CODE TIME TECHNOLOGIES

Abassi RTOS

CMSIS Version 3.0

RTOS API

Table of Contents

1 INTRODUCTION ... 5

1.1 DISTRIBUTION CONTENTS ... 5
1.2 COMPLIANCE ... 5

2 FEATURES ... 6

2.1 UNSUPPORTED FEATURES ... 6
2.2 BUILD OPTIONS ... 6
2.3 EXTRA INFORMATION ... 8
2.3.1 main() threading ... 8
2.3.2 Signals ... 9
2.3.3 Function Arguments ... 9
2.3.4 Abassi components ... 9

3 METRICS ... 11

List of Tables

TABLE 1-1 DISTRIBUTION 5
TABLE 2-1 DISTRIBUTION 6
TABLE 2-2: BUILD OPTIONS FOR CMSIS MINIMUM FEATURE SET 8
TABLE 2-3: MAIN() INITIAL CODE 9
TABLE 2-4 DESCRIPTORS CROSS-REFERENCE 9
TABLE 2-5 PRIORITY NUMBERING TRANSFORMERS 10
TABLE 2-6: PRIORITY REMAPPING 10
TABLE 3-1 CMSIS API CODE SIZE REQUIREMENTS 11

1 Introduction

Introduced in version 3.0 of the Cortex Microcontroller Software Interface Standard, commonly known as CMSIS, a standard API for RTOS was defined. This document details the adaptation layer created by Code-Time Technologies to make the Abassi compliant with the CMSIS V3.0 RTOS API.

The CMSIS compliance is obtained through an adaptation layer; the native Abassi interface hasn't been modified. This CMSIS compliant adaptation layer internally uses the native Abassi components.

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`cmsis_oh.h`	Required include file for the CMSIS V3.0
`cmsis_os.c`	Adaptation layer source code

1.2 Compliance

The CMSIS V3.0 RTOS API layer is fully compliant with the standard except for three aspects. Abassi's possible state transitions do not match the CMSIS state transitions. This non-compliance cannot be removed as an active state thread is put into the inactive state by using the thread suspension capabilities of Abassi. As explained in Abassi's User Guide, the transition from the ready-to-run state or from blocked state to the suspended state require the task to go into the running state before getting suspended. One must remember this constrain was added in Abassi as a protection mechanism against application lock-up. As the `osThreadCreate()` requires the task to be inactive, if a thread that was previously terminated never reached the running state, the call to `osThreadCreate()` will fail.

Extending Abassi's deadlock protection when suspending a task, all CMSIS function `osXxxxxCreate()` possess an added protection against deadlocks too. If an instance of a service to be re-initialized with a `osXxxxxCreate()` have one or more task blocked on it, the re-initialization will not occur and the fact reported as the Service ID returned by the function is set to NULL;

The adaptation layer does not support multiple instance of thread for a single definition. This restriction is enforced in `osThreadCreate()`. So if the number of instances specified in `osThreadDef()` is different than one, the thread creation will fail and will be reported as such.

In the standard, there is no mention indicating the function `osSemaphoreWait()` should not be used in an ISR. As it is possible that a task blocking occurs when calling this function, the Abassi adaptation layer always returns 0 when this function is called in an ISR.

The CMSIS adaptation layer is not MISRA-C:2004 compliant.

2 Features

2.1 Unsupported Features

Some features are optional in the CMSIS V3.0 RTOS API. Abassi does not support the followings:

- The `osWait()` function is not supported as this feature does not fit with Abassi's architecture. As required, the definition of `osFeature_Wait` is set to 0.

The Message Queue Management and the Mail Queue Management features availability depends on the setting of Abassi's build option `OS_MAILBOX`. If mailboxes are part of the build, the Message Queue Management and the Mail Queue Management features are available and as required, the definition of `osFeature_MessageQ` and `osFeature_MailQ` are set to 1. If Abassi's mailboxes are not part of the build, with the build option `OS_MAILBOX` set to zero, the Message Queue Management and the Mail Queue Management features are not available and as required, the definition of `osFeature_MessageQ` and `osFeature_MailQ` are set to 0.

