Elektrobit

EB tresos® AutoCore OS architecture notes TRICORE

product release 6.1

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

https://www.elektrobit.com/support

Legal disclaimer

Table of Contents

Begin here

1. Overview

These architecture notes provide details of the target hardware as it is used by EB tresos AutoCore OS. It is assumed that you are already familiar with the contents of the related user documentation [ASCOS_DOCUMENTATION].

In the following list, you find a summary of each chapter in this document with recommendations for reading. You should read some chapters in full while others can be selectively read depending on your derivative as indicated:

- Chapter 1, “Overview of the TriCore architecture“ provides an overview of the aspects of the target hardware architecture that are relevant for EB tresos AutoCore OS. You should read this chapter in full.
- Chapter 2, “Supported derivatives“ provides details about the individual derivatives that EB tresos AutoCore OS supports. Each derivative lists the memory protection and timer hardware that is supported. You should read the sections of this chapter that are relevant for the target hardware that you use in your project.
- Chapter 3, “Timers“ describes the timers supported by EB tresos AutoCore OS. You should read the sections for the timers that are listed for your derivative.
- Chapter 4, “Memory protection“ describes the memory protection hardware and implementation. You should read the sections that are listed for your derivative.
- Chapter 5, “Implementation details“ provides details of the implementation of EB tresos AutoCore OS on the target hardware. You should read the parts of this chapter that are relevant for your needs.
- Chapter 6, “Reference manuals used“ lists the hardware reference manuals that were used during the implementation of EB tresos AutoCore OS. The bibliography gives further details about the documents, including revision and publication date where available.

2. Typography and style conventions

The signal word *WARNING* indicates information that is vital for the success of the configuration.

WARNING ⚠	**Source and kind of the problem** What can happen to the software? What are the consequences of the problem? How does the user avoid the problem?

The signal word *NOTE* indicates important information on a subject.

NOTE ⓘ	**Important information** Gives important information on a subject

The signal word *TIP* provides helpful hints, tips and shortcuts.

TIP 💡	**Helpful hints** Gives helpful hints

Throughout the documentation, you find words and phrases that are displayed in **bold**, *italic*, or `monospaced` font.

To find out what these conventions mean, see the following table.

All default text is written in Arial Regular font.

Font	Description	Example
Arial italics	Emphasizes new or important terms	The *basic building blocks* of a configuration are module configurations.
Arial boldface	GUI elements and keyboard keys	1. In the **Project** drop-down list box, select Project_A. 2. Press the **Enter** key.
Monospaced font (Courier)	User input, code, and file directories	The module calls the `BswM_Dcm_RequestSessionMode()` function. For the project name, enter `Project_Test`.
Square brackets [ ]	Denotes optional parameters; for command syntax with optional parameters	`insertBefore [<opt>]`
Curly brackets {}	Denotes mandatory parameters; for command syntax with mandatory parameters	`insertBefore {<file>}`

Font	Description	Example
Ellipsis …	Indicates further parameters; for command syntax with multiple parameters	`insertBefore [<opt>…]`
A vertical bar |	Indicates all available parameters; for command syntax in which you select one of the available parameters	`allowinvalidmarkup {on|off}`

1. Overview of the TriCore architecture

This chapter gives a short overview of the OS-related parts of the TriCore architecture. For an in-depth understanding of the architecture, see the *TriCore TC1.6P & TC1.6E User Manual (Volume 1)* [TC_ARCH_161_VOL1] and *TriCore TC 1.6.2 core architecture manual (Volume 1)* [TC_ARCH_162_VOL1]. For derivative-specific information, see the corresponding manuals for the derivative that your project uses, for example, user manual, target specification, or data sheet.

TriCore microcontroller units (MCUs) have one or more 32-bit microprocessor cores. There are many different derivatives of TriCore, all more or less compatible with each other at the processor level. They differ in the exact core implementation and the variety and location of peripherals and internal memory.

The EB tresos AutoCore OS kernel is written in such a way that the source code should be compatible with most, if not all, TriCore derivatives. This can be ensured using a common TriCore implementation design. As a result, we prefer common code to code that is heavily optimized but can be used by only one derivative.

