Elektrobit

EB tresos® AutoCore OS architecture notes CORTEXM

product release 6.0

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

Europe	Japan	USA
Phone: +49 9131 7701 6060	Phone: +81 3 5577 6110	Phone: +1 888 346 3813

Support URL

https://www.elektrobit.com/support

Legal notice

Table of Contents

1. Overview of the Cortex-M architecture

This chapter gives a short overview of the Cortex-M architecture. It is limited to the most interesting points from a software developer's point of view. Detailed information can be found on www.arm.com.

1.1. Instruction sets

The Cortex-M family supports only instructions encoded within the Thumb/Thumb 2 instruction set. Switching to the ARM instruction set is not possible.

1.2. Register set

The Cortex-M architecture contains 13 general purpose registers `R0`-`R12`. Three additional registers have a special usage model:

`SP`
: also written `R13`, is the stack pointer and is used implicitly for instructions accessing the stack, e.g. `push`. The stack pointer is a banked register. This means that there is a separate register for each CPU mode. Depending on a control setting of the active CPU mode, the actual register is selected whenever `SP` is used within an instruction.

`LR`
: also written `R14`, denotes the link register. Function call instructions like `bl` or `blx` will store the return address in the link register.

`PC`
: also written `R15`, is the program counter and always points to the address of the currently executed instruction.

1.3. CPU modes

Cortex-M family CPUs may execute in two distinct CPU modes: Thread mode is the normal mode of operation. It can be configured to to either run privileged and unprivileged. In contrast, any kind of trap, for example interrupts or exceptions, utilizes the handler mode. Handler mode is always privileged.

1.4. Exceptions

The Cortex-M family implements eight different exceptions:

Reset
: The Reset exception is triggered if the CPU is reset.

NMI
: The NMI is the non maskable interrupt which can be triggered both by hardware and by software.

HardFault
: The HardFault exception denotes severe and often unrecoverable failures, for example an exception that occurs within the execution of an exception handler.

MemManage
: If a memory protection violation occurs, a MemManage exception is triggered. MemManage accounts for violations of both data and instruction memory transactions.

BusFault
: The BusFault exception is triggered if faults are detected during memory transactions, for example on the system buses.

UsageFault
: The UsageFault exception denotes errors during the execution of an instruction. Possible causes include division by zero, executing an undefined instruction or misaligned memory accesses.

Debug monitor
: The debug monitor exception is generated if a `BKPT` (software breakpoint) instruction is executed.

SVCall
: The execution of an `SVC` (supervisor call) instruction triggers a supervisor call exception.

Additionally Cortex-M family CPUs support two system level interrupts:

PendSV
: PendSV can be used for system calls. In contrast to the SVCall exception, this interrupt is triggered by setting a bit in a system register `ICSR`.

SysTick
: The SysTick interrupt is generated by the system timer, a 24 bit counter integrated into the Cortex-M processor core.

1.5. Memory protection

As an option, Cortex-M family CPUs support memory protection utilizing a memory protection unit (MPU). The memory protection unit allows to define access rights for a number of memory regions. For each region, the base address, region size and the associated rights have to be defined.

NOTE

In the scope of this document, if not mentioned otherwise, the term MPU refers to the ARMv7-M Optional Memory Protection Unit. Specific details on the ARMv7-M MPU can be found on www.arm.com

The memory region size has to be a power of two. The minimum supported region size is 32 bytes. Memory regions with a size larger than 128 bytes are divided into eight subregions. Each subregion inherits the access rights of the parent region but can be enabled or disabled individually.

2. Implementation details

This chapter lists implementation details of the EB tresos AutoCore OS that are specific to the Cortex-M port.

2.1. Instruction set support

Since the Cortex-M family only supports the Thumb/Thumb 2 instruction set, switching between instruction sets is not possible, see Section 1.1, “Instruction sets”.

2.2. Function call semantics

The operating system uses the standard ARM calling conventions as defined in the "Procedure Call Standard for the ARM architecture" (AAPCS).

- `R0-R3` and `R12` are volatile registers and are preserved by the function making a function call.
- `R4-R11` are non-volatile registers and are preserved by the function being called.
- `R0-R3` can be used to pass the first four arguments. Remaining arguments are passed using the stack.
- `R0` and `R1` are used as return registers.
- `SP` is used as stack pointer.
- `LR` is used as link register.

