_t *mailbox,
   EMBX_UINT       bit)
```

Arguments:

`mailbox`	Handle for the enable/status pair the be set.
`bit`	Bit to be set (0-31).

Returns: None.

Description: Set a particular bit within the mailbox pairs STATUS register asserting any interrupt associated with that bit.

Comments: See also: *EMBX_Mailbox_InterruptClear*

EMBX_Mailbox_InterruptDisable

EMBX_Mailbox_InterruptEnable

EMBX_Mailbox_StatusGet

EMBX_Mailbox_StatusMask

EMBX_Mailbox_StatusSet

EMBX_Mailbox_Register

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_Register(
  void                   *pMailbox,
  int                     intNumber,
  int                     intLevel,
  EMBX_Mailbox_Flags_t    flags)
```

Arguments:

`pMailbox`	Pointer to the base address of the hardware mailbox being registered.
`intNumber`	Interrupt identifier of the interrupt that this mailbox can generate on the local processor, or -1 if this mailbox cannot generate interrupts on this processor.
`intLevel`	The interrupt level to attach the mailbox interrupt to, or -1 if this is not applicable, either because the operating system does not support interrupt levels or because the mailbox cannot generate interrupts on this processor.
`flags`	These flags control the way the mailbox is registered. They are used to select active or passive mode, to specify whether the mailbox support hardware mutual exclusion, and to choose which set of registers, if any, is suitable for generating interrupts.

Returns:

`EMBX_INVALID_ARGUMENT`	The supplied interrupt configuration could not be satisfied by the operating system.
`EMBX_NOMEM`	There was insufficient system memory to register the mailbox.
`EMBX_SUCCESS`	The call was successful.

Description: Register a hardware mailbox. This function is used to register both local mailboxes (those that can generate interrupts on the local processor) and remote mailboxes that cannot.

It is possible to register a passive mailbox. A passive mailbox will not be addressed by the mailbox driver until that mailbox sends an interrupt to the driver. This is used to register mailboxes that need to be exposed by a partner device before they can be addressed. Typically when a processor registers a passive mailbox the same mailbox is registered as active on the processor that exposes it. An active mailbox is automatically set to generate an interrupt and wake up the passive partners.

Table 15.EMBX_Mailbox_Flags_t flags

Flag	Description
`EMBX_MAILBOX_FLAGS_SET1`	Local mailbox interrupts by using the first hardware register set.
`EMBX_MAILBOX_FLAGS_ST40,` `EMBX_MAILBOX_FLAGS_SET2`	Local mailbox interrupts by using the second hardware register set.
`EMBX_MAILBOX_FLAGS_PASSIVE`	Do not address this mailbox until it generates an interrupt.
`EMBX_MAILBOX_FLAGS_ACTIVATE`	Wake up the passive partners of the mailbox.

Comments: See also: *EMBX_Mailbox_Deregister*

EMBX_Mailbox_ReleaseLock

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_ReleaseLock(EMBX_MailboxLock_t *lock)
```

Arguments:

`lock`	The handle of the lock to release.

Returns: None.

Description: Leave a region of mutual exclusion protected by the lock handle. This should only be called by the processor that owns the lock.

Comments: See also: *EMBX_Mailbox_TakeLock*

EMBX_Mailbox_StatusGet

Definition:

```
#include <embxmailbox.h>

EMBX_UINT EMBX_Mailbox_StatusGet(EMBX_Mailbox_t *mailbox)
```

Arguments:

`mailbox`	Handle for the enable/status pair.

Returns: The value of the STATUS register.

Description: Obtain the value of the 32-bit STATUS register.

This function is typically used during bootup to pass data values through the mailbox registers.

Comments: See also: *EMBX_Mailbox_InterruptClear*

EMBX_Mailbox_InterruptDisable

EMBX_Mailbox_InterruptEnable

EMBX_Mailbox_InterruptRaise

EMBX_Mailbox_StatusMask

EMBX_Mailbox_StatusSet

EMBX_Mailbox_StatusMask

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_StatusMask(
   EMBX_Mailbox_t *mailbox,
   EMBX_UINT       set,
   EMBX_UINT       clear)
```

Arguments:

`mailbox`	Handle for the enable/status pair.
`set`	Mask of bits to be set.
`clear`	Mask of bits to be cleared.

Returns: None.

Description: Apply a pair of mask values to the 32-bit STATUS register.

This function is typically used during bootup to pass data values through the mailbox registers.

Comments: See also: *EMBX_Mailbox_InterruptClear*

EMBX_Mailbox_InterruptDisable

EMBX_Mailbox_InterruptEnable

EMBX_Mailbox_InterruptRaise

EMBX_Mailbox_StatusGet

EMBX_Mailbox_StatusSet

EMBX_Mailbox_StatusSet

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_StatusSet(
   EMBX_Mailbox_t *mailbox,
   EMBX_UINT       value)
```

Arguments:

`mailbox`	Handle for the enable/status pair.
`value`	The value to be copied into the STATUS register.

Returns: None.

Description: Set the value of the 32-bit STATUS register.

This function is typically used during bootup to pass data values through the mailbox registers.

Comments: See also: *EMBX_Mailbox_InterruptClear*

EMBX_Mailbox_InterruptDisable

EMBX_Mailbox_InterruptEnable

EMBX_Mailbox_InterruptRaise

EMBX_Mailbox_StatusGet

EMBX_Mailbox_StatusMask

EMBX_Mailbox_Synchronize

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_Synchronize(
   EMBX_Mailbox_t  *local,
   EMBX_UINT        token,
   EMBX_Mailbox_t **pRemote)
```

Arguments:

`local`	Mailbox pair where the token request should be advertised.
`token`	The token to be exchanged with a remote partner.
pRemote	Points to the mailbox handle where the remote mailbox will be stored.

Returns:

`EMBX_NOMEM`	The were not sufficient system memory to register the mailbox.
`EMBX_SUCCESS`	The call was successful.

Description: Exchange a token with a remote mailbox.

This function is used by the transports to establish a logical connection between mailbox registers. The function works by examining the hardware mailbox for another processor wishing to synchronize with the same token.

If there is no processor currently advertising that token then this function will block until such a processor becomes available.

When such a processor is found then the remote processor will be supplied with a handle to the caller's local mailbox while the caller receives a handle to the mailbox of the other processor.

Comments: See also: *EMBX_Mailbox_Alloc*

EMBX_Mailbox_Free

EMBX_Mailbox_TakeLock

Definition:

```
#include <embxmailbox.h>

EMBX_VOID EMBX_Mailbox_TakeLock(EMBX_MailboxLock_t *lock)
```

Arguments:

`lock`	The handle of the lock to wait for.

Returns: None.

Description: Enter a region of mutual exclusion protected by the lock handle. Once this function returns no other calls to this function will complete until `EMBX_Mailbox_ReleaseLock` is called.

Note: Due to the multi-processor nature of the lock this function will busy wait consuming all available local processor cycles until the lock is available. For this reason the lock should be held for as short a time as possible

Note: Contention for the lock within a single processor should itself be protected by a normal operating system primitive to minimize unnecessary busy waiting.

