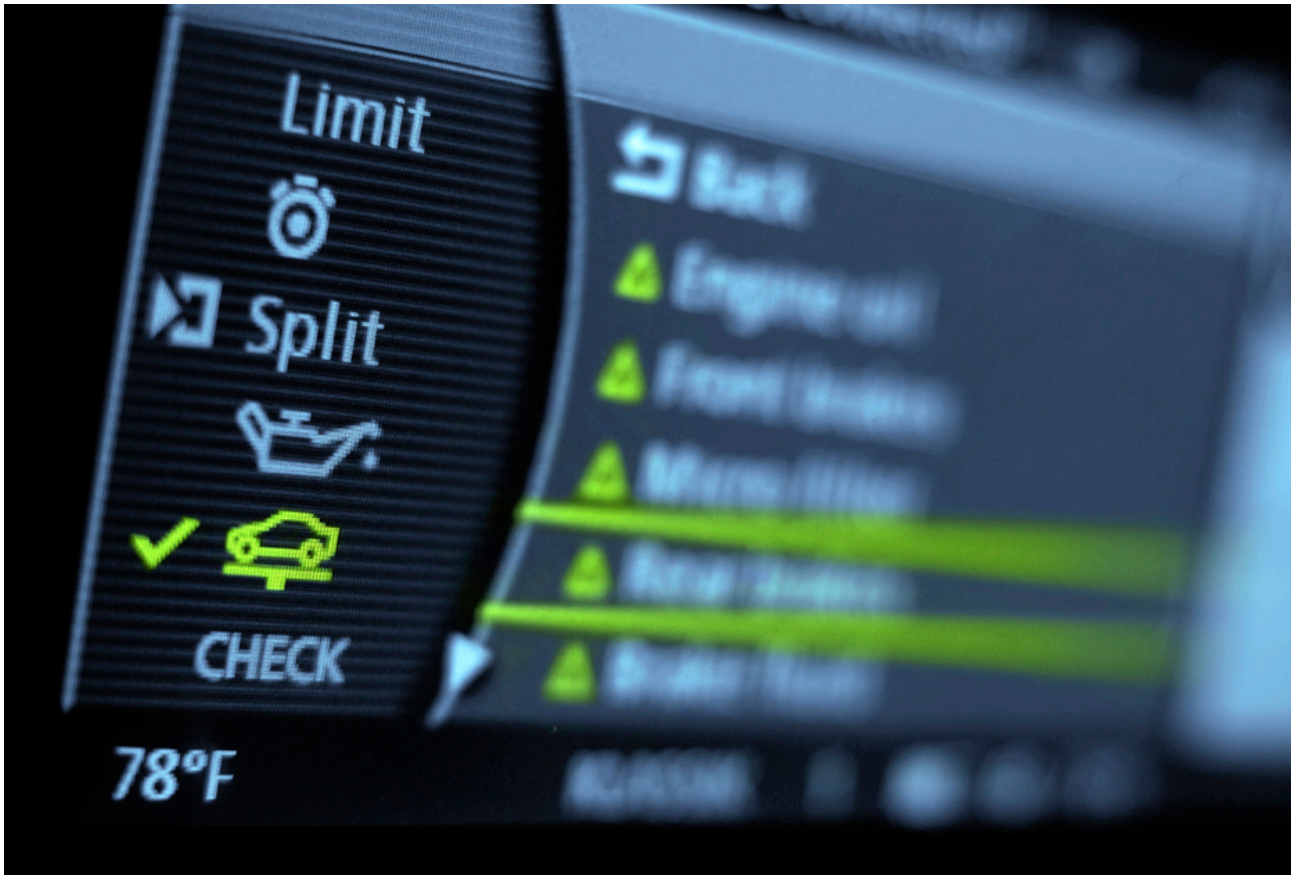
Elektrobit

EB tresos® AutoCore OS architecture notes TriCore

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

https://www.elektrobit.com/support

Legal disclaimer

Confidential and proprietary information

Table of Contents

1. Overview

1.1. Overview of the Architecture

TriCore has a 32-bit microprocessor core. 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. Some early versions have silicon bugs that must be worked around in software.