The core-specific differences are implemented as transparently as possible. Switching between derivatives should be possible without any change at the application level if the application code does not deal with any derivative-specific hardware.

Ensure that you compile the software with the correct compiler switches, including switches for the CPU core and macro definitions to achieve correct execution on the target.

If you use precompiled kernel and user libraries, make sure that these match the compiler switches documented for the specific derivative.

1.1. The CPU core

The cores of the Infineon TriCore AURIX family are based on the TriCore TC1.6.1/TC1.6.2 series cores and provide a compatible instruction set. Depending on the specific derivative, parts of the TriCore core's features may be disabled or may differ slightly from the family specification.

1.2. The general-purpose registers

TriCore processors have 16 data registers `D0..D15` and 16 address registers `A0..A15`. All the registers are 32 bits wide and can be used as source and destination in many processor operations.

The processor supports a limited number of instructions with 64-bit parameters. For those instructions the data registers can be used pairwise as 64-bit registers, for example, `D0/D1`, or `D2/D3`. The mnemonic `En` is used to represent the pair `Dn/Dn+1`.

Four of the address registers `A0`, `A1`, `A8`, and `A9`, are designated as global registers for purposes such as base pointers. `A10` is the stack pointer and `A11` is the return-address register.

The return-address register `A11` is automatically filled by the processor whenever a function is called, or an interrupt or exception is accepted. The previous value is automatically saved. In the case of `FCALL`, this will be on the stack. Otherwise, it is done by the context management system of the processor.

1.3. The system registers

TriCore contains several system registers. Those that are relevant to EB tresos AutoCore OS are discussed here. See the TriCore user manuals for full description of the derivative that you are using.

`PSW`
: The processor status word (`PSW`) contains the memory protection set selector, the processor mode setting, and the global register write enable flag.

`SYSCON`
: The system configuration register (`SYSCON`) contains the protection enable flags that enables the use of the memory protection registers.

`ICR`
: The interrupt control register (`ICR`) contains the current CPU priority number (`CCPN`) and the interrupt enable flag (IE) and thus controls the interrupt level above which interrupts are enabled. The interrupt level is used as an arbitration priority.

`ISP`
: The interrupt stack pointer (`ISP`) contains the initial value of the kernel stack pointer. The processor automatically copies this to the stack pointer when the kernel is first entered via system call, interrupt, or exception.

`PCXI, FCX,` and `LCX`
: The `PCXI` register references the head of a linked list of context-save areas (CSAs), which contain the saved state of the processor at each function call and interrupt. There is also a linked list of free CSAs whose head is indexed by the `FCX` register. The `LCX` register determines the end of the free list, reserving enough CSAs to be able to handle the resulting exception. EB tresos AutoCore OS and the processor itself manage the CSA lists for all tasks and ISRs.

`BIV` and `BTV`
: The base interrupt vector (`BIV`) and base trap vector registers (`BTV`) contain the addresses of the interrupt and exception vector tables.

`DPRxx, DPM, CPRxx,` and `CPM`
: The code and data protection registers define the memory regions that can be read, written, and executed.

All the above registers are managed by the EB tresos AutoCore OS kernel. Your application must not explicitly modify any of these registers.

1.4. Main memory

TriCore processors are equipped with multiple memory blocks, having non-uniform access behavior. Every processor core has a so-called *data scratchpad* memory (DSPR) that is closely coupled to the core and has the fastest access time.

1.5. Caches

All TriCore processor cores are equipped with an instruction cache. TC1.6P and TC1.6.2 cores are additionally equipped with a data cache, but TC1.6E cores are not. The data cache is physically located in the same memory as the DSPR; hence it is neither necessary nor possible to have cached memory accesses to the DSPR of the core performing the access. There is no hardware coherency between the data caches and the respective memory. So, if another bus master changes a line of cached memory, the cache still retains the old values. Other bus masters could be, for example, a different core, or a DMA controller.