The Memory Pool Management feature does not use Abassi's native memory block components, as the two are not compatible. Therefore there is no need to define the build option `OS_MEM_BLOCK`, or define it with a non-zero value, when using CMSIS RTOS API Memory Pool Management.

2.2 Build Options

To inform Abassi that the CMSIS adaptation layer is used, the build option OS_CMSIS_RTOS must be defined. The value of the definition is not important, but if this build option is not defined, Abassi will not be configured for the CMSIS adaptation layer. To comply with the CMSIS RTOS API, there are constrains on the build options as some optional features of Abassi are in conflict with the CMSIS standard. To eliminate the conflicts, these build options must be set to the values indicated in the following table:

Table 2-1 Distribution

Build Option	Value	Description
`OS_CMSIS_RTOS`	`Don't care`	Must be defined to enable the CMSIS RTOS API
`OS_PRIO_MIN`	`6`	The possible priority level are defined in the standards and the minimum priority value, according to Abassi's numbering, is 6
`OS_PRIO_CHANGE`	`≠ 0`	The standard requires the function `osThreadSetPriority()` which means task priorities can be modified at runtime.
`OS_RUNTIME`	`0`	Runtime service creation is not CMSIS compliant
`OS_STATIC_BUF_MX`	`0`	Runtime service creation is not CMSIS compliant
`OS_STATIC_MBX`	`0`	Runtime service creation is not CMSIS compliant
`OS_STATIC_NAME`	`0`	Runtime service creation is not CMSIS compliant
`OS_STATIC_SEMA`	`0`	Runtime service creation is not CMSIS compliant
`OS_STATIC_STACK`	`0`	Runtime service creation is not CMSIS compliant
`OS_STATIC_TASK`	`0`	Runtime service creation is not CMSIS compliant
`OS_TASK_SUSPEND`	`≠ 0`	The standard requires the function `osThreadTerminate()` which is mapped to Abassi's task suspension operation.
`OS_TIMEOUT`	`≠ 0`	The CMSIS standard requires timeout on services.
`OS_TIMER_SRV`	`≠ 0`	The CMSIS standard requires the timer services.
`OS_TIMER_US`	`≠ 0`	The CMSIS standard requires a timer.

OS_USE_TASK_ARG	≠ 0	The CMSIS standard requires the capability of passing arguments to the thread function.

The value of each one of the above build options is verified in the header file cmsis_os.h. Any non-compliance will generate an error at compile time.

The CMSIS adaptation layer for Abassi does not use dynamic memory allocation. Therefore, unless the application requires dynamic memory allocation then the build option OS_ALLOC_SIZE should be set to a value of zero; that's unless it is desired the application use memory reserved for the component OSalloc(). This said, using OSalloc(), with or without memory reserved for it with OS_ALLOC_SIZE, does have its advantages as the memory allocation gets multi-threading protection through Abassi's internal kernel mutex.

The following table shows the setting of the build options that will deliver the strict minimum set of features required with the CMSIS standard:

Table 2-2: Build Options for CMSIS Minimum Feature Set

```
#define OS_CMSIS_RTOS       1   /* Abassi need this to be defined for CMSIS        */

#define OS_ALLOC_SIZE       0   /* When !=0, RTOS supplied OSalloc                 */
#define OS_COOPERATIVE      0   /* When !=0, the  kernel is in cooperative mode    */
#define OS_EVENTS           1   /* != 0 when event flags are supported             */
#define OS_FCFS             0   /* Allow the use of 1st come 1st serve services    */
#define OS_IDLE_STACK       0   /* If IdleTask supplied & if so, stack size        */
#define OS_LOGGING_TYPE     0   /* Type of logging to use                          */
#define OS_MAILBOX          1   /* != 0 when mailboxes are used                    */
#define OS_MAX_PEND_RQST    xxU /* Maximum number of requests performed in ISRs    */
#define OS_MEM_BLOCK        0   /* If the block memory pool part of the build      */
#define OS_MIN_STACK_USE    x   /* If the kernel minimizes its stack usage         */
#define OS_MTX_DEADLOCK     0   /* != 0 to enable mutex deadlock protection        */
#define OS_MTX_INVERSION    0   /* >0 priority inheritance, <0 priority ceiling    */
#define OS_NAMES            0   /* != 0 when named Tasks / Semaphores / Mailboxes  */
#define OS_NESTED_INTS      0   /* != 0 operating with nested interrupts           */
#define OS_OUT_OF_MEM       0   /* If trapping out of memory conditions            */
#define OS_PRIO_CHANGE      1   /* If a task priority can be changed at run time   */
#define OS_PRIO_MIN         6   /* Max priority, Idle = OS_PRIO_MIN, AdameEve = 0  */
#define OS_PRIO_SAME        1   /* Support multiple tasks with the same priority   */
#define OS_ROUND_ROBIN      0   /* Use round-robin, value specifies period in uS   */
#define OS_RUNTIME          0   /* If create Task / Semaphore / Mailbox at run time */
#define OS_SEARCH_ALGO      0   /* If using a fast search                          */
#define OS_STACK_CHECK      0   /* Set to != for checking stack coillisions        */
#define OS_STARVE_PRIO      0   /* Priority threshold for starving protection      */
#define OS_STARVE_RUN_MAX   0   /* Maximum Timer Tick for starving protection      */
#define OS_STARVE_WAIT_MAX  0   /* Maximum time on hold for starving protection    */
#define OS_STATIC_BUF_MBLK  0   /* When OS_STATIC_MBLK != 0, # of memory bytes     */
#define OS_STATIC_MBLK      0   /* If !=0 how many block memory descriptors        */
#define OS_STATIC_BUF_MBX   0   /* When OS_STATIC_MBX != 0, # of buffer elements   */
#define OS_STATIC_MBX       0   /* If !=0 how many mailboxes                       */
#define OS_STATIC_NAME      0   /* If named mailboxes/semaphore/task, size in char  */
#define OS_STATIC_SEM       0   /* If !=0 how many semaphores and mutexes          */
#define OS_STATIC_STACK     0   /* if !=0 number of bytes for all stacks           */
#define OS_STATIC_TASK      0   /* If !=0 how many tasks (excluding A&E and Idle)  */
#define OS_STATIC_TIM_SRV   0   /* If !=0 how many timer services                  */
#define OS_TASK_SUSPEND     1   /* If a task can suspend another one               */
#define OS_TIMEOUT          1   /* !=0 enables blocking timeout                    */
#define OS_TIMER_CB         0   /* !=0 gives the timer callback period             */
#define OS_TIMER_SRV        1   /* !=0 includes the timer services                 */
#define OS_TIMER_US         xx  /* !=0 enables timer & specifies the period in uS  */
#define OS_USE_TASK_ARG     1   /* If tasks have arguments                         */
```

2.3 Extra information

Some features in the standard are dependent on the underlying RTOS. This sub-section explains in more details what they are.

2.3.1 main() threading

In Abassi, the function `main()` is normally used to start the RTOS, through the component `OSstart()`. Doing so converts the function `main()` into the highest priority task. This is not the model used in the CMSIS RTOS API, but the adaptation layer complies with the standard by supplying all the related information. The definition of `osFeature_MainThread` is set to 0, reporting that `main()` is not a thread upon entry.

The proper generic `main()` initial code, valid for any compliant CMSIS RTOS API, should look alike what is shown in the following table:

Table 2-3: main() initial code

```
   …

osThreadDef(TaskMain, osPriorityNormal, 1, STACK_SIZE);

   …

main()
{

   …

   if (!osKernelRunning()) {
      osKernelStart(osThread(TaskMain), Arg);
   }
   else {
      TaskMain(Arg);
   }

   return(0);
```

2.3.2 Signals

The Abassi adaptation layer supports 31 signal flags, and this is reported in the definition of `osFeature_Signals`.