Comments: See also: *EMBX_Mailbox_ReleaseLock*

EMBX_Mailbox_UpdateInterruptHandler

Definition:

```
#include <embxmailbox.h>

EMBX_ERROR EMBX_Mailbox_UpdateInterruptHandler(
  EMBX_Mailbox_t *mailbox,
  void           (*handler)(void *),
  void            *param)
```

Arguments:

`mailbox`	Mailbox handle to be updated with an alternative interrupt handler.
`handler`	Specifies the function that will be called back when an interrupt is generated for the allocated mailbox.
`param`	Contains an arbitrary data pointer that will be passed to the callback handler.

Returns:

`EMBX_INVALID_ARGUMENT`	The supplied mailbox handle was invalid.
`EMBX_SUCCESS`	The call was successful.

Description: Update the interrupt handler associated with a particular local interrupt handle.

Comments: See also: *EMBX_Mailbox_Alloc*

EMBX_ModifyTuneable

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_ModifyTuneable(EMBX_tunable_t key,
                               EMBX_UNIT value)
```

Arguments:

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

Returns:

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

Description: Modify tuneable system wide configuration values. EMBX uses sensible default values for parameters such as thread priority and thread stack size, however, some users need to tune these values to reduce memory usage or optimize thread interactions. Many values have been made tuneable to allow users to modify their systems without recompiling Multicom components.

This call alters a single tuneable parameter selected by key, which can be one of the values in *Table 16*:

Table 16.Tuneable values for EMBX parameters

Value	Description
`EMBX_TUNEABLE_THREAD_STACK_SIZE` (Value interpreted as size in bytes).	Tune the stack size of any thread created by either EMBX or MME. All such threads use the same stack size and these stacks are used to make calls to any instantiated MME transformer. The stack requirements of transformers must be considered when tuning this value.
`EMBX_TUNEABLE_THREAD_PRIORITY` (Value interpreted as an OS priority level).	Tune the priority of threads created by any EMBX transport. These threads are typically used for administrative messages (such as port creation or closure) but may also be required by some transports to perform communication. A fairly high value is recommended and the value should always be higher than all MME threads (see *MME_ModifyTuneable*).

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

Note: Tuneables that affect thread initialization do not modify any thread that has already been created. It is therefore recommended that tuneables are modified during system boot before any transport is opened.

Comments: See also: *MME_ModifyTuneable*

EMBX_Offset

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_Offset(EMBX_TRANSPORT tp,
     EMBX_VOID *address,
     EMBX_INT  *offset)
```

Arguments:

`address`	Specifies the pointer to be translated.
`offset`	Specifies a pointer to the opaque value to be initialized.

Returns:

`EMBX_INVALID_TRANSPORT`	The transport handle is invalid or the transport does not support this operation.
`EMBX_INVALID_ARGUMENT`	The address or offset is an invalid pointer.
`EMBX_INCOHERENT_MEMORY`	The call was successful and the supplied memory address resides in incoherent memory.
`EMBX_SUCCESS`	The call was successful.

Description: Translate an address, which is within the memory pool of an initialized transport, to an opaque value suitable for transmission on that transport. If the address is not within the bounds of the transport's memory pool then the call will fail.

This call may not be supported on all transport types, which is indicated by the `allowsPointerTranslation` field in the transport information structure.

Comments: See also: *EMBX_Address*

EMBX_OpenTransport

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_OpenTransport(EMBX_CHAR      *name,
                              EMBX_TRANSPORT *tp)
```

Arguments:

`name`	Specifies the name of the transport to open.
`tp`	Specifies a pointer to the transport handle to be initialized.

Returns:

`EMBX_DRIVER_NOT_INITALIZED`	The driver has not yet been initialized.
`EMBX_INVALID_ARGUMENT`	The name or transport pointers are invalid or the transport name does not exist.
`EMBX_NOMEM`	There were insufficient resources available to create internal data structures.
`EMBX_SYSTEM_ERROR`	The transport failed to initialize with an unrecoverable error or the driver interrupted the call.
`EMBX_SYSTEM_INTERRUPT`	The transport initialization was interrupted by the OS.
`EMBX_SUCCESS`	The call was successful.

Description: Open the named transport, initialize it if necessary and return a transport handle for use in subsequent API calls. If the transport is already waiting for initialization to complete then this call will block until either:

– the initialization is successful

– it is interrupted by the driver (due to an `EMBX_Deinit` or `EMBX_UnregisterTransport`)

– it is interrupted by the operating system

Comments: See also: *EMBX_CloseTransport*

EMBX_Deinit

EMBX_Init

EMBX_UnregisterTransport

EMBX_Receive

Definition:

```
#include <embx.h>

typedef enum { EMBX_REC_MESSAGE, EMBX_REC_OBJECT }
EMBX_RECEIVE_TYPE;

typedef struct {
   EMBX_RECEIVE_TYPE type;
   EMBX_HANDLE       handle;
   EMBX_VOID        *data;
   EMBX_UINT         offset;
   EMBX_UINT         size;
} EMBX_RECEIVE_EVENT;

EMBX_ERROR EMBX_Receive(EMBX_PORT port,
       EMBX_RECEIVE_EVENT *event)
```

Arguments:

`port`	Specifies the port handle to receive the buffer from.
`event`	Specifies a pointer to an event structure to be initialized.

Returns:

`EMBX_INVALID_PORT`	The port handle was invalid.
`EMBX_INVALID_ARGUMENT`	The event pointer was invalid.
`EMBX_INVALID_STATUS`	No messages were available on the port.
`EMBX_SUCCESS`	The call was successful.

Description: Receive a message or object update event from a communication port. This call will fail, not block, if no such event is pending. The port handle must be one that was returned by `EMBX_CreatePort`; handles that originated from either `EMBX_Connect` or `EMBX_ConnectBlock` will result in the call failing with `EMBX_INVALID_PORT`.

The returned event structure is filled in with details from the originating `EMBX_SendMessage` or `EMBX_SendObject` call. The receiver must assume that only the number of bytes specified starting at the given offset contain valid data. For an event of type `EMBX_REC_MESSAGE` the handle will be set to `EMBX_INVALID_HANDLE_VALUE` and the offset will always be zero. The data pointer field is a valid machine pointer to the beginning of the message buffer or object which the receiver can use directly to access the contents. In the case of an `EMBX_REC_OBJECT` event this pointer is identical to that which would be returned from `EMBX_GetObject` given the object handle also in the event structure.

The receiver of a message event is responsible for freeing the message buffer once it has no further need for it, as if the buffer had been returned from `EMBX_Alloc`, or sending the buffer on to another port.

Comments: See also: *EMBX_Alloc*

EMBX_CreatePort

EMBX_Free

EMBX_GetObject

EMBX_SendMessage

EMBX_SendObject

EMBX_ReceiveBlock

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_ReceiveBlock(EMBX_PORT           port,
                             EMBX_RECEIVE_EVENT *event)
```

Arguments:

`port`	Specifies the port handle to receive the buffer from.
`event`	Specifies a pointer to an event structure to be initialized.

Returns:

`EMBX_INVALID_PORT`	The port handle was invalid.
`EMBX_INVALID_ARGUMENT`	The event pointer was invalid.
`EMBX_PORT_CLOSED`	The port handle was closed while blocked.
`EMBX_PORT_INVALIDATED`	The port handle was invalidated while blocked.
`EMBX_SYSTEM_INTERRUPT`	The call was interrupted by the OS before receiving anything.
`EMBX_SUCCESS`	The call was successful.