1.6. Floating-point unit

Some TriCore derivatives provide an FPU. This FPU does not have dedicated floating-point operand registers. Instead, the normal general-purpose registers are used. Therefore, EB tresos AutoCore OS provides no special support for tasks that are configured to use the FPU.

EB tresos AutoCore OS itself does not use floating-point operations.

1.7. Memory protection

Each TriCore CPU core provides a memory protection unit (MPU). Depending on the derivative, there is support for up to 18 data and 10 code memory protection regions. A description of memory protection regions configured by EB tresos AutoCore OS can be found here: Chapter 4, “Memory protection“

1.8. Interrupt controller

The interrupt controller provides a priority-based interface between the interrupt sources in the MCU and the interrupt mechanism of the CPU core. Each interrupt source can be given an interrupt priority ranging from the lowest level 1 to the highest level 255.

The interrupt controller contains an interrupt control register (`ICR`), a base interrupt vector register (`BIV`) and a several service request nodes (`SRNs`), each of which has a service request control register (`SRC`).

The interrupt controller registers can be divided into two categories:

- Registers that are relevant to the whole interrupt controller, for the cores on which EB tresos AutoCore OS runs. These registers are reserved for EB tresos AutoCore OS.
- Registers that are specific to individual interrupt sources. Under some circumstances your application might need to write to these registers; for example, when you need to trigger a software interrupt.

If your application writes to the interrupt source registers, you must ensure that you observe the following restrictions:

- For the interrupt sources that are configured for EB tresos AutoCore OS, you must ensure that you do not change the core assignment, priority, and enable/disable status.
- For the interrupt sources that are not configured for EB tresos AutoCore OS, you must ensure that you do not set the core assignment to a core on which EB tresos AutoCore OS is configured to run.
- You must ensure that the software that modifies the registers does not cause race conditions, either in the hardware or with EB tresos AutoCore OS.

2. Supported derivatives

EB tresos AutoCore OS supports the TriCore derivatives listed in this chapter.

2.1. TC32XL

The TC32XL, a representative of Infineon's AURIX2G line of processors, contains one core with an additional lockstep core. Lockstep cores are also called *checker cores* in the TriCore documentation. Lockstep cores are used to detect sporadic faults in their corresponding processor cores. For a detailed overview of the core, see [TC3XX_TS_1_210].

Table 2.1, "TC32XL hardware units used" lists the hardware units used by EB tresos AutoCore OS. For the details about all the peripherals provided by the TC32XL, see [TC3XX_TS_1_210].

Core	TC1.6.2P
MPU	18 data regions and 10 code regions
Interrupt Controller (IR)	255 service request priority levels
Timers	1 STM instance named STM0
Core ID mapping	not applicable
Inter-core interrupts	not applicable

Table 2.1. TC32XL hardware units used

2.2. TC38XQ

The TC38XQ, a representative of Infineon's AURIX2G line of processors, contains four processor cores:

- Cores 0 and 1 are equipped with an additional lockstep core.
- Cores 2 and 3 do not come with a lockstep core.

Lockstep cores are also called *checker cores* in the TriCore documentation. Lockstep cores are used to detect sporadic faults in their corresponding processor cores. Cores 2 and 3 may not be suitable for use in safety applications for high ASILs without additional measures. Both the read-only flash memory and the read/write data memory are equipped with error-detection features to detect transient bit errors in the memory. For a detailed overview of the core, see [TC38X_TS_V251].

Table 2.2, "TC38XQ hardware units used" lists the hardware units used by EB tresos AutoCore OS. For the details about all the peripherals provided by the TC38XQ, see [TC38X_TS_V251].

Core	TC1.6.2P
MPU	18 data regions and 10 code regions
Interrupt Controller (IR)	255 service request priority levels
Timers	4 STM instances named STM0, STM1, STM2 and STM3
Core ID mapping	▶ Core 0: CPU0 ▶ Core 1: CPU1 ▶ Core 2: CPU2 ▶ Core 3: CPU3
Inter-core interrupts	▶ Core 0: GPSR20 ▶ Core 1: GPSR21 ▶ Core 2: GPSR22 ▶ Core 3: GPSR37

Table 2.2. TC38XQ hardware units used

3. Timers

3.1. System Timer Module (STM)

EB tresos AutoCore OS offers a driver for the STM with compare registers. The timer named STM[stm_id]_T0 uses compare register 0 and the timer named STM[stm_id]_T1 uses compare register 1. These timers can be selected as the underlying timer when you configure a hardware counter in the OS configuration.