2.3.3 Function Arguments

Abassi's technique to pass arguments to the function implementing a task is performed through the `TSKsetArg()` and `TSKgetArg()` components. This mechanism is still used as the adaptation layer creates a "pre" function to the task's function. So when `osThreadDef()` is used, the function argument is memorized in the `osThreadDef_t` data structure. Also performed when `osThreadDef()` is used, the pre-function is created, which performs the extraction of the argument from Abassi's task descriptor through `TSKgetArg()` and then calls the task's real fucntin, passing the argument extracted with `TSKgetArg()`. The argument is set in the descriptor, using `TSKsetArg()` when the function `osThreadCreate()` is used and the function attached to the thread is the "pre" function..

2.3.4 Abassi components

When using the CMSIS adaptation layer, all Abassi's components are available, according to the setting of the build options. There is a one-on-one mapping of the CMSIS descriptors to Abassi's descriptors. The following table shows the mapping:

Table 2-4 Descriptors cross-reference

CMSIS	ABASSI
osThreadId	TSK_t *
osTimerId	TIM_t *
osMutexId	MTX_t *
osSemaphoreId	SEM_t *
osMessageQId	MBX_t *
osPoolId	n/a

`osMailQId`	n/a

The CMSIS priority numbering, through the `osPriority` enumeration, does not match the numbering used by Abassi. If priorities are involved when using native components of Abassi, the priority numbering must be re-mapped.

Two components are available to re-map the numbering between Abassi & CMSIS.

Table 2-5 Priority numbering transformers

Tranformer	Description
`PRIO_CMSIS_2_ABASSI(XXX)`	Transforms CMSIS priority value `XXX` to Abassi's numbering
`PRIO_ABASSI_2_CMSIS(XXX)`	Transforms Abassi's priority value `XXX` to CMSIS's enumeration

For example, to set the priority ceiling of a mutex to a priority of `osPriorityAboveNormal`, one would do the following:

Table 2-6: Priority remapping

```
   …
osMutexId MyMutex;
   …

   MTXsetCeilPrio(MyMutex, PRIO_CMSIS_2_ABASSI(osPriorityAboveNormal));

   …
```

The fact the Abassi native components are still available when using the CMSIS adaptation layer means the CMSIS standard can be extended in a non-standard way. It also means that all the features Abassi offers are still available. There are a lots of features that can be enable which don't require runtime configuration as their configuration can be specified solely by the build options. The most important are:

- Fixed time round robin
- Task starvation protection
- Mutex priority inheritance
- Mutex priority ceiling
- Mutex deadlock detection

3 Metrics

This section gives some information on the extra code size required when the CMSIS adaptation layer is used with Abassi; the numbers were obtained using a Cortex M3 as the target device. Latency measurements are not provided as the use of any adaptation layer degrades the intrinsic performance of the native RTOS. If real-time performance is critical, then Abassi's native components should be the preferred choice.

Table 3-1 CMSIS API Code Size Requirements

Compiler / Tools	Version	Optimization	**Code Size**
Code Composer Studio	5.2.0.00069	`-O 3 -mf 0`	`< 2000 bytes`
GCC (Atollic)	4.0.1	`-Os`	`< 2025 bytes`
IAR Embedded Workbench	6.30.6.3387	`Level High / Size`	`< 2000 bytes`
Keil uVision	4.50.0.0	`Level 3 (-O3)`	`< 1975 bytes`

NOTE: Smaller code size should occur if some CMSIS services are not used. This is possible, as most linkers will remove unused functions.

Abassi's CMSIS adaptation layer requires approximately the same size as the code used by Abassi itself. On most port the Abassi code size for the CMSIS feature set[1] is in the order of 2 KB. The reason the adaptation layer requires that much memory finds its origins in the need to select the appropriate function exit condition to report. The standard specifies for most CMSIS functions to report between 2 to 5 different exit conditions. Each of these conditions must be verified and the bulk of the code required lays in all these conditions, not in the use of the Abassi's components within the CMSIS adaptation layer functions.

[1] This is the minimum feature set for the CMSIS API. This means for example the priority inversion or the starvation protection are not part of the Abassi build.