Description: Receive a message or object update event from a communication port, blocking if no message is available. The port handle must be one that was returned by `EMBX_CreatePort`; handles that originated from either `EMBX_Connect` or `EMBX_ConnectBlock` will result in the call failing with `EMBX_INVALID_PORT`. If the port is closed by another task, while this call is blocked, then it will return `EMBX_PORT_CLOSED`. If the driver invalidated the port because of an `EMBX_Deinit`, `EMBX_UnregisterTransport` or some other transport specific event it will return `EMBX_PORT_INVALIDATED`. In the later case it is still the responsibility of the application to close the port handle.

The returned event structure (see *EMBX_Receive on page 130*) is filled in with details from the originating `EMBX_SendMessage` or `EMBX_SendObject` call. The receiver must assume that only that number of bytes specified starting at the given offset contain valid data. For an event of type `EMBX_REC_MESSAGE` the handle will be set to `EMBX_INVALID_HANDLE_VALUE` and the offset will always be zero. The data pointer field is a valid machine pointer to the beginning of the message buffer or object, which the receiver can use directly to access the contents. In the case of an `EMBX_REC_OBJECT` event this pointer is identical to that which would be returned from `EMBX_GetObject` given the object handle also in the event structure.

The receiver of a message event is responsible for freeing the message buffer once it has no further need for it, as if the buffer had been returned from `EMBX_Alloc`, or sending the buffer on to another port.

Comments:	See also: *EMBX_Alloc* *EMBX_CreatePort* *EMBX_Deinit* *EMBX_Free* *EMBX_SendMessage* *EMBX_SendObject* *EMBX_UnregisterTransport*

EMBX_RegisterObject

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_RegisterObject(EMBX_TRANSPORT tp,
         EMBX_VOID    *object,
         EMBX_UINT     size,
         EMBX_HANDLE  *handle)
```

Arguments:

`tp`	Specifies the transport the object handle is to be registered in.
`object`	Specifies the pointer to the beginning of the object's memory to be registered.
`size`	Specifies the size of the object to register.
`handle`	Specifies the object handle variable to initialize.

Returns:

`EMBX_INVALID_TRANSPORT`	The transport handle is invalid.
`EMBX_INVALID_ARGUMENT`	The object or handle pointers is invalid.
`EMBX_NOMEM`	There were insufficient resources to register the object.
`EMBX_SUCCESS`	The call was successful.

Description: Register the given object with the specified transport, returning a transport unique handle for the object. The returned handle is used to identify the object by any application in the system with an open transport handle, to the same transport it is registered in, in calls to `EMBX_GetObject`, `EMBX_SendObject` and `EMBX_UpdateObject`.

The same physical object can be registered with multiple transports as well as multiple times in the same transport.

Comments: See also: *EMBX_DeregisterObject*

EMBX_GetObject

EMBX_SendObject

EMBX_UpdateObject

EMBX_RegisterTransport

Definition:

```
#include <embx.h>

typedef EMBX_Transport_t *EMBX_TransportFactory_fn(EMBX_VOID *)

EMBX_ERROR EMBX_RegisterTransport(EMBX_TransportFactory_fn *fn,
             EMBX_VOID    *arg,
             EMBX_UINT     arg_size,
             EMBX_FACTORY *handle)
```

Arguments:

`fn`	Specifies the function to be called to create a transport instantiation.
`arg`	Specifies the argument to be passed to the factory function.
`size`	Specifies the size of the argument in bytes.
`handle`	Specifies the factory handle variable to initialize.

Returns:

`EMBX_INVALID_ARGUMENT`	One or more of the arguments were invalid.
`EMBX_NOMEM`	There were insufficient resources to register the object.
`EMBX_SUCCESS`	The call was successful.

Description: Register the given transport factory function and argument pair with EMBX. This may be called before or after `EMBX_Init`, depending on the system environment being used. If called before `EMBX_Init` this will simply register the function/argument pair with the EMBX driver. If the driver is currently initialized then this call will both register the function/argument pair with the driver and call the function with that argument to try and create a transport instantiation.

The call makes a copy of the argument, based on the `arg_size` parameter; therefore the `arg` parameter does not need to be maintained after the call.

The returned factory handle can be used to un-register the factory at a later time using `EMBX_UnregisterTransport`.

Comments: See also: *EMBX_Init*

EMBX_UnregisterTransport

EMBX_SendMessage

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_SendMessage(EMBX_PORT  port,
           EMBX_VOID   *buffer,
           EMBX_UINT    size)
```

Arguments:

`port`	Specifies the destination port for the message.
`buffer`	Specifies a pointer to the buffer to send.
`size`	Specifies the size of the message.

Returns:

`EMBX_INVALID_PORT`	The port handle was invalid.
`EMBX_INVALID_ARGUMENT`	The buffer was not valid or the specified message size was larger that the buffer.
`EMBX_NOMEM`	There were insufficient resources either on the sending or receiving side to complete the operation.
`EMBX_SUCCESS`	The call was successful.

Description: This transfers the given message buffer to the destination port along with the size value, which must not be larger than the size of buffer. The API only guarantees that the first “`size`” bytes of the buffer will be transmitted to the destination; however the transport implementation is free to send more of the buffer if it wishes. The physical size of the buffer on the destination will always be the same size as the source buffer, even on a copying transport. If “`size`” is zero, the effect is that of transferring ownership of the underlying buffer to the destination without incurring any data copying penalty in a copy based transport.

The port handle must have been obtained through a call to `EMBX_Connect` or `EMBX_ConnectBlock`; if the port handle was obtained from `EMBX_CreatePort` then the call will fail with `EMBX_INVALID_PORT`. The buffer must have been allocated on the same transport using `EMBX_Alloc` or received from a port that exists in the same transport as that which contains the destination port. If the buffer belongs to a different transport then the call will fail with `EMBX_INVALID_ARGUMENT`.

After a buffer has been sent to the destination, the buffer is no longer owned by the sender and may have been freed by the driver; hence, the sender must **NOT** use this buffer pointer again.

Comments: See also: *EMBX_Alloc*

EMBX_Connect

EMBX_ConnectBlock

EMBX_CreatePort

EMBX_Receive

EMBX_ReceiveBlock

EMBX_SendObject

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_SendObject(EMBX_PORT   port,
                           EMBX_HANDLE handle,
                           EMBX_UINT   offset)
                           EMBX_UINT   size)
```

Arguments:

`port`	Specifies the destination port for the object.
`handle`	Specifies a handle to the object to be sent.
`offset`	Specifies the offset from the start of the object where data will be updated from.
`size`	Specifies the amount of data to be updated on the destination.

Returns:

`EMBX_INVALID_PORT`	The port handle was invalid.
`EMBX_INVALID_ARGUMENT`	The object handle was not valid or the specified offset and size does not specify a valid region contained by the object.
`EMBX_NOMEM`	There were insufficient resources either on the sending or receiving side to complete the operation.
`EMBX_SUCCESS`	The call was successful.

Description: This call sends an object update event to the given destination port, updating the specified region of the object on the destination if it uses a local copy. If in such a case the storage for the local copy has not yet been allocated on the destination, this will be done. If the allocation fails then this call will also fail with `EMBX_NOMEM`.

The port handle must have been obtained through a call to `EMBX_Connect` or `EMBX_ConnectBlock`; if the port handle was obtained from `EMBX_CreatePort` then the call will fail with `EMBX_INVALID_PORT`. If the object handle is not valid in the port's transport then the call will fail with `EMBX_INVALID_ARGUMENT`.

Comments: See also: *EMBX_Connect*

EMBX_ConnectBlock

EMBX_CreatePort

EMBX_RegisterObject

EMBX_UpdateObject

EMBX_UnregisterTransport

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_UnregisterTransport(EMBX_FACTORY handle)
```

Arguments:

`handle`	Specifies the factory handle to unregister.