2.3. The stack pointer

Register `SP` is used as the stack pointer. Do not explicitly modify this register. The EB tresos AutoCore OS and the compiler manage the stack pointer for all tasks and category 1 and 2 interrupts in the system.

Tasks use the process stack as stack pointer. The operating system uses the main stack as stack pointer. Category 1 and 2 interrupts as well as hooks use the kernel stack as well.

2.4. CPU modes used by the operating system

Cortex-M family CPUs can run in two different modes, see Section 1.3, “CPU modes”. Additionally, thread mode allows to run code privileged or unprivileged. The current implementation of the EB tresos AutoCore OS supports privileged and non-privileged execution.

The following list gives an overview where each CPU mode is used within the EB tresos AutoCore OS:

Thread mode
: Thread mode is active after system reset. This means that the board startup code runs in thread mode until control is transferred to the EB tresos AutoCore OS by calling `StartOS()`.

  All tasks are executed in thread mode.

  Thread mode usually executes in privileged mode, except for non-trusted tasks. These execute in non-privileged mode.

Handler mode
: The CPU switches to handler mode whenever it takes an exception or services an interrupt. If exception handling is disabled by the configuration item `OsExceptionHandling`, exception handlers in the board support package are used (see also EB tresos AutoCore OS documentation). The exception handlers in the board support package are also executed in handler mode.

  Additionally, all hooks and interrupt service routines are executed in handler mode.

  Finally most of the kernel code is running in handler mode.

  Handler mode always executes in privileged mode.

2.5. Interrupt handling

The EB tresos AutoCore OS kernel configures all IRQs and handles all interrupts. The handling of interrupts is done by setting the appropriate registers of the interrupt controller. The operating system does not modify the (HW-)module-specific interrupt settings (e.g. the interrupt settings within a ADC module) outside of the interrupt controller.

You must assign one interrupt priority to each configured ISR and counter using the `OsCORTEXMIrqLevel` parameter.

Category 1 interrupts must have a higher priority than category 2 interrupts.

EB tresos AutoCore OS supports both category 1 and category 2 interrupts. User-defined interrupts can be automatically enabled at startup by setting the `OsEnable_On_Startup` attribute for the ISR to `TRUE`. (This is done by using the enable mechanism of the interrupt controller, the module specific handling is not done by the OS).

All interrupt routines are written in the following format:

```
ISR(isr_name)
{
  /* handling of interrupt */
```

```
  /* clear the interrupt flags */

  return;
}
```

The function body is written like a normal C function. The function should take care of the interrupt cause within the corresponding peripheral module. Not doing so may cause the very same interrupt request to be issued continuously.

2.5.1. Category 1 interrupts

For category 1 interrupts, the operating system stores the current context on the kernel stack before calling the ISR. OS API functions may not be called within category 1 ISRs.

2.5.2. Category 2 interrupts

Category 2 interrupts are handled by the kernel with a full wrapper function and potentially exit via the dispatcher. The generator configures the appropriate prologue and epilogue code to create an environment in which EB tresos AutoCore OS API functions may be called.

2.5.3. Nested interrupts

Nested interrupts are supported by the OS for both, category 1 and category 2 interrupts. Nesting of interrupts will occur according to the priority that is setup via the `OsCORTEXMIrqLevel` parameter.

2.5.4. Resources on interrupt level

Category 2 ISRs may use resources. They are supported by disabling all ISRs with a lower priority. As a consequence, a task may block ISRs if it takes a resource that is shared with ISR handlers as well.

2.6. Exception handling

The EB tresos AutoCore OS kernel handles all processor exceptions. The action taken depends on the severity of the exception, ranging from quarantining the task that caused the exception to complete system shutdown. To fully comply with the AUTOSAR requirements, the kernel must handle all CPU-generated exceptions.

2.6.1. Error reporting

The function `OS_GetErrorInfo()` returns the error information structure. When an error or protection hook is called as a result of an exception, the three *`parameter`* members of the error information structure contain the following information:

`parameter[0]`
: the program location at which the exception occurred.