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. To achieve this it must be compiled with the correct compiler switches and macro definitions. It is therefore important to ensure that the precompiled kernel and user libraries are for the TriCore derivative that is in use.

1.2. Supported Toolchains

Currently EB tresos AutoCore OS for TriCore supports the TASKING, HighTec GCC and Wind River Diab compilers. Other toolchains may be supported in the future on request.

All the C and assembler files - whether generated or supplied as source - can be compiled with all supported toolchains using the makefiles provided. The assembler files must be preprocessed using the C preprocessor.

1.3. Glossary

Abbreviations and technical terms used in this document

ALU
: Arithmetic-Logic Unit. The part of the CPU where computations are performed.

CPU
: Central Processing Unit

CSA
: Context Save Area. TriCore uses a linked list of CSAs instead of a stack for call and return operations.

CSFR
: Core Special Function Register. One of a set of special registers in the TriCore CPU that controls the way the CPU operates or maintains its internal state. These registers can be accessed by use of special instructions.

EABI
: Embedded Application Binary Interface. A specification of the programming interfaces (for example, C calling conventions) that are used.

FPU
: Floating-Point Unit. A part of the CPU (or sometimes a separate unit) that performs floating point calculations.

ISR
: Interrupt Service Routine

OIL
: OSEK Implementation Language

SRN
: Service Request Node. A special register in a peripheral device that can be used to control interrupt sources.

STM
: System Timer Module. A 64-bit free-running counter provided on all supported TriCore variants. The timer starts from 0 at reset and should never overflow during the design life of the processor. The STM units have compare registers that can be programmed to generate interrupts when the timer reaches a particular value.

2. Implementation Details

2.1. TriCore Details

The following sections describe those aspects of the hardware features that are relevant to the implementation of the EB tresos AutoCore OS. You find further information in the related TriCore hardware manuals.

2.1.1. Byte Ordering

The TriCore family is little-endian.

2.1.2. 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.

Trusted code (trusted applications, category 1 ISRs and the kernel itself) 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 at startup to permit read and 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 parameter `OsAppCpuMode` to the desired value. By default, non-trusted applications run in processor mode USER0. This means that the processor prohibits access to the peripheral area, to the CSFRs and to certain instructions such as ENABLE and DISABLE. Protection register set 1 is used for non-trusted applications, and is programmed for the currently-running application thread (task, ISR or hook function) whenever such a thread is started or resumed. The protection registers are not considered part of the thread's register set, so if the thread succeeds in altering the protection registers, the alterations are not honored during preemptions and the original values will be restored afterwards.

2.1.3. The Status Registers

TriCore contains several status registers. Those that are relevant to EB tresos AutoCore OS are discussed here. Full descriptions of the registers can be found in the TriCore User's Manuals for the derivative that you are using.

PSW
: The Processor Status Word contains the memory protection set selector, the processor mode setting and the global register write enable flag.

SYSCON
: The System Configuration Register contains the protection enable flags that enables the use of the memory protection registers.

ICR
: The Interrupt Control Register contains the current CPU priority number (CCPN) and the interrupt enable flag (IE) and thus controls the interrupt level (arbitration priority) above which interrupts are enabled.

ISP
: The Interrupt Stack Pointer 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 and Base Trap Vector registers 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.

SRNs
: Most peripheral modules have Service Request Registers (also known as Service Request Nodes) that can be programmed to generate interrupt requests at arbitrary priorities.

All of the above registers are managed by the kernel. The user should not explicitly modify any of these registers.

2.1.4. 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.

2.1.5. The Stack Pointer

Address register A10 is used as the stack pointer. This register should not be explicitly modified by the user. 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.