Returns:

`EMBX_DRIVER_NOT_INITIALIZED`	The driver has never been initialized.
`EMBX_INVALID_ARGUMENT`	The handle did not reference a registered transport factory
`EMBX_SYSTEM_ERROR`	An unexpected error occurred.
`EMBX_SYSTEM_INTERRUPT`	The call was interrupted by the OS before the operation completed.
`EMBX_SUCCESS`	The call was successful.

Description: Unregister the given transport factory. If a transport created using this factory is still in existence, then the transport and its ports are invalidated and the call waits for it to be closed down as in a call to `EMBX_Deinit`.

Comments: See also: *EMBX_Deinit*

EMBX_Init

EMBX_RegisterTransport

EMBX_UpdateObject

Definition:

```
#include <embx.h>

EMBX_ERROR EMBX_UpdateObject(EMBX_PORT   port,
                             EMBX_HANDLE handle,
                             EMBX_UINT   offset)
                             EMBX_UINT   size)
```

Arguments:

`port`	Specifies the destination port for the object.
`handle`	Specifies a handle to the object to be updated.
`offset`	Specifies the offset from the start of the object where data will be updated from.
`size`	Specifies the amount of data to be updated on the destination.

Returns:

`EMBX_INVALID_PORT`	The port handle was invalid.
`EMBX_INVALID_ARGUMENT`	The object handle was not valid or the specified offset and size does not specify a valid region contained by the object.
`EMBX_NOMEM`	There were insufficient resources either on the sending or receiving side to complete the operation.
`EMBX_SUCCESS`	The call was successful.

Description: This call updates the specified region of the object, identified by the object handle, on the destination CPU identified by the port; if it uses a local copy. If the local copy has not yet been allocated on the destination, this will be done; if that allocation fails then this call will also fail with `EMBX_NOMEM`. Unlike `EMBX_SendObject`, no event is queued on the port to be received by an application on the destination CPU.

The port handle must have been obtained through a call to `EMBX_Connect` or `EMBX_ConnectBlock`; if the port handle was obtained from `EMBX_CreatePort` then the call will fail with `EMBX_INVALID_PORT`. If the object handle is not valid in the port's transport then the call will fail with `EMBX_INVALID_ARGUMENT`.

The order in which data is copied to the destination, in a copy based transport, for calls to `EMBX_SendMessage`, `EMBX_SendObject` and `EMBX_UpdateObject` using the same port handle is guaranteed; this will be the same order as the calls complete in.

Comments: See also: *EMBX_Connect*

EMBX_ConnectBlock

EMBX_CreatePort

EMBX_RegisterObject

EMBX_SendMessage

EMBX_SendObject

EMBXSHM_mailbox_factory

Definition: `#include <embxshm.h>`

```
EMBX_Transport_t *EMBXSHM_mailbox_factory(EMBX_VOID *param)
```

Arguments:

`param`	Configuration parameter supplied to `EMBX_RegisterTransport`.

Returns: A pointer to the transport this factory constructed.

Description: One of the `EMBXSHM` factory functions. For details of usage see *Section 9.3.3: The mailbox factory function on page 82*.

Note: This function should never be called directly.

Comments: See also: *EMBX_RegisterTransport*

10.2 MME functions and macros

MME_AbortCommand

Definition:

```
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 be 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:

– Host function

– Non-blocking function call (the operation completes when the callback function has been called)

MME_AllocDataBuffer

Definition:

```
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 164*.
`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 be 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

– Host function call
– Blocking function call

See also: *MME_FreeDataBuffer*

MME_DeregisterTransformer

Definition: `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 be initialized.

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

Comments: Call type

– Host or companion function call

– Blocking function call

See also: *MME_RegisterTransformer*

Section 3.2.4: Example

MME_DeregisterTransport

Definition: `MME_ERROR MME_DeregisterTransport(const char* name)`

Arguments:

`name`	The name of an EMBX transport that has been registered with `MME_RegisterTransport()`

Returns:

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

Description: Deregisters an EMBX transport that is being used by MME on the CPU from which the call is made.

Comments: Call type:

– Host or companion function call

– Blocking function call

See also: *MME_RegisterTransport*

Section 3.2.4: Example

MME_FreeDataBuffer

Definition:

```
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 be 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

– Host function call

– Blocking function call

See also: *MME_AllocDataBuffer*

MME_GetTransformerCapability

Definition:

```
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 be 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:

– Host function call

– Blocking function call

See also: *MME_TransformerCapability_t*

MME_INDEXED_PARAM

Definition: `#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;
```

Comments: See also: *MME_PARAM*

MME_LENGTH

MME_PARAM_SUBLIST

MME_Init

Definition: `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`.

Comments: See also: *MME_InitTransformer*

MME_Term

MME_InitTransformer

Description:

```
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 be 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:

- Host function call
- Blocking function call

See also: *MME_Init*

MME_TermTransformer

MME_LENGTH

Definition: `#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;
```

Comments: See also: *MME_INDEXED_PARAM*

MME_PARAM

MME_PARAM_SUBLIST

MME_LENGTH_BYTES

MME_LENGTH_BYTES

Definition: `#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.

Comments: See also: *MME_INDEXED_PARAM*

MME_LENGTH

MME_PARAM

MME_PARAM_SUBLIST

MME_ModifyTuneable

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 17*:

Table 17.Tuneable values for MME parameters

Value[(1)]	Description
`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_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 17.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: Tuneables that affect thread initialization do not modify any thread that has already been created. It is therefore recommended that tuneables are modified during system boot before any transport is opened.

Comments: See also: *EMBX_ModifyTuneable*

MME_NotifyHost

Definition:

```
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 be 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 176.

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:

– Companion function call
– Non-blocking function call

MME_PARAM

Definition: `#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));
```

Comments: See also: *MME_INDEXED_PARAM*

MME_PARAM_SUBLIST

MME_LENGTH

MME_PARAM_SUBLIST

Definition: `#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`).

Comments: See also: *MME_PARAM*

MME_INDEXED_PARAM

MME_LENGTH

MME_RegisterTransformer

Definition:

```
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:

– Host or companion function call
– Blocking function call

See also: *MME_InitTransformer*
MME_DeregisterTransformer

MME_RegisterTransport

Definition: `MME_ERROR MME_RegisterTransport(const char* name)`

Arguments:

`name`	The name of an EMBX transport that has been registered with `EMBX_RegisterTransport`.

