CODE TIME TECHNOLOGIES

mAbassi RTOS

User's Guide

Disclaimer

Code Time Technologies Inc. provides this document “AS IS” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Code Time Technologies Inc. does not warrant that the contents of this document will meet your requirements or that the document is error-free. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. Code Time Technologies Inc. may make improvements and/or changes in the product(s) and/or program(s) described in the document at any time. This document does not imply a commitment by Code Time Technologies Inc. to supply or make generally available the product(s) described herein.

Table of Contents

List of Tables

1 Introduction

This document is the User's Guide for the mAbassi RTOS, the multi-processor version of Abassi, and it provides all the information the reader requires to configure and use the RTOS on multi-processor devices. It is an addendum to the User's Guide for the Abassi RTOS [R1]. Unless indicated, all features present in Abassi are also in mAbassi.

Although mAbassi is the multi-core version of the single core Abassi, mAbassi can be configured to operate in a single core fashion. When configured for single core operation, the code of mAbassi is functionally the same as Abassi.

1.1 Glossary

BMP	Bounded Multi-Processing.
Load Balancing	Distribute workload across multiple processing units.
SMP	Symmetric Multi-Processing.

2 Features

Every one of the features offered by the single core Abassi [R1] are also available in the multi-core version, named mAbassi. As mAbassi is the multi-core version of Abassi, there are new features:

- Symmetric Multi-Processing
- Bounded Multi-Processing
- True automatic load balancing
- Packed automatic load balancing
- No limits on the number of cores
- Can be used on a single core (the single core Abassi is an intrinsic part of the distribution)

2.1 Limitations

There are a few features in the single core Abassi that not available when the mAbassi RTOS is configured for multi-core.

- The cooperative emulation mode is not available.
- The build configuration of single task per priority is not supported.

If mAbassi is configured to operate on a single core, these features are available.

3 Overview

This section gives an overview of the mAbassi RTOS. The design choices are explained so as to give the reader an understanding of the decisions made when mAbassi was architected and implemented.

From the designer point of view, the only difference between the single core Abassi and the multi-core mAbassi is the name of the RTOS source files. In the single core Abassi, the name of the files are `Abassi.c` and `Abassi.h`, but in the multi-core mAbassi, the names were changed to `mAbassi.c` and `mAbassi.h`.

3.1 Design choices

Same as Abassi, see [R1].

3.1.1 Portability

Same as Abassi, see [R1].

3.1.2 Feature set

Same as Abassi, see [R1].

3.1.3 Scalability

Same as Abassi, see [R1].

3.1.4 Code Size

Same as Abassi, see [R1].

3.1.5 Data Size

Same as Abassi, see [R1].

3.1.6 Interrupts not disabled

By the nature of SMP, it is not possible to never disable the interrupts, unlike the single core version of Abassi. The reason for this is that it is necessary to have mutually exclusive access to the kernel between the different cores. As a task execution can move, due to pre-emption, from one core to another core at anytime (except in BMP mode, which offers the targeting of tasks to a specific core), a tiny critical region exists. Upon entering the kernel, the core on which the task is running must be known, as this information is used to report at large which core has entered the kernel. This means the core number has to be read first and a flag set to report that the task on that core is now in the kernel, inhibiting any other core from entering the kernel at the same time. These two very simple operations must be performed without risk of pre-emption in between, so the interrupts must be disabled. The same pre-emption protection is needed when retrieving the task descriptor of the currently running task. The core number must be obtained, as it is used to index an array holding the task descriptors running on all the cores.

On most devices, Abassi disables the interrupts for less than 5 to 10 cycles.

3.2 Services

Same as Abassi, see [R1].

3.2.1 Tasks

Same as Abassi, see [R1].

3.2.1.1 Load Balancing

For Symmetric Multi-Processing (SMP) and Bounded Multi-Processing (BMP), load balancing is automatically handled by mAbassi, so there is no involvement required from the application. A complete description of mAbassi load balancing algorithms is given in Section 9.

3.2.1.2 Multi-Processor Mode

mAbassi offers two types of multi-processing mode:

- Symmetric Multi-Processing (SMP)
- Bounded Multi-Processing (BMP)

When mAbassi is configured in SMP mode, tasks are free to run on any core. The automatic load balancing always determines the optimal core for a task to execute on, such that tasks to do continually swap cores.