When execution-budget monitoring is enabled, one of the STM units' compare registers is used for the purpose of interrupting the task or ISR when it exceeds its allotted budget. The selection can be either STM[stm_id]_T0 or STM[stm_id]_T1 and is done via the OS parameter OsTricoreExecutionTimer. The timer selected for this purpose cannot be used with a hardware counter.

The STM timer used for timestamp functionality can be selected using the OsTimestampTimer parameter. It uses registers TIM0 and TIM6 to give a 64-bit time. The timer selected for this purpose can also be used with a hardware counter or for execution budget monitoring.

EB tresos AutoCore OS makes use of following registers of the STM unit to provide a timer driver:

- Timer Register 0 (TIM0)
- Compare Register x (CMPx)
- Compare Match Control Register (CMCON)
- Interrupt Control Register (ICR)
- Interrupt Set/Clear Register (ISCR)

EB tresos AutoCore OS is responsible for configuring the corresponding interrupt source and handling the timer interrupt.

4. Memory protection

4.1. Memory protection using the MPU

EB tresos AutoCore OS executes tasks, ISRs, and application hooks of non-trusted applications with restricted memory access. Each access violation is handled by the OS, which calls the protection hook to notify the application.

In trusted mode, the MPU is set to grant full access to all valid memory areas. Consequently, the OS itself and all trusted objects are executed without memory restrictions.

Data and code protection is controlled by a several register pairs, each pair consisting of lower and upper bound registers, describing a single memory region. These memory regions are mapped to so-called protection register sets (PRS). Each of those register sets represents a subset of all the available data and code protection regions. Read, write, and execute permissions can be defined for each region within a PRS. The rights may differ for the same region in different PRSs.

The active PRS is selected by a field in the `PSW` and is independent of the processor mode. EB tresos AutoCore OS for TriCore uses PRS0 for trusted and PRS1 for non-trusted mode.

At OS start, all memory regions are invalidated before the first configuration. EB tresos AutoCore OS initializes the private data of OS applications and objects.

If an OS object or application has no private data, the corresponding MPU region is disabled. This is detected by the linker symbols as described in the EB tresos AutoCore OS documentation [ASCOS_DOCUMENTATION].

The following region descriptors are used by EB tresos AutoCore OS:

Descriptor	**PRS**	**Usage**
`CPR0`	PRS0	*<start of physical address space>* to 0xffffffff, set up statically during MPU initialization
`CPR1`	PRS1	entire .text section, execute, set up statically during MPU initialization
`DPR0`	PRS1	application data, read/write, changed dynamically on context switches
`DPR1`	PRS1	task/ISR private data, read/write, changed dynamically on context switches
`DPR2`	PRS1	stack, read/write, changed dynamically on context switches
`DPR3`	PRS1	global data, read-only, set up statically during MPU initialization
`DPR4`	PRS0	whole memory, read/write, set up statically during MPU initialization

Table 4.1. Region descriptors

5. Implementation details

5.1. Byte ordering

The TriCore family is little-endian.

5.2. Processor mode

If you configure non-trusted applications, EB tresos AutoCore OS enables and uses TriCore's memory protection unit by setting the appropriate bit in the `SYSCON` register.

Code used by trusted applications, category 1 ISRs, and the kernel itself, is considered to be trusted code. Trusted code runs in SUPERVISOR mode and thus can use the peripherals, CSFRs, and supervisor-mode instructions. Protection register set 0 is used for trusted code and is programmed during the startup phase to permit read/write access to all physical memory and execute access to all physical code memory.