Returns:

`MME_SUCCESS`	The transport has been successfully initialized.
`MME_INVALID_ARGUMENT`	The transport name is not registered.

Description: Registers an EMBX transport for MME use on the CPU from which the call is made.

On the host this completes once communication has been established with the a companion CPU that has made a call to `MME_RegisterTransport()`.

Comments: Call type:

– Host or companion function call

– Blocking function call

See also: *MME_DeregisterTransport*

MME_Run

Definition: `MME_ERROR MME_Run(void)`

Arguments: None.

Returns:

`MME_SUCCESS`	The main loop has terminated successfully.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not be initialized.
`MME_NOMEM`	The memory required to complete this command is not available.

Description: Commences the MME main execution loop on a companion CPU. This loop handles messages from the host and terminates when the host calls `MME_Term()` or `MME_DeregisterTransport()` for the transport to this CPU.

Completes when the host calls `MME_Term()` or `MME_DeregisterTransport()`.

Comments: Call type:

– Companion function call

– Blocking function call

See also: *`MME_Term`*

Completes when the host calls `MME_Term()` or `MME_DeregisterTransport()`.

MME_SendCommand

Definition:

```
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 be 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.

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

Comments: Call type: Host function call

Non blocking function call

See also: *MME_AbortCommand*

MME_Term

Definition: `MME_ERROR MME_Term(void)`

Arguments: None.

Returns:

`MME_SUCCESS`	The operation complete correctly.
`MME_DRIVER_NOT_INITIALIZED`	The MME driver has not be initialized.
`MME_HANDLES_STILL_OPEN`	Could not terminate, not all transformers have been terminated.

Description: Terminate a connection with a MME.

Free all the associated memory space.

Comments: Call type:

– Host or companion function call

– Blocking function call

See also: *MME_Init*

MME_TermTransformer

Definition: `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 be 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:

– Host function call

– Blocking function call

See also: *MME_InitTransformer*

MME_AbortCommand

10.3 MME constants, enums and types

MME_AbortCommand_t

Definition:

```
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

Definition:

```
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.

Comments: See also: *MME_Command_t*

MME_SendCommand

MME_Command_t

Definition:

```
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 will lead 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.

Comments: See also: *`MME_SendCommand`*

MME_CommandCode_t

Definition:

```
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.

Comments: See also: *MME_Command_t*

MME_SendCommand

MME_CommandEndType_t

Definition:

```
typedef enum
{
     MME_COMMAND_END_RETURN_NO_INFO
     MME_COMMAND_END_RETURN_NOTIFY
} 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.

Comments: See also: *MME_UINT*

MME_Command_t

MME_SendCommand

MME_CommandId_t

Definition: `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

Definition:

```
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 3.6: Issuing commands on page 29*.

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.

Comments: See also: *MME_CommandStatus_t*

MME_CommandStatus_t

Definition:

```
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 4.4.2: Deferred commands on page 41*.

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).

Comments: See also: *MME_Command_t*

MME_SendCommand

MME_DataBuffer_t

Definition:

```
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.

Comments: See also: *MME_ScatterPage_t*

MME_Command_t

MME_DataFormat_t

Definition:

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

Description: Used to define the format of the data a transformer support 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_ERROR

Definition:

```
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_EMBX_ERROR,
      MME_INTERNAL_ERROR,
      MME_NOT_IMPLEMENTED
} 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_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_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_EMBX_ERROR`	EMBX underlying MME has reported an error.
`MME_INTERNAL_ERROR`	There is an internal inconsistency.
`MME_NOT_IMPLEMENTED`	The function is not implemented - for example,`MME_RegisterTransport()` in Linux user mode.

MME_Event_t

Definition:

```
typedef enum
{
      MME_COMMAND_COMPLETED_EVT,
      MME_DATA_UNDERFLOW_EVT,
      MME_NOT_ENOUGH_MEMORY_EVT,
} 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`.

Comments: See also: *MME_Command_t*

MME_SendCommand

MME_GenericCallback_t

Definition:

```
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`.

Comments: See also: *MME_SendCommand*

MME_InitTransformer

MME_GenericParams_t

Definition:	`typedef void* MME_GenericParams_t`
Description:	Generic type used to exchange data between host and companion CPUs.
Fields:	None.
Comments:	See also: *MME_Command_t*

MME_GetTransformerCapability_t

Definition:

```
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_InitTransformer_t

Definition:

```
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

Definition: `#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()`.

Comments: See Also: *MME_InitTransformer*

MME_RegisterTransformer

MME_Priority_t

Definition:

```
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.

Comments: See also: *MME_SendCommand*

MME_ProcessCommand_t

Definition:

```
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 4: Writing an MME transformer on page 36*.

MME_ScatterPage_t

Definition:

```
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 5: MME_ScatterPage_t FlagsIn and FlagsOut on page 29*.

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.

Comments: See also: *MME_DataBuffer_t*

MME_TermTransformer_t

Definition: `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

Definition: `typedef MME_UINTMME_Time_t;`

Description: Describe the time in the MME environment.

Comments: See also: *MME_Command_t*

MME_TransformerCapability_t

Definition:

```
typedef struct {
     MME_UINT                  StructSize;
     MME_UINT                  Version;
     MME_DataFormat_t          InputType;
     MME_DataFormat_t          OutputType;
     MME_UINT                  TransformerInfoSize;
     MME_GenericParams_t       TransformerInfo_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).

Comments: See Also: *MME_GetTransformerCapability*

MME_TransformerHandle_t

Definition: `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.

Comments: See also: *`MME_InitTransformer`*

MME_TransformerInitParams_t

Definition:

```
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).

Comments: See Also: *MME_InitTransformer*

MME_UINT

Definition: `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.

Appendices

The following appendices are provided:

- *Transport configurations*
- *MME supplement*
- *Advanced build options*

Appendix A Transport configurations

This appendix provides **example** transport configurations for the latest targeted platforms. This information is supplementary to, and should be read in conjunction with *Chapter 8: Using the EMBX API on page 67* and *Chapter 9: Transport specifics on page 79*.

A.1 EMBXSHM: STb7100-Mboard

The STb7100-Mboard contains a single STb7100 device. The STb7100 is a multi-core device containing one ST40 processor and two ST231 companion processors, an audio decoder and a video decoder.

This platform uses the mailbox factory function, see *Section 9.3.3: The mailbox factory function on page 82* for more details.