When mAbassi is configured in BMP mode, it becomes possible to inform the load balancing algorithm to assign selected tasks to execute on a specific core. Upon creation, all tasks are free to execute on any core. Using the component `TSKsetCore()` (Section 6.2.1) informs the load balancing algorithm to always execute the specified task on the indicated core. The same component is also used to remove the attachment of a task to a specific core.

More information on SMP and BMP is given in Section 8.

3.2.2 Semaphores

Same as Abassi, see [R1].

3.2.3 Mutexes

Same as Abassi, see [R1].

3.2.4 Event Flags

Same as Abassi, see [R1].

3.2.5 Mailboxes

Same as Abassi, see [R1].

3.2.6 Timer

Same as Abassi, see [R1].

3.2.7 Interrupts Handlers

Same as Abassi, see [R1].

3.3 Constraints and Don'ts

Same as Abassi, see [R1].

3.3.1 Idle Task

Same as Abassi, see [R1].

NOTE: On a multi-core target configured for packed load balancing, the Idle Task immediately starts executing upon creation in `OSstart()`. Precautions must be taken to make sure the IdleTask does not use uninitialized services, as none of the application services can yet be initialized. This issue can be circumvented, either by using the runtime safe service creation feature of Abassi, or by using a global flag indicating the services descriptors are valid (when the service descriptor are global resources).

3.3.2 Interrupts

Same as Abassi, see [R1].

NOTE: Starting with version 1.50.52, the lockup condition described below was eliminated with special code in mAbassi to deal with the issue. The constraint about Adam & Eve running on core #0 remains.

NOTE: By its nature, a multi-core RTOS has to rely on inter-core interrupts for the load balancing. This implies a lot of care must be taken to enable the interrupts at the appropriate time. There are no real issues on the slave cores (non-#0 cores) but there could be problems with core #0 at start-up. Core #0 is where the Adam & Eve task runs on. It is also the core where `OSstart()` must be called. When it is time for Adam & Eve to create tasks and open services upon start, the interrupts must by then have been enabled with `OSeint()`. If the interrupts are disabled when a task is created or a service is opened and one or more tasks (including the Idle Task which is created and made ready to run in `OSstart()`), also create tasks and/or open services, it is possible to encounter a lock-up condition. All task creations and service openings are protected by a mutex. If another task locks the mutex when Adam & Eve is creating a task or opening a service, Adam & Eve will block on the mutex. As Adam & Eve becomes blocked, when the mutex is released, if the interrupts are disabled on Core #0, it becomes impossible to unblock Adam & Eve.

Another issue to take care is Adam & Eve is guaranteed to run on Core #0 only as long as the interrupts are disabled. Therefore, if peripherals local to Core #0 must be configured, the configuration must be applied before enabling the interrupts.

3.3.3 Task Suspension

Same as Abassi, see [R1].

3.3.4 Single Task per Priority

Not available in mAbassi.

3.4 Distribution Contents

The mAbassi RTOS source code distribution always has a minimum of 3 files:

`mAbassi.h`	The mAbassi RTOS definition file
`mAbassi.c`	The mAbassi RTOS code
`mAbassi_???_???.?`	The processor / compiler specific assembly file

Most of the distributions are supplied with code examples for specific hardware platforms, and some processor/compiler ports may also include device drivers. Consult the processor/compiler port document that applies to your target application.

4 Configuration

Every build option that is needed in the single core Abassi is also needed in mAbassi [R1]. But, as mAbassi is a superset of Abassi, a few new extra build options are required.

4.1 Build Options

The following sub-sections describe each of the extra build options that must be defined for mAbassi, and the meaning of their value. Another set of sub-sections explains the constraints on some of the single core Abassi build options.

4.1.1 mAbassi Specific Build Options

4.1.1.1 OS_MP_TYPE

The build option `OS_MP_TYPE` is used to configure the way mAbassi handles the multi-processor. This is a bit field, where the type of load balancing is specified (True or Packed; see Section 9), and if SMP or BMP is required. The following table lists the valid numerical values for `OS_MP_TYPE`. Note that to keep mAbassi MISRA compliant, the numerical value used in the definition must be of type unsigned.