`parameter[1]`
: the value of the corresponding status register. Which status register is evaluated depends on the exception that has been taken:

- `SHCSR` (system handler control and status register) for NMI, Debug, PendSV and SysTick exceptions. The value of this register is also present in the unexpected case, that a reserved slot in the exception table is jumped at.
- `HFSR` (hard fault status register) for HardFault.
- `CFSR` (configurable fault status) for MemManage, BusFault and UsageFault exceptions.

`parameter[2]`
: the value of the corresponding address register if applicable.

- `MMFAR` (mem manage fault address register) for MemManage exceptions.
- `BFAR` (bus fault address register) for BusFault exceptions.
- For all other exceptions, the value is `0`.

3. Memory protection using the NXP SMPU

3.1. Overview

The implementation of the memory protection architecture on Cortex-M CPUs is optional. Chip manufacturers, such as NXP, may leave the standard ARMv7-M MPU out and integrate a different means of memory protection. This chapter describes relevant implementation details of the NXP System Memory Protection Unit (SMPU) and how it is integrated into the Cortex-M architecture.

The distinguishing feature of the SMPU, in comparison to the ARMv7-M MPU, is its placement on the crossbar switch instead of being integrated into the CPU core. This allows the SMPU to control memory accesses from different bus masters: CPU core, DMA unit, Ethernet module or a debugger.

SMPU only protects Flash and SRAM memory regions. Access to peripherals, including the SMPU itself, is controlled by the Peripheral Bridge. In practice this means that all peripherals can be accessed only in privileged mode.

Depending on the derivative the SMPU supports 8 or 16 memory regions. Each memory region may span any range aligned to 32 Bytes up to 4 GB. For each memory region the words WORD0, WORD1, WORD2, WORD3 are configured.

`WORD0`
: contains the start address (lower bound) of the memory region, aligned to 32-Bytes.

`WORD1`
: contains the last address (upper bound) of the memory region.

`WORD2`
: contains the access permission flags for each bus master (`BM0-BM7`). Permissions for bus masters `BM4-BM7` are represented by 4 bits. Bus masters `BM0-BM3` have additional execute flag; their permissions are represented by 5 bits.

`WORD3`
: enables or disables the region. Process identifier and process identifier mask are not used by the EB tresos AutoCore OS

An access to a memory location which does not belong to a configured region results in an exception.

3.2. Implementation of memory protection

The operating system executes ISRs and application hooks of trusted application in handler mode which is a privileged mode.

Tasks of non-trusted applications are executed in user mode (unprivileged).

ISRs and application hooks of non-trusted applications are also executed in privileged handler mode.

NOTE

The SMPU is not able to protect kernel stack from non-trusted ISRs, because non-trusted ISRs use the kernel stack and run in handler mode. This is documented as a limitation. See also Chapter 5, “Limitations“.

The general rule for configuring SMPU regions states: "Granting permission is a higher priority than denying access for overlapping regions." According to this rule, the operating system uses the following memory regions:

Number	Usage	Core	Debugger	DMA	Ethernet
0	Entire address space (static)	`---` `---`	`rw-` `RW-`	`---` `---`	`---` `---`
1	Flash (static)	`r-x` `R-X`			
2	RAM (static)	`r--` `R--`		`r--` `R--`	`r--` `R--`
3	Kernel stack (static)	`---` `RW-`			
4	task/ISR stack (dynamic)	`rw-` `RW-`			
5	task/ISR data (dynamic)	`rw-` `RW-`			
6	application data (dynamic)	`rw-` `RW-`			

Table 3.1. MPU regions used by the EB tresos AutoCore OS

In the context of the table above the following words have a special meaning.

`static`
: The memory region is programmed once and never changed.

`dynamic`
: The memory region is programmed every time the OS switches to another task or ISR.

`[r,w,x,R,W,X,-]`
: Access rights are represented using 6 characters. The first 3 characters (in lower case) denote unprivileged access flags for reading, writing and executing. The second 3 characters (in upper case) denote the corresponding flags for the privileged access. The minus sign (−), denotes that the corresponding access type is forbidden.

The memory regions have the following meaning:

- Region 0 allows the debugger to read/write any location in the memory.
- Region 1 allows the OS kernel and all applications (tasks/ISRs) to read and execute code in the whole flash.
- Region 2 allows the OS kernel, all applications (tasks/ISRs), as well as the DMA and the Ethernet controllers to read data in the whole RAM.