```
EMBX_Transport_t *EMBXSHM_mailbox_factory(EMBX_VOID *param)
```

A.1.1 Mailbox configuration

This platform contains two mailbox peripherals providing four interrupts, two for the ST40 and one each for the companion processors, as shown in *Table 18*.

Table 18. Mailbox configuration for MB411/STB7100

Parameter	ST40	ST231 - video	ST231 - audio
mailbox address (0)	0xB9211000	0x19211000	0x19211000
interrupt number (0)	`OS21_INTERRUPT_MB_LX_DPHI`	`OS21_INTERRUPT_MBOX_SH4`	-1
interrupt level (0)	-1	-1	-1
flags (0)	`EMBX_MAILBOX _FLAGS_SET2`	`EMBX_MAILBOX _FLAGS_SET1`	0
mailbox address (1)	0xB9212000	0x19212000	0x19212000
interrupt number (1)	`OS21_INTERRUPT_MB_LX_AUDIO`	-1	`OS21_INTERRUPT_MBOX_SH4`
interrupt level (1)	-1	-1	-1
flags (1)	`EMBX_MAILBOX _FLAGS_SET2`	0	`EMBX_MAILBOX _FLAGS_SET1`

A.1.2 ST40 transport configuration

```
EMBXSHM_MailboxConfig_t config = {
  "shm",                            /* name */
  0,                                /* cpuID */
  { 1, 1, 1, 0, 0, 0, 0, 0 },      /* participants */
  0x60000000                        /* pointerWarp */
  0,                                /* maxPorts */
  16,                               /* maxObjects */
  16,                               /* freeListSize */
  0,                                /* sharedAddr */
  (2*1024*1024)                     /* sharedSize */
};
```

Note: *This transport is configured for three participants, if fewer processors are required edit the participants map.*

A.1.3 ST231-video transport configuration

```
EMBXSHM_MailboxConfig_t config = {
  "shm",                            /* name */
  1,                                /* cpuID */
  { 1, 1, 1, 0, 0, 0, 0, 0 },      /* participants */
  0                                 /* pointerWarp */
};
```

Note: *This transport is configured for three participants, if fewer processors are required edit the participants map.*

A.1.4 ST231-audio transport configuration

```
EMBXSHM_MailboxConfig_t config = {
  "shm",                            /* name */
  2,                                /* cpuID */
  { 1, 1, 1, 0, 0, 0, 0, 0 },      /* participants */
  0                                 /* pointerWarp */
};
```

Note: *This transport is configured for three participants, if fewer processors are required edit the participants map.*

A.1.5 Booting the platform

On the STb7100-Mboard the ST40 should be booted first. All processors should be booted using the normal toolset configuration scripts.

A.2 EMBXSHM: STb7109-Ref board

The STb7109-Ref board contains a single STb7109 device. The STB7109-Ref board (also known as the mb442) is a multi-core device containing one ST40 processor and two ST231 companion processors, an audio decoder and a video decoder. This platform also uses the mailbox factory function and in terms of its transport configuration is the same as the STb7100, documented in *Section A.1: EMBXSHM: STb7100-Mboard on page 192*.

A.3 EMBXSHM: STi7200-Mboard

The STi7200-Mboard contains a single STi7200 device. The STi7200 is a multi-core device containing one ST40 host processor and four ST231 companion processors, two audio decoders and two video decoders.

This platform uses the EMBXSHM mailbox factory function, see *Section 9.3.3: The mailbox factory function on page 82* for more details.

```
EMBX_Transport_t *EMBXSHM_mailbox_factory(EMBX_VOID *param)
```

A.3.1 Mailbox configuration

This platform contains four mailbox peripherals providing eight interrupts, four for the ST40 and one each for the companion processors, as shown in *Table 19* and *Table 20*.

Table 19. Mailbox configuration for (MB519 + MB520)/STi7200 video/audio 0

Parameter	ST40	ST231 - video0	ST231 - audio0
mailbox address (0)	0xFD800000	0xFD800000	0xFD800000
interrupt number (0)	`OS21_INTERRUPT_MBOX_SH4_AUD0`	-1	`OS21_INTERRUPT_MBOX`
interrupt level (0)	-1	-1	-1
flags (0)	`EMBX_MAILBOX _FLAGS_SET2`	0	`EMBX_MAILBOX _FLAGS_SET1`
mailbox address (1)	0xFD801000	0xFD801000	0xFD801000
interrupt number (1)	`OS21_INTERRUPT_MBOX_SH4_DMU0`	`OS21_INTERRUPT_MBOX`	-1
interrupt level (1)	-1	-1	-1
flags (1)	`EMBX_MAILBOX _FLAGS_SET2`	`EMBX_MAILBOX _FLAGS_SET1`	0

Table 20. Mailbox configuration for (MB519 + MB520)/STi7200 video/audio 1

Parameter	ST40	ST231 - video1	ST231 - audio1
mailbox address (2)	0xFD802000	0xFD802000	0xFD802000
interrupt number (2)	`OS21_INTERRUPT_MBOX_SH4_AUD1`	-1	`OS21_INTERRUPT_MBOX`
interrupt level (2)	-1	-1	-1
flags (2)	`EMBX_MAILBOX _FLAGS_SET2`	0	`EMBX_MAILBOX _FLAGS_SET1`

Table 20. Mailbox configuration for (MB519 + MB520)/STi7200 video/audio 1 (continued)

Parameter	ST40	ST231 - video1	ST231 - audio1
mailbox address (3)	0xFD803000	0xFD803000	0xFD803000
interrupt number (3)	`OS21_INTERRUPT_MBOX_SH4_DMU1`	`OS21_INTERRUPT_MBOX`	-1
interrupt level (3)	-1	-1	-1
flags (3)	`EMBX_MAILBOX _FLAGS_SET2`	`EMBX_MAILBOX _FLAGS_SET1`	0

A.3.2 ST40 transport configuration

The following code excerpt shows an example OS21 configuration for an STi7200 system where the ST40 host (32-bit SE mode) and the ST231 Audio0 companion are being used.

```
EMBXSHM_MailboxConfig_t config = {
  "shm",                         /* name */
  0,                             /* cpuID */
  { 1, 1, 0, 0, 0, 0, 0, 0 },   /* participants (Not used)*/
  0x00000000                     /* pointerWarp */
  0,                             /* maxPorts */
  64,                            /* maxObjects */
  64,                            /* freeListSize */
  (void *) 0x00000000,           /* sharedAddr */
  (1*1024*1024)                  /* sharedSize */
  0x800000000,                   /* PRIMARY warpRangeAddr (32-bit) */
  0x100000000,                   /* PRIMARY warpRangeSize */
  0x400000000,                 /* SECONDARY warpRangeAddr2 (32-bit) */
  0x100000000,                 /* SECONDARY warpRangeSize2 */
};
```

Note: *The primary Warp range is the LMI#1 memory region as this is where the ST40 host code is loaded by default, and hence where the EMBXSHM heap is allocated.*

STLinux

On STLinux the Multicom kernel module configuration for this would be;

```
embxmailbox.ko mailbox0=0xfd800000:44:set2
embxshm.ko mailbox0=shm:0:3:0x00000000:0:64:64:0:1024:0x40000000:
0x10000000:0x80000000:0x10000000
```

Note: *The primary Warp range is the LMI#0 memory region as this is where the STLinux 2.3 kernel must be loaded. However, this conflicts with the default load address of the ST231 companion code and hence must be modified if this code is to be loaded directly into the ST231 using tools such as* **st200gdb** *or* **st200xrun***.*

A.3.3 ST231- Audio 0 transport configuration

```
EMBXSHM_MailboxConfig_t config = {
   "shm",                          /* name */
   1,                              /* cpuID */
   { 1, 1, 0, 0, 0, 0, 0, 0 },     /* participants */
   0                               /* pointerWarp */
};
```

Note: This transport is configured for two participants, if a different number of processors is required edit the participants map.

A.4 EMBXSHM: STi7111-Mboard

The STi7111-Mboard contains a single STi7111 device. The STi7111 is a multi-core device containing one ST40 host processor and two ST231 companion processors, an audio decoder and a video decoder.

This platform uses the EMBXSHM mailbox factory function, see *Section 9.3.3: The mailbox factory function on page 82* for more details.