2.1.6. 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 (class 6) and context-list underflow (class 3, TIN 5) exceptions are used for task and ISR control purposes in order 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 the 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.

2.1.7. Interrupt Handling

The EB tresos AutoCore OS kernel handles all interrupts. The generator creates an interrupt vector table whose entries call appropriate kernel entry and exit functions. This applies to category 1 interrupts as well as category

2 interrupts, although the entry and exit functions for category 1 interrupts are much smaller. The interrupt vector table configured by the generator must be used; compiler-generated vector tables are not supported.

EB tresos AutoCore OS supports both category 1 and category 2 interrupts. User-defined interrupts can be automatically enabled at startup by setting the `ENABLE_ON_STARTUP` attribute for the ISR to `TRUE`. Interrupts that are not enabled at startup can be enabled later by calling `OS_EnableInterruptSource()`. It is possible to enable an interrupt source by writing directly to the SRN, but care must be taken to preserve the priority number put there by EB tresos AutoCore OS.

For the kernel's initialization of the SRNs all peripheral modules that are in use must be turned on if necessary before `StartOS()` is called. Failure to do this results in undefined behavior; some TriCore processors generate exceptions when writing to disabled peripheral modules, but others simply ignore the attempt with the SRN remaining uninitialized.

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. The generator ensures that all category 1 interrupts have higher priority than category 2 interrupts. Additional interrupt levels are occupied for the kernel's own interrupts, if applicable. Examples are the interrupt used for cross-core notification between instances of the kernel on different processor cores, or the interrupt used to detect that an execution-time budget has been exceeded. These interrupts are set to levels higher than the category 2 interrupts, but below the category 1 interrupts. You must leave a gap between the levels chosen for your category 2 and category 1 interrupts in the configuration, that the OS generator can use to allocate levels to its internal interrupts.

The arbitration priorities for interrupts (the levels programmed into the service request registers) are allocated in ascending order of user-specified level, starting at 1. Interrupts with the same user-specified level will have different arbitration priorities because TriCore does not support multiple interrupts at the same level. The run priority of the ISRs (the CPU priority used with the BISR instruction in the vector code) 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.

2.1.7.1. Category 1 Interrupts

Category 1 interrupts are handled using a minimal kernel wrapper. This category is useful when no EB tresos AutoCore OS functions are needed in the interrupt routine, therefore saving kernel overhead.

Since the category 1 ISRs bypass the EB tresos AutoCore OS kernel, it is strictly forbidden to call kernel functions from these interrupts. Violations of this rule are detected and handled according to the Extended Status setting in the OS configuration.

Execution-time monitoring is not suspended during category 1 interrupts, so time spent in these interrupts is counted as part of the budget for the interrupted task or ISR. It is therefore not recommended to use category 1 interrupts in conjunction with execution-time monitoring.

2.1.7.2. 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 can be called.

All interrupt routines are written in the following format:

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

The function body is written just like a normal C function. The compiler's `interrupt` keyword must never be used, because the OS already takes care of the needed actions such as saving the full context.

For compatibility with ProOSEK, there is an `ISR1` macro with an identical definition to the `ISR` macro.

2.1.8. Nested Interrupts

Nested interrupts are supported by this architecture. Nesting of category 2 interrupts will occur according to the priority levels that are set up via the `TRICORE_IRQLEVEL` parameter. Category 1 interrupts will nest in a similar manner, and additionally will interrupt any category 2 or kernel interrupt. If an interrupt is pending with a higher priority than the interrupt which is currently being serviced then the interrupt being serviced will be preempted.

2.1.9. Implementation Details for Interrupt Services

On the occurrence of an interrupt the processor branches to the 8-word code fragment in the vector table corresponding to the interrupt level (arbitration priority) that occurred. The code for the vector table is automatically built by the kernel source files using the configuration produced by the EB tresos AutoCore OS generator. Compiler-generated interrupt vector tables are *not* used.