- Region 3 allows the OS Kernel, the privileged (trusted) applications, and all ISRs to read and write to the kernel stack.
- Region 4 allows a non-privileged task/ISRs to read/write its private stack area.
- Region 5 allows a non-privileged task/ISR to read/write its global variables.
- Region 6 allows a non-privileged task/ISR to read/write the global variables of the associated application.

3.3. Configuring the system

The linker script for memory-protected systems on the Cortex-M architecture can be generated by the Perl program called `genld-CORTEXM.pl`, which can be found in the `bin` directory of the OS plug-in.

The configured region sizes are also passed to the Perl program to setup the required linker sections and the alignment.

To specify which object files belong to which application, set the corresponding make variables as described in the EB tresos AutoCore OS User's Guide section entitled "Generating the linker script with Perl".

The operating system evaluates section symbols defined by the linker (or defined in the generated linker script) to setup the MPU. Moreover, the symbols are used in the `board.c` file to initialize the global data and bss section.

The memory layout generated by the Perl script provides near-optimal protection. However, this comes at the cost of potentially leaving some areas of memory unused and unusable. In most real applications it will therefore be necessary to tune the linker script generation process, or hand-tune the linker script, to provide the best possible protection within the limitations of the processor or board.

NOTE If you write your own linker script, make sure that the linker symbols are correctly defined.

3.4. Memory access violations

When the SMPU detects a memory protection violation it generates a BusFault exception.

The cause for the access violation can be derived from the values of the fault status and fault address registers which are stored in the parameters of the global error information (cf. `OS_GetErrorInfo()`).

4. NXP S32K14X

4.1. Overview

The S32K14X is based on the Cortex-M4 family from ARM Ltd. It implements the ARMv7-M architecture. The derivative is able to process Thumb and Thumb 2 code. It contains a hardware floating point unit (FPU).

S32K14X is not configured with the optional ARMv7-M Core-MPU, and instead it is equipped with the NXP's System-MPU. For more details about the NXP MPU see Chapter 3, “Memory protection using the NXP SMPU“.

Specific details on the ARMv7-M architecture can be found on www.arm.com.

4.2. Memory map

The S32K14X implements a Harvard bus architecture in the following way:

- the low-order memory `0x00000000-0x1FFFFFFF` is accessed through the code bus and therefore can be used for storing program instructions;
- the high-order memory `0x20000000-0xFFFFFFFF` is accessed through the system bus and does not allow execution of the memory content.

The complete flash lies within the low-order memory. The RAM is split in two contiguous regions, extending up and down from the `0x20000000` address. Because of this, burst read and write accesses should not cross the `0x20000000` boundary.

4.3. Interrupt handling

The Cortex-M4 family uses the "NVIC" (nested vectored interrupt controller) to manage interrupt requests. For each possible interrupt request source, a vector entry exists that contains the address of a handler routine. The interrupt vector table starts immediately after the exception vector table. The interrupt controller only supports hardware vector mode, which means that the CPU will automatically branch to the vector address if an interrupt request was triggered.

The EB tresos AutoCore OS will automatically create wrapper code for all configured ISRs and place the corresponding addresses in the interrupt vector table.

The NVIC implementation found in this derivative supports up to 147 distinct interrupt vectors. Each interrupt vector corresponds to a specific interrupt source. The OS maps each configured interrupt service routine to one interrupt vector.

You must assign an interrupt priority in the range of `1` to `15` for each ISR and counter using the `OsCORTEXMIrqLevel` parameter. The value `1` denotes the highest priority. The same priority may get assigned to multiple ISRs.

4.4. Timer units

The EB tresos AutoCore OS contains a timer driver for the Low Power Interrupt Timer "LPIT".

The LPIT timer module uses BUS_CLK as clock source. The EB tresos AutoCore OS programs its internal prescaler to the value set by the macro `BOARD_CORTEXM_LPIT_PRESCALER`.

The LPIT timer module has 4 timer channels, three of which (channel-0, channel-1 and channel-2) are available for the user to configure time-triggered events, while channel-3 is used by the OS and thus reserved.

4.5. Board support package

The board support package contains files that perform the initial startup and initialization of the MCU. It is provided in source-code form. Using this as a starting point it should be possible to construct startup code for your own board in a straightforward manner.