Table 4-1: OS_MP_TYPE setting

`OS_MP_TYPE` value	Description
`0U or 1U`	Single core, same as Abassi
`2U`	SMP with true load balancing
`3U`	SMP with packed load balancing
`4U`	BMP with true load balancing
`5U`	BMP with packed load balancing

If the build option `OS_N_CORE` (see Section 4.1.1.4) is set to a value of one (1), the build option `OS_MP_TYPE` is internally overloaded to a value of zero (0), meaning mAbassi operates in single core mode, functionally identical to Abassi.

4.1.1.2 OS_SPINLOCK_BASE

Some devices possess a spinlock hardware module. When mAbassi is targeted for such a device, it is possible for mAbassi to use the hardware spinlock peripheral instead of its own software spinlock. The utilization of a hardware spinlock module implies associating individual hardware spinlocks to the software spinlocks used by mAbassi and the application. The build option `OS_SPINLOCK_BASE` is used to define the base index mAbassi and its optional add-ons use. If the option `OS_SPINLOCK_BASE` is not specified, it is assumed to have a value of zero (0). The hardware spinlock indexes from `OS_SPINLOCK_BASE` to `OS_SPINLOCK_BASE+4` inclusively are reserved for mAbassi internal operations.

Table 4-2 Spinlock Hardware Register Index Usage

Spinlock index	Description
OS_SPINLOCK_BASE+0	Internally used by mAbassi
OS_SPINLOCK_BASE+1	Internally used by mAbassi
OS_SPINLOCK_BASE+2	Internally used by mAbassi
OS_SPINLOCK_BASE+3	Internally used by mAbassi
OS_SPINLOCK_BASE+4	Internally used by mAbassi
OS_SPINLOCK_BASE+5	Used by the optional open-source lwIP IP stack

4.1.1.3 OS_START_STACK

Each core is assigned a start-up code, mainly useful for hardware initialization. That same initialization code defines the minimalist task that runs on any free cores, when there are less tasks running than the number of cores on the device. The stack size of these start-up / do-nothing tasks is defined by the build option OS_START_STACK.

When mAbassi is configured in SMP mode, there is a total of (OS_N_CORE – 1) of these tasks. When mAbassi is configured in BMP mode, there is a total of OS_N_CORE of these tasks, but one of these cannot be used for start-up configuration, as it is kept hidden from the application.

4.1.1.4 OS_N_CORE

The build option OS_N_CORE defines how many cores the target device has. It can be set to a smaller value than the total number of cores on the device, but not greater than. The value of OS_N_CORE must be positive. If OS_N_CORE is set to 1, it makes mAbassi behave identically to the single core version of Abassi. The code used in mAbassi is very close to that of the single core Abassi code.

If the build option OS_MP_TYPE (see Section 4.1.1.1) is set to a value of 0 or 1, indicating to configure mAbassi in single core mode, the build option OS_N_CORE is internally overloaded to a value of 1.

NOTE: mAbassi numbers the cores from 0 to OS_N_CORE-1. The numbering matches the device core physical numbering, so the mAbassi number is a physical numbering (with possibly an offset if the first core on the device is not 0). When mAbassi is configured to operate as the single core version of Abassi, the mAbassi core number is always 0, no matter on which physical core Abassi executes.

4.1.2 Constraints on Abassi Build Options

4.1.2.1 OS_COOPERATIVE

The build option OS_COOPERATIVE that makes the Abassi RTOS kernel operate (truly, emulate) in a cooperative mode instead of preemptive is not available on mAbassi. The reason is quite simple: a cooperative RTOS is intrinsically a single core RTOS. Therefore, the build option OS_COOPERATIVE must always be set to a value of 0 with mAbassi. If it is set to a non-zero value, a compile-time error is generated.

If mAbassi is configured to operate in single core mode, this option is available.

4.1.2.2 OS_PRIO_SAME

The build option OS_PRIO_SAME, which controls if the RTOS supports multiple tasks at the same priority or a single task per priority, must be set to true (a non-zero) value in mAbassi. When mAbassi is configured for multi-core, this build option is internally overloaded and set to a non-zero value. It makes sense to have this constrain in mAbassi, as restricting the RTOS to a single task per priority defeats the multi-core purpose, i.e. with true load balancing, a single task would be running at any time.