```
EMBX_Transport_t *EMBXSHM_mailbox_factory(EMBX_VOID *param)
```

A.4.1 Mailbox configuration

This platform contains two mailbox peripherals providing four interrupts, two for the ST40 and one each for the companion processors, as shown in *Table 21*.

Table 21. Mailbox configuration for MB618/STi7111

Parameter	ST40	ST231 - video0	ST231 - audio0
mailbox address (0)	0xFE211000	0xFE211000	0xFE211000
interrupt number (0)	`OS21_INTERRUPT_MB_LX_DPHI`	`OS21_INTERRUPT_MBOX_SH4`	-1
interrupt level (0)	-1	-1	-1
flags (0)	`EMBX_MAILBOX _FLAGS_SET2`	`EMBX_MAILBOX _FLAGS_SET1`	0
mailbox address (1)	0xFE212000	0xFE212000	0xFE212000
interrupt number (1)	`OS21_INTERRUPT_MB_LX_AUDIO`	-1	`OS21_INTERRUPT_MBOX_SH4`
interrupt level (1)	-1	-1	-1
flags (1)	`EMBX_MAILBOX _FLAGS_SET2`	0	`EMBX_MAILBOX _FLAGS_SET1`

A.4.2 ST40 transport configuration

The following code excerpt shows an example OS21 configuration for an STi7111 system where the ST40 host (29-bit mode) and the ST231 audio and video companions are being used.

```
EMBXSHM_MailboxConfig_t config = {
  "shm",                          /* name */
  0,                              /* cpuID */
  { 1, 1, 1, 0, 0, 0, 0, 0 },  /* participants (Not used)*/
  0x00000000                      /* pointerWarp */
  0,                              /* maxPorts */
  16,                             /* maxObjects */
  16,                             /* freeListSize */
  0,                              /* sharedAddr */
  (1*1024*1024)                   /* sharedSize */
  0x0C000000,                     /* PRIMARY warpRangeAddr (29-bit) */
  0x100000000,                    /* PRIMARY warpRangeSize */
  0x000000000,                    /* No SECONDARY warp range */
  0x000000000,                    /* No SECONDARY warp range*/
};
```

Note: This transport is configured for two participants, if fewer processors are required edit the participants map.

STLinux

On STLinux the Multicom kernel module configuration for this would be:

```
embxmailbox.ko mailbox0=0xfe211000:136:set2,mailbox1=0xfe212000:137:set2

embxshm.ko mailbox0=shm:0:7:0x00000000:0:16:16:0:1024:0x0c000000:0x10000000
```

A.4.3 ST231-video transport configuration

```
EMBXSHM_MailboxConfig_t config = {
  "shm",                            /* name */
  1,                                /* cpuID */
  { 1, 1, 1, 0, 0, 0, 0, 0 },     /* participants */
  0                                 /* pointerWarp */
};
```

Note: This transport is configured for three participants, if fewer processors are required edit the participants map.

A.4.4 ST231-audio transport configuration

```
EMBXSHM_MailboxConfig_t config = {
  "shm",                            /* name */
  2,                                /* cpuID */
  { 1, 1, 1, 0, 0, 0, 0, 0 },     /* participants */
  0                                 /* pointerWarp */
};
```

Note: This transport is configured for three participants, if fewer processors are required edit the participants map.

Appendix B MME supplement

This appendix provides supplementary information to *Part 2 MME user guide*.

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 22. 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.

Comments: 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,

/* ... */
}
```

Comments: 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);
```

Comments: 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 *Section 2.2: Building Multicom on page 14* 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 Manual toolset selection

By using variables specified on the make command line it is possible to override the automatic toolset selections. This is used either to suppress the build for a particular processor and operating system in order to reduce compilation time or to forcibly enable a processor and operating system combination, when your environment is not automatically detected.

Table 23. Make variables used for manual toolset selection

Make variable	Description
`ENABLE_IA32_LINUX` `DISABLE_IA32_LINUX`	Linux (user mode) for Intel *x*86.
`ENABLE_IA32_WINNT` `DISABLE_IA32_WINNT`	Windows NT/2000/XP for Intel *x*86.
`ENABLE_ST40_LINUX` `DISABLE_ST40_LINUX`	Linux (user mode) for ST40.
`ENABLE_ST40_LINUX_KO` `DISABLE_ST40_LINUX_KO`	Linux (kernel mode) for ST40.
`ENABLE_ST40_OS21` `DISABLE_ST40_OS21`	OS21 for ST40.
`ENABLE_ST231_OS21` `DISABLE_ST231_OS21`	OS21 for ST231.
`ENABLE_SPARC_SOLARIS` `DISABLE_SPARC_SOLARIS`	Solaris for SPARC.

C.2 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 Section 2.2: Building Multicom on page 14.*

All these facilities are controlled at build time by a single environment or make variable `DEBUG_CFLAGS`. The contents of this variable are place 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 24. Pre-processor macros that enable diagnostic code

Pre-processor macro	Purpose
`EMBX_VERBOSE`	Enable all debug assertions within the EMBX tree. This will also cause the tracing code to be compiled although when used alone this macro will not cause any tracing to be output since each module must have tracing separately enabled.
`EMBX_INFO_MAILBOX=1`	Enable the mailbox library tracing, this shows every call to the hardware mailbox management code (requires `EMBX_VERBOSE`).
`EMBX_INFO_SHELL=1`	Enable shell level tracing, this shows every call to the EMBX API (requires `EMBX_VERBOSE`).
`MME_INFO=1`	Enable tracing within the MME implementation.
`MME_VERBOSE`	Enable all debug assertions with the MME tree.
`EMBX_INFO_SHM=1`	Enable tracing with the shared memory transport, this shows all internal calls between the shell and the shared memory transport.
`RPC_VERBOSE`	Enables tracing within the RPC micro server. The RPC micro server is used only from Linux user mode and trace every transaction between user space and kernel space.

Thus to build an Multicom tree with the maximum possible amount of diagnostic code the following command line could be used.

