Multicom 4

Multicom 4 porting guide

1 Introduction

Multicom 4 has been designed to be source compatible with the Multicom 3 multimedia engine (MME) API. As such, few application or codec code changes should be required to migrate to Multicom 4.

This document provides some tips and guidance on how to port applications and codec code to Multicom 4.

Related documents

Multicom 4 user manual (ADCS 8182595)

Contents

2 Porting to Multicom 4

The following sections consider different aspects of the Multicom 3 implementation and describes how they are treated by Multicom 4.

2.1 ICS configuration

Multicom 4 is now based on a new communications system named the Inter-core system (ICS). All extended mailbox communication library (EMBX) configuration that was used by Multicom 3 is now deprecated.

In particular SoC Mailbox as well as CPU address ranges and participant maps are now configured by a peer platform board support package (BSP) library.

Thus calls to:

```
EMBX_Mailbox_Init
EMBX_Mailbox_Register
EMBX_RegisterTransport
EMBX_Init
```

can be replaced by a single call to:

```
ICS_cpu_init(0)
```

`ICS_cpu_init()` synchronizes with all CPUs specified in the ICS BSP and hence all CPUs must be up and running when this function is called. If one or more CPUs fail to call this function within a certain period, then the function timeouts and returns an error. If users wish to test with only a subset of the SoC CPUs, then they should use the `ics_cpu_init()` API which allows a bitmask of present CPUs to be specified.

2.2 MME buffer and memory management

Multicom 4 is designed to operate using shared memory SoC communications only, and no data copies are performed during communication, resulting in an efficient implementation. To achieve this, all data buffers are transferred using a 'zero-copy' technique where only the physical addresses of the data are transferred between the CPUs.

At the API level, MME deals only with virtual addresses and hence all virtual addresses need to be translated to and from physical ones on each CPU. Previously this was achieved under Multicom 3 by specifying what were known as **Warp ranges** in the EMBX transport configuration.

Multicom 4 replaces the concept of Warp ranges with individual memory region registrations, initiated from the host device driver. This provides a more dynamic and fine grain manipulation of the memory mappings, hence providing better data integrity and error protection across the SoC.

2.3 MME buffer pool

MME still supports the MME data buffer allocation APIs, however, the size of the buffer pool from which these buffers are allocated is no longer configured using the EMBX shared memory (EMBXSHM) transport configuration. Instead the buffer pool is created during MME initialization. Unlike Multicom 3 there is a unique buffer pool for each CPU (rather than a single, globally shared one).

By default the buffer pool size is set to a small value (enough to accommodate MME message meta-data). If you wish to allocate large data buffers from the MME data buffer pool then modify the `MME_TUNEABLE_BUFFER_POOL_SIZE` tuneable by calling;

```
MME_ModifyTuneable(MME_TUNEABLE_BUFFER_POOL_SIZE, size);
```

In order for this to have any affect it must be called before `MME_Init()`.

2.4 MME memory registration

All buffers allocated from the MME data buffer pool are guaranteed to be accessible by all CPUs. Both cached and uncached translations of this memory are allowed.

If your wish to send commands using data buffers not drawn from the MME data buffer pool, then they must first be registered with MME.

To do this call the following function with either a cached or uncached translation of the memory buffer to be registered:

```
MME_RegisterMemory(transformer, buf, size,&handle)
```

This causes the memory region to be registered in the CPU associated with the referenced transformer. Failure to do so causes any `MME_SendCommand()` operations that reference that memory region to fail.

Memory registration is an expensive operation and hence this should be done during transformer initialization and not during the time critical runtime.

Once the transformer has been terminated, you should remove the memory region registration by calling:

```
MME_DeregisterMemory(handle)
```

2.5 MME_Run()

In Multicom 3 it was mandatory for the companion applications to call the `MME_Run()` call after MME was initialized. In Multicom 4 this is no longer necessary, but the `MME_Run()` API is still provided for backwards compatibility. Calling `MME_Run()` on Multicom 4 will simply cause the calling task to block until MME is terminated on that CPU.

2.6 MME_WaitCommand()

Multicom 4 introduces a new method for waiting for command completion. Previously all command completion was indicated by using a transformer callback. Multicom 4 has extended this to also allow the user to block waiting for an individual command to complete. This can be beneficial in cases where there are multiple threads in operation and it can also reduce the number of OS thread schedules required for each command issue.

The API for waiting for a command completion is;

```
MME_WaitCommand(handle, cmdId, eventp, timeout)
```

In order for `MME_WaitCommand()` to function, the MME command must have been issued with a `CmdEnd` code of `MME_COMMAND_END_RETURN_WAKE`. See the *Multicom 4 user manual* (ADCS 8182595) for more details.

3 Revision history

Table 1. Document revision history

Date	Revision	Changes
19-Oct-2009	A	Initial release.