If mAbassi is configured to operate in single core mode, this option is available.

4.2 Build Option Selection

Same as Abassi, see [R1].

mAbassi version 1.57.57 and up no longer require at least one task to always be running. Therefore, it is not necessary to include an Idle task, so the build option `OS_IDLE_STACK` can always be set to 0.

When the build option `OS_IDLE_STACK` is set to 0 (No Idle Task), the restriction on starvation protection for tasks running at the lowest priority (`OS_PRIO_MIN`) is removed: tasks running at the lowest priority can now be under starvation protection.

4.3 Build Examples

Same as Abassi, see [R1].

5 Quick Start

Same as Abassi, see [R1].

6 Components

This section describes the new components the mAbassi RTOS offers. These components are in supplement to the single core Abassi components. From a designer or from an application point of view, mAbassi is functionally identical to the single core Abassi. All components of Abassi, described in [R1] are available in mAbassi with exactly the same functionality and constrains.

6.1 Core Components

The only components that are mAbassi specific are related to the fact that applications operate on multiple cores. The core components are listed in the following table:

Table 6-1 Core Component list

Section	Name	Description
6.1.1	`COREgetID`	Get the core the current task is executing on
6.1.2	`CORElock`	Lock a spinlock
6.1.3	`COEunlock`	Unlock a spinlock

6.1.1 COREgetID

Synopsis

```
#include "mAbassi.h"

int COREgetID(void);
```

Description

The component `COREgetID()` reports the core number the current task is executing on.

Availability

Always

Arguments

`void`

Returns

`int` Core number, the value ranges from 0 to `OS_N_CORE-1`

Component type

Function

Options

Notes

The component `COREgetID()` allows an application to apply or avoid any core specific functionality. As a task (if not assigned to a core with BMP) can be preempted at any time and switch cores, the peripheral accesses of core specific operations must be performed with the interrupts disabled. The use of the `COREgetID()` component must be part of that interrupt disabled region.

See also

`OS_MP_TYPE` (Section 4.1.1.1)
`OS_N_CORE` (Section 4.1.1.4)

6.1.2 CORElock

Synopsis

```
#include "mAbassi.h"

void CORElock(int LockNmb, volatile void *Address, int WaitFor,
              int WrtVal);
```

Description

The component `CORElock()` is used to lock a spinlock [R2]. The component interface supports both hardware and software spinlocks. `CORElock()` does not return until the lock has been obtained.
When locking a spinlock, the operation basically consist of writing a core specific value to a variable indicating that the variable is locked. This implies there is a pre-determined value that indicates there are no locks on the spinlock. All this information is passed through the arguments. The address of the variable is specified with the argument `Address`, the unlock value is specified with the argument `WaitFor`, and the core-specific value is indicated with the argument `WrtVal`.
When using a hardware spinlock, the spinlock number is specified with the argument `LockNmb`. If the spinlock is a pure software spinlock, the argument `LockNmb` is ignored.

Availability

Always

Arguments

`LockNmb`	Hardware spinlock number, when hardware spinlock are used Ignored if the spinlock is implemented in software
`Address`	Address of the spinlock variable
`WaitFor`	Value of the variable when a spinlock is unlocked
`WrtVal`	Core-specific value to write into `Address` to indicate the spinlock is locked

Returns

`void`

Component type

Function

Options

Notes

The selection of the argument `WrtVal` is critical. This is due to the fact that the purpose of a spinlock is to give exclusive access to a resource to only one core at a time. Due to some hardware spinlock implementations, some of types of hardware spinlock are not re-entrant. This means a core can lock once, but if a second lock is tried on the same spinlock by the same core, the spinlock will not be re-lockable. This will stall the task forever, as it is waiting to obtain the lock.

A spinlock is a form of very lightweight mutex, and mAbassi's mutexes are re-entrant, but the component `CORElock()` was not designed to be re-entrant (see Section 10 on how to make `CORElock()` re-entrant). To skip extraneous locks, the variable value to write when locking a spinlock must remain unique across cores, but the same on a per core basis. That way, when trying to lock the spinlock, if the variable value is already the value to write, it is known that the lock of the spinlock has already been achieved.
As a task (if not assigned to a core with BMP) can be preempted at any time and switch cores, the locking and unlocking of a spinlock should be performed with the interrupts disabled. The use of the `CORElock()` and `COREunlock()` components must be part of that interrupt disabled region.
mAbassi uses / reserves the hardware spinlocks number 0, 1, 2, 3 and 4 The application must not use any of these numbers for the argument `LockNmb`, otherwise there will be conflict with the internal operation of mAbassi.