```
make install DEBUG_CFLAGS="-DEMBX_VERBOSE -DEMBX_INFO_MAILBOX=1 -
DEMBX_INFO_SHELL=1 -DEMBX_INFO_SHM=1 -DRPC_VERBOSE"
```

Note: Enabling all possible tracing is extremely performance damaging and will be likely to swamp all other standard output. Such complete tracing is often useful for hardware bring up but rarely desirable at other times.

`DEBUG_CFLAGS` does not have to be specified on the make command line; it can also be set as an environment variable.

C.3 Running the test suites

The Multicom distribution contains two automatic test suites, one for RPC and one for EMBX. These suites are particular useful for checking that Multicom is fully functional after porting to a new platforms.

Table 25. Directories contains the test suites

Test suite	Directory
EMBX Transport	`$RPC_ROOT/src/embx/test/transport`
MME	`$RPC_ROOT/src/mme/test`
RPC	`$RPC_ROOT/src/rpc/test`

All suites share a similar structure and unless otherwise stated the information in the section applies equally to both. For this reason the variable `RPC_TEST` will be used to represent the directory containing whichever test suite is used.

Each test suite requires two co-operating processors to run correctly. Five environment variables are required to configure the processors on which the test suite runs. These are described in *Table 26 on page 205*.

Table 26. Environment variables for test suites

Variable	Description
`PLATFORM`	The name of the platform being compiled for, which is typically the name of the board in lowercase. A list of supported platforms can be obtained by navigating to `$RPC_ROOT/src` and typing `make platforms`.
`OS_0`	See *Table 4: Make variables on page 15*. In addition to the operating systems supported by the examples, the test suites also support the operating system `linux_ko`, meaning test as a Linux kernel module. (`ko` is the file extension used for kernel modules in 2.6.*x* Linux kernels).
`OS_1`	
`HTI_0`	The name[1] or IP address of the host-target interface (ST Micro Connect) for CPU 0. See also `ENABLE_STMC2=1`.
`HTI_1`	The name[1] or IP address of the host-target interface (ST Micro Connect) for CPU 1. See also `ENABLE_STMC2=1`.
`ENABLE_STMC2=1`	Enable support for the ST Micro Connect 2. In this mode the `HTI_0` and `HTI_1` variables should specify the ST Micro Connect 2 TargetString, in the form: <*stmc_name*>:<*target_pack*>[:<*core_name*>] [[*parameter1*=*value1*] [*parameter2*=*value2*] ...] See the *ST TargetPack user manual ADCS 8020851* for details.

1. For ST40/Linux and ST40/OS21 the name of the interface is the host name of the device itself. For ST200/OS21 the name of the interface is specified in the toolset's configuration files.

Once the environment variables have been set up appropriately the test can be invoked as follows:

```
cd <RPC_TEST>
make run
```

C.4 Tuneable parameters

Both EMBX and MME allow parameters such as thread priority and thread stack size to be tuned without recompiling Multicom components. The functions to modify tuneable parameters are:

```
EMBX_ERROR EMBX_ModifyTuneable(EMBX_Tuneable_t key, EMBX_UINT value)
MME_ERROR MME_ModifyTuneable(MME_Tuneable_t key, MME_UINT value)
```

Each call to one of the above functions allows a single tuneable value to be updated. `EMBX_ModifyTuneable` allows the priority of EMBX worker threads to be altered as well as being able to modify the stack size of all EMBX and MME threads. `MME_ModifyTuneable` allows only the priority of MME worker threads to be altered since the EMBX call is used to modify the stack size of MME threads.

Revision history

Table 27. Document revision history

Version	Date	Comments
G	April 08	Supports the R3.2 Multicom release. Corrected the footnote to *Table 13 on page 83* in *Section 9.3.3: The mailbox factory function*.
F	Mar 08	Supports the R3.2 Multicom release. Removed references to older parts: ST20, OS20 and ST220 throughout. Updated *Table 2: Targeted platforms on page 10*. Updated *Section 2.1: Setting up the distribution on page 13*. Updated *Section 2.2.1: Building Linux kernel modules on page 14*. Added *Section 2.3.2: Running examples on Linux on page 16*. Updated *Section 3.2.1: Initializing MME on page 23*. Added footnote to *Section 3.9.2: Linux on page 35*. Added *Section 3.9.3: STLinux 2.3 and udev support on page 35*. Corrected *Section 4.5: Aborting commands on page 44*. *Section 9.3.3: The mailbox factory function*: – Inserted *Table 12 on page 82*. – Updated *Table 13 on page 83*, `participants`, `warpRangeAddr` and `warpRangeSize` and added `warpRangeAddr2` and `warpRangeSize2`. – Added *Host 32-bit space enhanced mode support on page 84* and *warpRangeAddr2 and warpRangeSize2 on page 85*. – Removed “The EMPI factory function” and the “Generic factory function”. Added *Section 9.4.1: The mailbox factory function on page 85*. Added `EMBX_INCOHERENT_MEMORY` to *EMBX_Address on page 87* and *EMBX_Offset on page 128*. *Chapter 10: Function descriptions*, removed EMBXSHM_empi_mailbox_factory and corrected *Section 10.2*, `MME_Init` description. *Appendix A: Transport configurations*, added: – *A.1: EMBXSHM: STb7100-Mboard on page 192* – *A.2: EMBXSHM: STb7109-Ref board on page 193* – *A.3: EMBXSHM: STi7200-Mboard on page 194* – *A.4: EMBXSHM: STi7111-Mboard on page 196* Corrected *Section C.2: Debugging assertions and logging on page 203*, *Table 24*, `MME_INFO`. Updated *Section C.3: Running the test suites*, *Table 26 on page 205*.

Table 27. Document revision history (continued)

Version	Date	Comments
E	Jan 06	*Section 2.2.1: Building Linux kernel modules on page 14* added. *Section 3.4.5: Cache management*, *Table 5: MME_ScatterPage_t FlagsIn and FlagsOut on page 29* updated descriptions including footnotes. *Section 4.4.2: Deferred commands on page 41* and *Pipelined transformers on page 42* changed MME_COMMAND_DEFERRED to MME_TRANSFORM_DEFFERED. *Section : warpRangeAddr and warpRangeSize on page 84* added. *Section 9.3.5: The generic factory function on page 86* added, including Event notification to EMBXSHM_OPCODE_BUFFER_FLUSH. *EMBX_Mailbox_Register on page 118* updated. *EMBX_ModifyTuneable on page 127* added. *MME_ModifyTuneable on page 152* added. *C.3: Running the test suites on page 204* added example. *C.4: Tuneable parameters on page 205* added.
D	Feb 05	*Section 3.4.3: Subdividing a data buffer on page 27* - removed first paragraph. Updated banners throughout to match maturity of document.
C	Feb 05	*Section 2.3: Examples on page 15* updated. *Section 3.4.3: Subdividing a data buffer on page 27* renamed. *Section 3.4.4: Data buffers in Linux user mode on page 28* added. *Section 9.3.3: The mailbox factory function on page 82* updated. Section A.1.4: Booting the platform updated. Section A.2.2: Booting the platform updated. *Appendix C: Advanced build options on page 203* all sections updated.
B	Sept 04	*Chapter 1: Introduction on page 9* reworked to include MME. *Chapter 2: Getting started on page 13* reworked to include MME. *Part 2 MME user guide on page 18* added. *Section 5.1.1: Structure of a typical system on page 53* added. *Section 9.4: EMBXSHMC on page 85* added. *Section 10.2: MME functions and macros on page 141* added. *Section 10.3: MME constants, enums and types on page 163* added. *Appendix B: MME supplement on page 198* added. Table 21: Mailbox configuration for STi5528-Mboard updated. Section A.3: EMBXSHM: STm8000-Demo board updated. Table 22: Mailbox configuration for MB379/STm8000 updated. Section A.4: EMBXSHM: ST220-EVAL added. *Appendix C: Advanced build options on page 203* removed redundant material. *Table 25: Directories contains the test suites on page 204* updated.
A	Oct 03	First complete version, submitted to ADCS.

Index