2.1.10. Resources on Interrupt Level

Category 2 ISRs can use resources. They are supported by raising the CPU's current interrupt level to the same level as the highest ISR that uses the resource.

2.1.11. Hardware Counters

This version of EB tresos AutoCore OS offers a driver for System Timer Module (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 a hardware counter is defined 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 (OS attribute `TRICORE_EXECUTION_TIMER`) can be either `STM[stm_id]_T0` or `STM[stm_id]_T1`. The timer selected for this purpose cannot be used with a hardware counter.

When arrival-rate monitoring is enabled, a hardware counter is used for timing the arrivals and for re-enabling interrupts that are disabled when their arrival rate is exceeded. The hardware counter is selected by means of the OS attribute `RATE_MONITOR_COUNTER`. The selected counter can be used for alarms and schedule tables in the normal way if desired.

2.2. 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 a reference 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 a reference to it is stored in PCXI. Returning from the subroutine or interrupt reverses this procedure.

The upper context registers include A10 (stack pointer) and A11 (return address), 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 in `boardAsm.s` 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.

The number of CSAs required can be configured for each core in the OS configuration. The generator will output this number in `Os_objects.make`, and the build files of the demo application will use this number to instruct the linker script generator to allocate memory for the given number of CSAs. Otherwise, the number of CSAs specified in the configuration has no effect. If the any of the tools in this process are not used, the CSA areas can be defined manually by settings the appropriate symbols for each used core `<n>`: The symbol

OS_csaCore<n>_BEGIN specifies the address of the first CSA in a contiguous block of CSAs. The number of CSAs inside this block is specified by the symbol OS_csaCore<n>_NCSA.

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

2.3. Compiler Issues

All supplied assembler files must be preprocessed with the C preprocessor before assembling. The default makefiles include rules to ensure that this happens.

Each qualified OS release is tested with a particular compiler, compiler version and compiler switches. These can be found in the quality statement delivered with the EB tresos AutoCore OS. Use of differing settings may work but is not tested.

To compile the EB tresos AutoCore OS, the following preprocessor macros need to be set in the compilation environment, typically by using appropriate compiler switches:

- OS_TOOL needs to be set according to the used toolchain, the possible values are:
  - OS_tasking for TASKING
  - OS_gnu for HighTec GCC
  - OS_diab for Wind River Diab
- OS_ARCH needs to be set to OS_TRICORE
- OS_CPU needs to be set to select the correct CPU derivative, for example OS_TC277

2.3.1. Floating Point

The EB tresos AutoCore OS kernel does not perform any floating point calculations.

The optional TriCore FPU uses the same register set as the integer ALU, therefore the kernel does not need to provide a floating-point context switch.

On derivatives without FPU it is possible to use the software floating point library supplied by the toolchain. Software floating point libraries must be reentrant to ensure that a context switch does not destroy any floating point storage area. Check the floating point library documentation for reentrancy support or use software methods - for example OSEK resources - to make sure that no data structures are corrupted when used by different tasks simultaneously.

2.4. Implementation Specific Behavior

The startup code for EB tresos AutoCore OS is provided in source-code form for common TriCore derivatives and evaluation boards. Using this as a starting point it should be possible to construct startup code for your own board.

The startup code provided with EB tresos AutoCore OS does not rely on the compiler's startup code for correct operation. Attempting to use portions of the compiler's startup code in conjunction with the supplied code will probably fail with linker errors. It should be possible, however, to write completely new startup code that uses the compiler's startup modules. However, it should be emphasized that this approach has not been tested, and may have implications for the certification of the system.

2.4.1. Start-up Order

This section describes in general terms the startup sequence used by the board-support packages supplied with EB tresos AutoCore OS.

The TriCore CPU usually starts execution at address 0xa0000020 with the startup configuration contained in the demo code. The assembly-language startup code at boot disables interrupts, sets an initial value in the PSW, invokes the special C routines `OS_InitSp()` and `OS_InitCsaList()` to initialize the stack pointer and free CSA list. Until this is done it is not possible to call a normal C function. Finally, the code jumps to `BoardStart()`.