See also

`COREunlock` (Section 6.1.3)

6.1.3 COREunlock

Synopsis

```
#include "mAbassi.h"

void COREunlock(int LockNmb, volatile void *Address, int WrtVal);
```

Description

The component `COREunlock()` is used to unlock a spinlock [R2]. The component interface supports both hardware and software spinlocks.
When unlocking a spinlock, the operation simply involves writing the value indicating the spinlock if free. The address of the variable is specified with the argument `Address`, and the unlock variable value is specified with the argument `WrtVal`.
When using a hardware spinlock, the spinlock number is specified with the argument `LockNmb`. If the spinlock is a pure software spinlock, the argument `LockNmb` is ignored.

Availability

Always

Arguments

`LockNmb`	Hardware spinlock number, when hardware spinlock are used Ignored if the spinlock are implemented in software
`Address`	Address of the spinlock variable
`WrtVal`	Value to write into `Address` to indicate the spinlock is free

Returns

`void`

Component type

Macro (Safe)

Options

Notes

There are no checks performed to verify if the spinlock is locked by the core unlocking the spinlock, or that the spinlock is locked.
See the Notes section in the `CORElock()` component description (see Section 6.1.2) for protection against a task switching core.
mAbassi uses / reserves the hardware spinlocks number 0, 1, 2, 3 and 4. The application must not use any of these numbers for the argument `LockNmb`, otherwise there will be conflict with the internal operation of mAbassi.

See also

`CORElock` (Section 6.1.2)

6.2 Task Components

The only task components that are mAbassi specific are related to the fact the application executes on multiple cores. The new task components are listed in the following table:

Table 6-2 Core Component list

Section	Name	Description
	`TSKsetCore`	Assign a task to a core, or set a task as core agnostic

6.2.1 TSKsetCore

Synopsis

```
#include "mAbassi.h"

void TSKsetCore(TSK_t *Task, int CoreID);
```

Description

The component `TSKsetCore()` is used to either force a task to always execute on a specific core, or to specify that a task can execute on any core. The association of tasks to core numbers is what is called BMP (Bounded Multi-Processing) in real-time kernel literature. The task to associate or release from an association is specified by the argument `Task`. When the argument `CoreID` is negative, the task indicated by the argument `Task` is released from core association. When the argument `CoreID` is non-negative, the task specified by the argumernt `Task` becomes associated with the core number `CoreID`; it will then only execute on that core.

Availability

Only when the build option `OS_MP_TYPE` is set to a value of 4 or 5 and the value of the build option `OS_N_CORE` is greater than 1

Arguments

`Task`	Descriptor of the task to associate to a core or release from an association
`NewArg`	`>= 0` : Core number to associate the task
	`< 0` : Release the task from a core association

Returns

`void`

Component type

Macro (Safe)

Options

Notes

If the argument `CoreID` has a value greater or equal to `OS_N_CORE`, the task will never run. As the cores are numbered from 0 to `OS_N_CORE-1`, specifying a value of `OS_N_CORE` or greater associates the task with a non-existing core will most likely makes the application misbehave.
When the component `TSKsetCore()` is applied on a the task already executing on a core other than the one specified by `CoreID`, the task will remain on ist current core until a load balancing happens on any of the cores in the system. Then, the task will be migrated to the targeted core.

This means the proper way to make sure a task will execute on the target core right upon creation is to first create the task in the suspended mode, then use `TSKsetCore()`, followed by a resuming of the newly created task. If the task is created in the ready to run mode, then it is possible that it will execute on a different core right upon its creation as, by default, all newly created tasks are not associated with a core.

See also

`OS_MP_TYPE` (Section 4.1.1.1)
`OS_N_CORE` (Section 4.1.1.4)

7 Appendix A: mAbassi Start-up

mAbassi, like Abassi, starts with the function `main()`, which becomes the first task in the application upon using the component `OSstart()`. As for the single core Abassi, the `OSstart()` component must be called in order to get the RTOS up and running. In the case of multi-core, there is a restriction, as `OSstart()` must be called from Core #0. Calling `OSstart()` on a core other core #0 does nothing. Making `OStart()` do nothing on cores other than core #0 is a protection against initializing the RTOS environment more than once.