For non-trusted applications, you can choose to run the processor in USER0 mode or in USER1 mode by setting the parameter `OsAppCpuMode` to the desired value. By default, non-trusted applications run in processor mode USER0. See [TC_ARCH_162_VOL1] for a detailed description of processor modes. Protection register set 1 is used for non-trusted applications. This is programmed for the currently running task, ISR, or hook function whenever such an executable is started or resumed.

5.3. Function call semantics

TriCore's EABI allows for two different ways of passing parameters to functions.

Using the register calling convention, parameters are passed in registers as far as possible; pointer parameters are passed in address registers `A4-A7` and scalar parameters are passed in data registers `D4-D7`. Any excess parameters, and all parameters following an ellipsis, are passed on the stack.

Using the stack calling convention, all parameters are passed on the stack regardless of type.

EB tresos AutoCore OS uses the register calling convention exclusively. The stack calling convention is not used. Furthermore, the system-call mechanism does not support any stack parameters at all, so system calls are limited to a maximum of 4 scalar and 4 pointer parameters with no ellipsis.

5.4. The stack pointer

Address register `A10` is used as the stack pointer. The application should not explicitly modify this register. EB tresos AutoCore OS and the compiler handle the stack pointer management for all the tasks and interrupts in the system.

TriCore's EABI demands that the stack grows downwards from a high address with pre-decrement. That means that the initial value of the stack pointer may lie outside the permitted memory region of the stack. Also, due to the large number of registers and the CSA mechanism for handling call and return, it is very common for functions to use no stack at all. That means that the stack pointer can retain its initial value even in quite deeply nested function calls.

5.5. Context save areas

TriCore processors have a unique way of handling nested function calls and interrupts. In addition to a normal stack, the processor maintains a linked list of context-save areas (CSAs). Each CSA stores 16 32-bit registers, and thus is 64 bytes in length. The first register location in each CSA is the previous `PCXI`, and the `PCXI` register contains an index to the head of the stored CSAs, thus a linked list is formed. When a subroutine is called or an interrupt is taken, the next CSA is taken from the free list, the upper context registers are written to it, and its index is stored in `PCXI`. Returning from the subroutine or interrupt reverses this procedure.

The upper context registers include the stack pointer `A10` and the return address `A11`, thus a stack-frame mechanism is automatically implemented. It is also possible to save and restore the lower-context registers using `SVLCX` and `RSLCX` instructions. The `BISR` instruction combines `SVLCX` with an interrupt enabling and priority setting instruction.

The free CSA list must be constructed by software before any subroutines can be called or interrupts handled. EB tresos AutoCore OS provides a special function, `OS_InitCsaList()`, to do this. This function is called from the assembler startup code using a `JL` instruction and returns using a `JI` instruction. The function therefore does not require a CSA and must never be called from normal C-code.

When the processor uses all the CSAs on the free list, i.e. `FCX==LCX`, an exception is generated by the processor. Therefore a few CSAs are reserved past the `LCX` register to handle this exception.

5.6. Exception handling

The EB tresos AutoCore OS kernel handles all processor exceptions. The action taken will depend 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.

Furthermore, the system-call i.e. class 6 exceptions, and context-list underflow i.e. class 3 TIN 5 exceptions, are used for task and ISR control purposes to switch from user mode to supervisor mode. The `BTV` register must therefore point to the exception vector table provided by the kernel. If the `BTV` register is modified, the kernel will be effectively disabled, and the system will not function correctly.

The NMI exception is handled by a C-function called `OS_Trap7Handler()`. This function reads the cause of the trap from the `TRAPSTAT` register and clears it by writing the `TRAPCLR` register. Afterwards, a panic is raised, which causes EB tresos AutoCore OS to shut down.

If you wish to use NMI for application-specific purposes, you can write your own function `void OS_Trap7Handler(void)` to override the one in the kernel library.

5.7. Interrupt handling

This section provides information on the architecture-specific interrupt handling. For general information see the EB tresos AutoCore OS documentation [ASCOS_DOCUMENTATION].