Note that the EB tresos AutoCore OS relies on some configuration being performed during startup. Carefully evaluate the shipped example files from the board support package. Additionally, section Section 4.5.6, “Required configuration for startup files” contains a list with common requirements.

4.5.1. Directory structure

The board directories are located in the EB tresos Studio installation directory under `demos/AutoCore_OS/os_demo_<derivative>_<version>/source/boards/<boardname>`. Each board directory contains source files for the board support package (BSP) and a makefile fragment. Optionally, the directory may contain linker and debugger files as well. Typically boards do have a:

- `board.h`
- `board.c`
- `boardStartup.s`
- `<boardname>.mak`

4.5.2. Header File: board.h

The EB tresos AutoCore OS kernel configuration files include the board.h header file to obtain values for board-specific parameters.

4.5.2.1. Board macros for S32K14X

board.h defines the following macros:

BOARD_BUS_CLK_FREQ
: This definition contains the frequency of BUS_CLK (the frequency at which the system timer module runs).

BOARD_CORTEXM_LPIT_PRESCALER
: This define specifies the prescaler value that is used for the LPIT timer. BOARD_CORTEXM_LPIT_PRESCALER is divided by this prescaler value.

OS_BoardLPITNsToTicks
: This macro converts nanoseconds to system timer ticks.

OS_BoardLPITTicksToNs
: This macro converts system timer ticks to nanoseconds.

4.5.3. Source File: boardStartup.s

The assembly-language source file boardStartup.s contains the small startup code which is executed after reset. Moreover, it contains stub routines for invalid interrupts and invalid exceptions.

4.5.4. Source File: board.c

The C source file board.c contains the function BoardStart().

BoardStart() initially configures the CPU clock settings. Moreover, the operating system does not rely on compiler startup-code regarding the initialization of the data and bss memory sections. This is also done in BoardStart(). Finally, any further initialization is performed in this function.

4.5.5. Make Files: <boardname>.mak

The files <boardname>.mak is a makefile fragment which holds information which (board specific) files shall be compiled.

4.5.6. Required configuration for startup files

If you want to customize the startup files, make sure to provide the following configuration and functionality before calling the function `StartOS()`. Otherwise the EB tresos AutoCore OS may not work properly. Note that this list may not be exhaustive. It is important that you carefully evaluate the shipped example files in the board support package.

- The CPU frequency shall be configured in an appropriate manner.
- The data and bss memory sections shall be initialized.
- The timer specific definitions and macros shall be provided in the header `board.h`.
- The symbol `boardResetStart` shall be provided. It serves as the initial entry point after the MCU has been reset.
- The main stack pointer shall be initialized with a suitable value (e.g. top of kernel stack `OS_kernStack0`).
- The FPU shall be enabled and automatic saving of the FPU context shall be disabled. You can use the function `OS_CORTEXM_EnableVFPCoprocessor()` for this purpose.

See also the EB tresos AutoCore OS documentation.

5. Limitations

- `OsTrappingKernel`: The current implementation of the EB tresos AutoCore OS is restricted to a trapping kernel. This means that OS services are always called using a system call mechanism.
- Kernel stack is not protected against untrusted ISRs: On all derivatives with Cortex-M Architecture ISRs are executed in handler mode, which is always privileged. On Cortex-M derivatives that integrate the ARMv7-M MPU it is possible to protect the kernel stack and allow untrusted ISRs to write to their own stack region. However, it is practically impossible to achieve this for derivatives with the NXP MPU.

Appendix A. Error codes

A.1. Generator error codes

Cortex-M-specific error codes are listed here. For architecture-independent error codes, see the EB tresos AutoCore OS User's Guide.

1. Errors

Code	Description
_10000	*The SW counter {0} has the HW incrementer {1}, but does not drive a simple schedule table. Please reference {0} from a simple schedule table or disable the HW incrementer.* A software counter with a HW incrementer must drive a simple schedule table. Please make sure, that a simple schedule table references this counter or disable the HW incrementer.
_10001	*The interrupt with vector {0} has a configured priority of 1. In a configuration with activated Microkernel, Safety Os on CORTEXM only supports interrupt priorities that are numerically higher than 1.* Configure a priority numerically higher than 1.

A.2. Kernel error codes

Please refer to the AUTOSAR OS Users Guide.