The function `BoardStart()` in `board.c` is a normal C function, with the restriction that it must not assume any initialization of statically-allocated variables in the `.data` or `.bss` sections. This routine is responsible for setting the CPU clock frequency and several of the core special-function registers. In particular, the ISP (interrupt stack pointer) register must be set to its correct working value and the BTV (base trap vector) register must point to the special initialization trap vectors. The routine also initializes the `.data` section from its ROM image and clears the `.bss` section. Note that the private `.data` and `.bss` areas of OS-Applications are not initialized here. Global address register `A8` is initialized, so it points to the kernel data of the core which executes `BoardStart()`. Finally, this function calls `main()`.

The `main()` function must be provided by the user. Here 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. Failure 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 the EB tresos AutoCore OS need to be started using the `StartCore()` system service and will in turn start executing the `main()` function. Finally the system service `StartOS()` is called with the desired application mode as parameter.

The `StartOS()` service starts the EB tresos AutoCore OS kernel and under normal circumstances never returns. The BIV and BTV registers are set to their correct operational values, auto-start tasks and alarms are started, hardware timers and interrupt service request registers are initialized, application `.data` and `.bss`

areas are initialized and all stacks are filled with the fill value (normally 0xeb). Finally the StartupHook functions (if configured) are called. When `StartOS()` finishes, the highest priority active task runs. If no task is active the idle loop waits with interrupts enabled until a task becomes active.

2.4.2. Initialization

The following table indicates which code is responsible for initialization of the important TriCore hardware devices.

Device	**Initialized by**
Free CSA List	`boardAsm.s`
PLL	`BoardStart()` (`board.c`)
ENDINIT	`BoardStart()` (`board.c`)
BTV	`BoardStart()` (`board.c`) and `StartOS()`
BIV	`StartOS()`
ISP	`BoardStart()` (`board.c`)
Global `.data`/`.bss`	`BoardStart()` (`board.c`)
Global address register `A8`	`StartCore()` sets this register on all cores which it starts. On core 0, which starts automatically, `BoardStart()` (`board.c`) sets `A8` to core 0's entry in `OS_kernel_ptr`.
Global address registers `A0` and `A1`	If small data areas are used (depends on compiler), these registers are initialized by `BoardStart()` (`board.c`) and `boardAsm.s`.
Timer unit's clock control register	`main() (user-provided)`
Kernel stack	`StartOS()`
Application `.data`/`.bss`	`StartOS()`
Task stacks	`StartOS()`
Kernel data structures	`StartOS()`
Interrupt sources	`StartOS()` (only those with `ENABLE_ON_STARTUP = TRUE`)
Timer unit's counters etc.	`StartOS()`
Memory protection unit	`StartOS()`

Table 2.1. Initialization of TriCore hardware

Any unit not listed is not directly relevant to the kernel and is therefore not initialized. However, if the interrupt sources in a peripheral module are used the peripheral module's clock control register must be initialized before

calling `StartOS`. The initialization cannot be performed in the `StartupHook` because the kernel initializes the interrupt sources *before* calling the `StartupHook`.

2.4.3. Shutdown Behavior

The system can be stopped by calling the `ShutdownOS()` service provided in EB tresos AutoCore OS. When called, the service will lock all interrupts and call all the configured `ShutdownHook()` functions. After the global `ShutdownHook()` returns, the service will stay in an endless loop. If the user wishes to change this behavior it is possible to do so in the global `ShutdownHook()`. The `ShutdownHook()` does not have to return to the `ShutdownOS()` service.

2.4.4. Error Reporting

When an `ErrorHook()` or `ProtectionHook()` is called as a result of a trap, the 3 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.

2.4.5. The Extended Runtime Checker

No TriCore-specific checks are implemented.