The interrupt level specified in the OS configuration is only used as a guideline by the generator. The actual levels used will be optimized to minimize the size of the vector table and to allow the possibility of reducing the number of arbitration cycles. The generator ensures that all category 1 interrupts have higher priority than category 2 interrupts. Kernel interrupts such as that used for detecting when the execution-time budget has been exceeded have priority higher than category 2 but lower than category 1.

The arbitration priorities for interrupts are allocated in ascending order of the configured level, starting at 1. The arbitration priorities are the levels programmed into the service request registers. Interrupts with the same configured level will have different arbitration priorities because TriCore has no native support for multiple interrupts at the same arbitration priority. The run priority of the ISRs is also determined automatically and is equal to the highest arbitration priority among interrupts with the same configured level. This means that the configured level works as expected - during an interrupt service routine all interrupts of equal or lower configured level are locked out.

The level acknowledge to the interrupt controller is automatically done by the CPU.

5.7.1. Nested interrupts

Nested interrupts are supported by this architecture. Nesting of category 2 interrupts occurs according to the priority levels that are configured via the `OsTricoreIrqLevel` parameter. Category 1 interrupts nest in a similar manner, and additionally interrupt any category 2 or kernel interrupt, unless interrupts are disabled. If an interrupt is pending with a higher priority than the interrupt currently being serviced, then the interrupt being serviced is preempted.

While the kernel is executing the interrupt prologue and epilogue, the current CPU priority number (`CCPN`) is set to kernel lock level. Right before the user code is called `CCPN` is set so that the interrupt controller only accepts levels higher than the currently processed.

5.8. The startup phase

5.8.1. Startup order

This section describes in general terms the sequence needed to start EB tresos AutoCore OS.

- Physical core 0 is started automatically.
- Ensure that the interrupts are disabled and an initial value is set to the `PSW`.
- Ensure that the stack pointer is set to the Startup stack.
- Ensure that the CSA list in the internal memory is initialized.
- Ensure that the CPU and the safety ENDINIT bits are cleared. `OS_WriteEndinit()` and `OS_WriteSafetyEndinit()` can be used for this purpose.
- The clocks and PLLs should be set up. It is also possible to use the default clock settings provided by the hardware and to configure the clocks/PLLs later after `main()` has been called. The late configuration of clocks/PLLs might impact the startup timing. The EB tresos AutoCore OS kernel does not provide functionality to set up clocks/PLL. The initialization has to be done by user code.
- Ensure that the cache is set up. `OS_SetupCache()` can be used for this purpose.
- Ensure that the base trap vector register (`BTV`) is set to the trap table `OS_startupTrapTable`.
- Ensure that the interrupt stack pointer register (`ISP`) is set to its correct working value.
- Ensure that the `.data` section is initialized from its ROM image and that the `.bss` section is cleared. Note that the private `.data` and `.bss` areas of OS applications are not initialized at this point.
- Ensure that the STM modules are initialized.
- Ensure that the global address register `A8` is set to core 0's entry in `OS_kernel_ptr`.
- Call `main()`.
- Now, all the system-specific initialization can be performed. In particular, all peripheral modules that generate interrupts must be enabled here by having their clock control registers set to the working value. Fail-

ure to do this means that EB tresos AutoCore OS cannot initialize the interrupt service request registers in the module and may fail with an exception when it attempts to do so.

- In a multi-core system, other cores managed by EB tresos AutoCore OS need to be started using the `StartCore()` system service that will in turn start executing the `main()` function on those other cores.
- Call `StartOS()`. This service starts the EB tresos AutoCore OS kernel and under normal circumstances never returns.
- In a multi-core system, call `StartOS()` for all configured EB tresos AutoCore OS cores.

5.8.2. Startup stack

The OS generates a kernel stack located in a section named `.bss.core<x>.os_kernstack_*` where `<x>` is the ID of each core used. For single-core systems, this will be `0`. It is assumed that the system starts on this stack. A linker symbol like `OS_INITIAL_SP<x>` is defined, pointing to the end of the section, and used for initialization of the stack pointer.

Changing the startup stack is possible by changing the linker script. This may be necessary if the kernel stack is not sufficient for user-supplied startup code. Alternatively, you can increase the size of the kernel stack by adding a dummy ISR or increasing the stack size of an existing ISR.