A key operation performed within `OSstart()` is to control when the other cores (aside from core #0) can start executing within the RTOS environment, and what they are executing at start-up. Each non-zero core is assigned a start-up task, attached to the function named `COREstartN()`, where N is the core number (numbered from 1 to `OS_N_CORE-1`). That function serves a dual purpose: it is the start-up code for the core, and, equally important, it is also the task executed when a core is unused. Unused core situations happen when the load balancing cannot pack all cores. Very few things need to be done in the `COREstartN()` function[1]:

1) Hardware initialization for the core, if needed

2) Configuring / enabling the interrupts

3) Set the global variable `G_CoreIdleDone[MyCoreID]` to a non-zero value

4) Infinite loop, ideally putting the core in power down mode

The hardware initialization step may not always be required. It would typically be needed to configure the interrupt controller if each core was assigned a dedicated interrupt controller. The second step is required, as mAbassi uses inter-core interrupts as part of its scheduler. For example, when a service used on one core triggers a task switch of the task running on another core, an interrupt is issued to inform the other core that a task switch must be performed.

The third step, setting the global variable `G_CoreIdleDone[MyCoreID]` to a non-zero value is used to inform Adam & Eve the initialization on the non-zero core is done and the interrupts are now enabled. Finally, an infinite loop is required as the Idle Core task can never return, nor get suspended or blocked. When available, the core should be put into a power down mode (capable of responding to interrupts) as it is not used for anything.

The assembly support file supplied with the distribution holds all the `COREstartN()` functions normally needed; these functions can be overloaded with new ones. There should be no need for the application to create and use its own. The only case when custom `COREstartN()` functions would be needed is if the device has peripherals only accessible by a restricted set of cores.

As an example, the `COREstartN()` functions used for the ARM9 are shown in the following table. This code is exactly what is implemented in the distribution.

[1] Starting at version 1.45.46, the non-core #0 start-up mechanism was modified to not need a change of priority.

Table 7-1 COREstartN example for the ARM9

```
#include "mAbassi.h"

void COREstartN(void)
{
  GICinit();                              /* Needed to enable ISR from other cores     */
  OSeint(1);                              /* Enable the interrupt                      */
  G_CoreIdleDone[COREgetID()] = 1;        /* Report to Adam & Eve I am ready           */
for (;;) {                              /* Infinite loop                             */
    asm(" wfi");                          /* Power down: wait for interrupts           */
  }
}
```

NOTE: The task using the functions `COREstartN()` are guaranteed to operate on the selected core until the global variable `G_CoreIdleDone[MyCoreID]` is set to a non-zero value. So there is no need to protect against core switches, as explained in Section 6.1.1.

8 Appendix B: BMP

BMP (Bounded Multi-Processing) is alike SMP, with the added capability of assigning a task to run on a specific core number. The interest of forcing tasks to execute on specific cores arises from the possibility of using a dedicated core to handle all the application interrupts, leaving the other cores interrupt free (excluding inter-core interrupts). Having interrupt free cores delivers the maximum CPU utilization on these cores, as they are not constantly interrupted.

The load balancing algorithm for BMP is more complex than for SMP, therefore if task assignments are not used in an application, mAbassi should be configured for SMP, to maximize the kernel CPU efficiency and code size. Not only is BMP load balancing more complex, it is also prone to having running tasks switch cores without getting pre-empted, blocked or suspended. Here's an example of such a task switch:

On a dual core, let's consider 2 running tasks of the same priority, and both tasks are free to execute on any core. Task #0 executes on core #0 and task #1 executes on core #1. If task #0 gets pre-empted or blocked, and the next running task (task #2) is one that has been assigned to execute on core #1, then a core switch must happen. In sequence, the kernel running on core #0 will interrupt core #1 to force a load balancing to be performed on core #1. The load balancing determines that task #2 must run on core #1, and the current task executing on core #1, task #1, can then only execute on core #0. So, a task switch occurs on core #1. After the task switch on core #1, core #0 immediately regains access to the kernel and makes task #1 execute on core #0. The higher the number of cores on a device, the more complex is the handling of the task core switching. Therefore, a lot of care must be taken during the design phase of a BMP application to minimize the CPU expenditure of having task core switching.