3. Extended OIL Attributes

The EB tresos AutoCore OS generator uses the TriCore-specific OIL attributes listed in the following table. The attributes are described fully in the subsequent sections.

Object	Attribute	Type
OS	MICROCONTROLLER	ENUM
OS, OS_CONTAINER CORE_CONFIG	CORE_ID	UINT32
OS, OS_CONTAINER CORE_CONFIG	TRICORE_EXE-CUTION_TIMER	ENUM
OS, OS_CONTAINER CORE_CONFIG	TRICORE_NUM_CSA	UINT32
ISR	TRICORE_VECTOR	ENUM
ISR	TRICORE_IRQLEVEL	UINT32
ISR	STACKSIZE	UINT32
COUNTER	TRICORE_TIMER	ENUM
COUNTER	TRICORE_IRQLEVEL	UINT32

Table 3.1. TriCore OIL attributes

3.1. OS Attributes

The OS object has the following TriCore-specific attributes:

MICROCONTROLLER
: This is where the specific derivative of the TriCore family to be used is selected. Here all derivatives that are currently supported by the EB tresos AutoCore OS kernel can be found.

OS_CONTAINER CORE_CONFIG
: This complex attribute contains configuration parameters that apply to the kernel instance running on the processor core given by the `CORE_ID` attribute inside the container. The container is defined once for each core used by the EB tresos AutoCore OS. The container contains the following TriCore-specific attributes:

  TRICORE_EXECUTION_TIMER
  : This is where the hardware timer to be used for execution-time protection is selected.

  TRICORE_NUM_CSA
  : This attribute specifies the number of CSAs available on the respective core.

3.2. ISR Attributes

ISR objects have the following TriCore-specific attributes:

TRICORE_VECTOR
: This attribute specifies the TriCore service request register to which the ISR will be attached.

TRICORE_IRQLEVEL
: This attribute specifies the interrupt priority relative to all other ISRs of the same category in the system. The level can be any integer greater than zero; the generator gathers all ISRs in each category together in order of `TRICORE_IRQLEVEL`, then assigns real arbitration and run priorities in ascending sequence, with category 1 ISRs always having higher priority than category 2 ISRs. This ensures that the minimum number of interrupt levels is used, thus minimizing the size of the vector table.

STACKSIZE
: This attribute specifies how many bytes of stack are required by the interrupt service routine.

3.3. COUNTER Attributes

COUNTER objects have the following TriCore-specific attributes:

TRICORE_TIMER
: This attribute specifies the TriCore hardware timer that the counter will use. At the time of writing, the supported timers are STM comparators 0 and 1 on derivatives where the STM has this feature. Other timers are not supported. A value of USER_COUNTER specifies that the counter is a user-counter which is incremented by the application software itself.

TRICORE_IRQLEVEL
: This attribute specifies the interrupt priority relative to all other category 2 ISRs in the system. The level can be any integer greater than zero; the generator gathers all ISRs in each category together in order of TRICORE_IRQLEVEL, then assigns real arbitration and run priorities in ascending sequence. This ensures that the minimum number of interrupt levels is used, thus minimizing the size of the vector table.

4. Linker Script Generation

Linker scripts for EB tresos AutoCore OS are complicated by the need to segregate the `.data` and `.bss` sections of applications from the kernel and from each other. To this end, EB tresos AutoCore OS provides a tool to assist in generating linker scripts. In many cases the output of the tool can be used directly, but can be modified if necessary. The tool is provided in source-code form as a Perl program and so can be customized to suit your local needs. The linker script tool is called `genld-TRICORE.pl`.

The script is called from the command line. The script provides usage information when called without parameters. Some information provided to the linker script generator depends on the configuration and is provided by the OS generator in the generated `Os_objects.make`, such as the information which core each OS-Application is assigned to, which tasks belong to each OS-Application, and similar. The build files shipped with the demo application show how the information provided by the OS generator is transformed to command line options of the linker script generator.