5.9. Idle task

If there is no task ready to run, the system starts an internal idle task.

The idle function results in an endless loop with global interrupts enabled. Rescheduling is triggered as soon as an interrupt or exception is signaled by the CPU.

Besides the default idle mode `IDLE_NO_HALT`, there is an additional idle mode `OS_IDLE_WAIT` for the TriCore architecture. When you set a core to the idle mode `OS_IDLE_WAIT`, the endless idle loop executes the `wait` instruction after it has enabled the interrupts.

You can control the behavior of the idle task using `ControlIdle()`. See the API reference for `ControlIdle()` in the EB tresos AutoCore OS documentation [ASCOS_DOCUMENTATION].

5.10. Error reporting

When an `ErrorHook()` or `ProtectionHook()` is called as a result of a trap, the three parameter members of the structure provided by `OS_GetErrorInfo()` contain the following information:

- `parameter[0]`: the program location where the trap occurred
- `parameter[1]`: the trap class
- `parameter[2]`: the trap identification number (TIN)

Trap class and TIN are fully described in the processor documentation. For asynchronous traps, the program location in `parameter[0]` might not be the address of the instruction that caused the exception.

5.11. Linker script and sections

EB tresos AutoCore OS uses a certain naming convention for linker sections that contain its own objects. This allows arranging those objects more easily in memory via a linker script, for example to facilitate memory protection. Some parts of this naming convention are similar to what is used by the majority of toolchains, to designate implicitly and explicitly initialized data sections, for example the `.bss` and `.data` sections. Other parts are custom, for example the `.shared` or `.core<x>` sections. The `.shared` section is used to mark a linker section as shared and thus must be accessible by all cores. A subsequent `.core<x>` section may be used to indicate where the section should be preferably located if core `x` can access some memories faster than others. If instead `.os` follows the word `.shared,` the data in such a section is still shared but has no preferred association with a specific core. On the other hand, if `.shared` is not part of a linker section's name, its `.core<x>` part now becomes mandatory and designates the core which has exclusive access to the data in that section and thus the section must be placed in memories tightly coupled to core `x`. Table 5.1, "OS linker sections" contains a list of the linker sections which must be handled specially in the linker script as just described.

Description	**Linker section**
startup trap table	`OS_StartupVectors`
interrupt vector table	`OS_InterruptVectors`
exception vector table	`OS_ExceptionVectors`
OS kernel stack	`.bss.core<x>.os_kernstack*`
OS task stacks	`.bss.core<x>.os_taskstack*`
explicitly initialized data	▶ `.data.core<x>.*` ▶ `.data.shared.core<x>.*` ▶ `.data.shared.os.*`
implicitly initialized data	▶ `.bss.core<x>.*` ▶ `.bss.shared.core<x>.*` ▶ `.bss.shared.os.*`

Table 5.1. OS linker sections

The linker section `OS_ExceptionVectors` accommodates the exception vector table. Each entry contains up to 32 bytes of code. The exception vector table start address must be aligned on a 256-byte boundary, and each individual entry vector must be aligned on a 32-byte boundary.

Furthermore, the EB tresos AutoCore OS generator generates an array for each core which is intended to be used as OS kernel stack. These stacks are used during the startup phase and to process interrupts, exceptions, and hooks. All EB tresos AutoCore OS API functions also use these stacks. The names of the sections are `.bss.core<x>.os_kernstack*` for the OS kernel stack of core `x`.

Tasks, on the other hand, use different *stack slots*. These stack slots are used to optimize memory consumption of tasks that do not preempt each other and therefore can share the same stack. They are located in linker sections named `.bss.core<x>.os_taskstack*` for tasks that run on the core `x`.

Explicitly initialized data, that is the set of C-language objects for which initial values are explicitly specified by the developer, are located in linker sections whose names start with `.data.` On the other hand, the linker sections that begin with `.bss` accommodate all those C-language objects which lack explicit initial values and thus are initialized with zero. For either kind, the names may contain further custom parts to designate whether they are shared by all cores in the system, or whether they are tightly coupled to one of them. This naming convention is explained at the beginning of this section.