9 Appendix C: Load Balancing

The automatic load balancing operation is an intrinsic part of the mAbassi scheduler, as when any task becomes ready to run, or becomes blocked or suspended, a possible task switch could occur. As mAbassi operates on multi-core devices, the availability of multiple cores allows multiple tasks to execute in parallel. mAbassi offers the selection between four different types of load balancing algorithms:

- True load balancing for SMP
- True load balancing for BMP
- Packed load balancing for SMP
- Packed load balancing for BMP

The initial two (true load balancing for SMP and BMP) allows up to `OS_N_CORE` tasks at the same priority to execute in parallel. If there are less than `OS_N_CORE` tasks ready to run at the highest priority, the extra cores are not utilized. If there are more tasks at the highest priority than `OS_N_CORE`, then all cores are executing tasks. Tasks that are in the ready to run mode will become running in a first come first serve manner. Or, if round-robin is enabled, a round robin scheme will allow all tasks to get their share of the available total CPU.

The two other types of load balancing, the packed load balancing for SMP and BMP, populate all cores with the highest priority ready to run tasks. This type of load balancing breaks the priority rules where a lower priority task should not be executing when a higher priority task is. There is an important gain if the application is architected based on a loose priority scheme. When this is the case, then packed load balancing maximizes the CPU usage. The true load balancing does not offer such CPU maximization.

The allocation of a task to a core is performed automatically, and there is no need for the involvement of the application or the designer to do anything extra.

10 Appendix D: Re-entrant spinlock

The component pair of `CORElock()` (Sections 6.1.2) and `COREunlock` (Section 6.1.3) are not re-entrant. It is very easy to add a bit of code to make them re-entrant. The following table shows two functions, one to lock a re-entrant spinlock and the other to unlock a re-entrant spinlock. As for `CORElock()` and `COREunlock()`, a spinlock variable is needed. In extra, a counter associated to the spinlock is required for re-entrance. The way to use these two functions is shown in the next table. There is no need conditional around `CORElock()`, conditional to verify if the lock is already owned by the task as `CORElock()` simply returns when the lock is already owned.

Table 10-1 Spinlock re-entrance example

```
#include "mAbassi.h"

int Reent_SpinLock(int *Spinlock, int *SpinCnt)
{
   IsrState = OSintOff()                 /* Spinlocks must be interrupt protected     */
   CORElock(OX_SPINLOCK_BASE+15, SpinLock, 0, 1+COREgetID());
   *SpinCnt++;                           /* Counter used to make CORElock() trully    */
                                         /* re-entrant                                */
   return(IsrState);                     /* Return previous ISR enable/disable state  */
}

/* ---------------------------------------------------------------------------------- */

void Reent_SpinUnlock(int *Spinlock, int *SpinCnt, int IsrState)
{
   if (--*SpinCnt == 0) {                /* One less re-entrance nesting              */
       COREunlock(OX_SPINLOCK_BASE+15, SpinLock, 0);    /* Count is 0, time to unlock */
   }
   OSintBack(IsrState);                  /* Bring ISR enable/disable back             */
   return;
}
```

Table 10-2 Spinlock re-entrance use

```
#include "mAbassi.h"

int MySpinLock = 0;
int MySpinCnt  = 0;

      …
/* ------------------------------------------------------------------------------- */

void MyModule1(void)
{
int OldISR;
      …
   OldISR = Reent_SpinLock(&MySpinLock, &MySpinCnt);
   MyModule2();
   Reent_SpinUnlock((&MySpinLock, &MySpinCnt, OldISR);
      …
   return;
}

/* ------------------------------------------------------------------------------- */

void MyModule2(void)
{
int OldISR;
      …
   OldISR = Reent_SpinLock(&MySpinLock, &MySpinCnt);
      …
   Reent_SpinUnlock((&MySpinLock, &MySpinCnt, OldISR);
      …

   return;
}
```

11 References

[R1] Abassi RTOS – User's Guide, available at http://www.code-time.com
[R2] http://en.wikipedia.org/wiki/Spinlock, Spinlock description