5. Board Support

5.1. Directory Structure

The sample board directories are located under the demo's directory in a subdirectory called `boards`. A typical board name is `TriboardTC2X7` and the directory typically contains the following files (in the example, the CPU derivative is `TC29XT` and the board name is `TriboardTC2X7`):

- `board.c`
- `board.h`
- `boardAsm.s`
- `board-diab.ldscript`
- `board-gnu.ldscript`
- `board-tasking.ldscript`
- `TC29XT-tasking.ldscript`
- `Triboard.mak`
- `TriboardTC2X7.mak`

There may also be scripts to load programs into the debugger.

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 such as clock frequencies. The header typically contains configuration examples for several frequency settings to choose from for the supplied clock configuration code. You can extend the provided settings with your own, or use your own clock configuration code.

The EB tresos AutoCore OS needs to convert between time units and timer ticks to support specification of times given in time units (for example, nanoseconds) in the OS configuration. Since the OS does not know the clock configuration performed by the startup code, these conversion macros need to be provided to the OS by definition in `board.h`.

These macros are:

`OS_BoardNsToTicks(nS)`
: This macro provides a way to convert nanoseconds (nS) to ticks of the STM module clock. It must compute entirely in the preprocessor because it is used in constant initializers. It must perform the conversion

without overflow and with minimal rounding errors. The OS provides example implementations for various frequencies in the header `Os_timeconversion.h`, for example `OS_NsToTicks_90000000(nS)` for an STM frequency of 90 MHz.

`OS_BoardTicks2Ns(tik)`
: This macro provides a way to convert ticks (tik) of the STM module clock to nanoseconds. There are example implementations for various frequencies in the header `Os_timeconversion.h`, for example `OS_TicksToNs_90000000(tik)` for an STM frequency of 90 MHz.

`OS_INITIAL_PSW`
: This macro defines the initial value of the processor status word (PSW). It is used by the demo startup code for the initialization core, and by the OS for all cores started using `StartCore()`.

`OS_STM_PRESCALE`
: This macro defines the prescaler in the STM module applied to the STM module clock. If no prescaler is available, this must be set to 1.

In addition, `board.h` defines the prescaler settings and includes the timer configuration file (for example: `Os_TRICORE_stmconfig.h`) to define the TicksToNs and NsToTicks macros for the STM timers.

5.3. Source File: `boardAsm.s`

The assembly-language source file `boardAsm.s` contains the small assembler program that is jumped to on reset. The program initializes the CPU, constructing a limited environment in which a restricted C function can be called. On completion, the C function `BoardStart()` is jumped to.

5.4. Source File: `board.c`

The C source file `board.c` contains the function `BoardStart()`. This function performs further initialization of the CPU and memory before calling the user-supplied `main()` function. When `main()` is called, the CPU is executing at the desired speed, supports a full C environment and can accept EB tresos AutoCore OS system calls.

5.5. Make Files: `*.mak`

The makefiles in the `board` directory make the source files of the demo startup code known to the build system and contain the rules to call the linker script generator with the configuration-specific information provided by the OS generator.

5.6. Linker Scripts: `*.ldscript`

The linker scripts contained in the `board` directory contain static information that is included by the linker script generator in the created linker script. There are several versions of the file in the syntax of the different supported toolchains. The scripts define the size and addresses of the different memories available on the used derivative, and static symbols such as the base address of the memory block used for the CSAs.

Appendix A. Generator Error Codes

TriCore-specific error codes are listed here. For architecture-independent error codes see the EB tresos AutoCore OS documentation.

1. Errors

Code	Description
OS_10001	*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.

1. Information

Code	Description
OS_10000	*Parameter OsTricoreNumCSAs not configured for core {0}, using {1}.* Each core in a TRICORE system needs its own list of context save areas (CSAs). If the number of CSAs to use is not properly configured, the OS will use a default value instead.