See EB tresos AutoCore OS documentation [ASCOS_DOCUMENTATION] for additional information about linker scripts and the requirements memory protection imposes on them.

5.12. Atomic functions support

The use of atomic functions for EB tresos AutoCore Generic is supported by the atomics module that is provided by EB tresos AutoCore Generic Base. For a standalone delivery of EB tresos AutoCore OS the underlying atomics functions may be used directly.

5.12.1. Usage restrictions

5.12.1.1. Atomic functions

The atomic functions on TriCore do not take caches into account. Consequently, it is necessary to put all atomic objects with type `os_atomic_t` into non-cacheable memory or to disable data caches completely. This ensures that all threads of execution can observe each other's memory accesses.

Note that the implementation uses specialized instructions to work on atomic objects, for example, `cmpswap.w` or `swapmsk.w`. These instructions require that objects of type `os_atomic_t` are naturally aligned.

5.12.1.2. Hardware specific constraints for `EB_FAST_LOCK`

For the TriCore family, EB tresos AutoCore OS features an optimized implementation to ease the overhead incurred by entering and leaving exclusive areas. This optimization is effective only for the processor modes `USER1` and `SUPERVISOR`. Consequently, this optimization doesn't affect program execution in processor mode `USER0`. For further information about processor modes, see the configuration parameters `OsAppCpuMode` and `OsTrusted` of OS applications.

As far as timing is concerned, an exclusive area impairs monitoring of execution time budgets. Therefore, you can no longer rely on this mechanism when your application is inside an exclusive area. It might even become unreliable outside exclusive areas when your application spends too much time inside them.

Note that the time stamps returned by `OS_GetTimeStamp()` aren't affected by exclusive areas.

Note also, you must not call any OS API functions from inside exclusive areas.

6. Reference manuals used

The following documents were used during the development of EB tresos AutoCore OS for TriCore:

- TriCore TC1.6P & TC1.6E User Manual (Volume 1) [TC_ARCH_161_VOL1].
- TriCore TC 1.6.2 core architecture manual (Volume 1) [TC_ARCH_162_VOL1].
- AURIX TC3xx Target Specification [TC3XX_TS_1_210].
- AURIX TC33x/TC32x User's Manual Appendix [TC33X_TC32X_APPX_UM_20].
- TC32x Target Data Sheet / Target Specification [TC32X_DS_10].
- TC33x/TC32x Data Sheet Addendum [TC33X_TC32X_DS_AD_14].
- AURIX TC38x Target Specification [TC38X_TS_V251].
- AURIX TC38x User's Manual Appendix [TC38X_UM_APPX_20].

Bibliography

[ASCOS_DOCU-MENTATION]	*EB tresos AutoCore OS documentation*: product release 6.1, 3.1_AutoCore_OS_-documentation.pdf
[TC_ARCH_161_-VOL1]	Infineon:*TriCore TC1.6P & TC1.6E User Manual (Volume 1)*, V1.0 D10, February 2012
[TC_ARCH_162_-VOL1]	Infineon:*TriCore TC 1.6.2 core architecture manual (Volume 1)*, V1.2.2, January 2020
[TC32X_DS_10]	Infineon:*TC32x Target Data Sheet / Target Specification*, V1.0, October 2020
[TC33X_TC32X_-APPX_UM_20]	Infineon:*AURIX TC33x/TC32x User's Manual Appendix*, V2.0, February 2021
[TC33X_TC32X_-DS_AD_14]	Infineon:*TC33x/TC32x Data Sheet Addendum*, V1.4, November 2020
[TC38X_TS_V251]	Infineon:*AURIX TC38x Target Specification*, V2.5.1, April 2018
[TC38X_UM_AP-PX_20]	Infineon:*AURIX TC38x User's Manual Appendix*, V2.0, February 2021
[TC3XX_TS_1_210]	Infineon:*AURIX TC3xx Target Specification*, V2.1.0, February 2017