



# **INSTRUCTION MANUAL FOR PORTING FROM THE TRICORE ARCHITECTURE TO THE ARM ARCHITECTURE**

Shanghai ZC Technology® Technical Service

# INSTRUCTION MANUAL FOR PORTING FROM THE TRICORE ARCHITECTURE TO THE ARM ARCHITECTURE

Shanghai ZC Technology® Technical Service

<b>1</b>	<b>OVERVIEW .....</b>	<b>3</b>
1.1	ABOUT THIS MANUAL .....	3
1.2	CPU ARCHITECTURE.....	3
1.2.1	Arm Architecture .....	3
1.2.2	TriCore Architecture .....	3
1.3	ARMv7-M ARCHITECTURE.....	4
1.4	TC1.6P ARCHITECTURE .....	5
<b>2</b>	<b>COMPARISON OF CORE ARCHITECTURE.....</b>	<b>6</b>
2.1	PROGRAMMER'S MODEL.....	6
2.1.1	Data Type.....	6
2.1.2	Byte Ordering and Alignment .....	7
2.1.3	Operating Modes.....	8
2.2	INSTRUCTION SET .....	9
2.3	GENERAL PURPOSE REGISTERS .....	9
2.4	EXCEPTIONS AND INTERRUPTS .....	11
2.4.1	Interrupt Priority Level.....	11
2.4.2	Interrupt and Exception Handling .....	12
2.4.3	Interrupt and Trap Vector Tables .....	14
2.5	MEMORY.....	15
2.5.1	Memory Address Space .....	15
2.5.2	Addressing Modes .....	17
2.5.3	Cache .....	18
2.6	FLOATING POINT UNIT .....	20
2.7	DEBUG.....	20
<b>3</b>	<b>COMPARISON OF FUNCTIONAL SAFETY DESIGN .....</b>	<b>21</b>
3.1	CORE SAFETY .....	21
3.2	MEMORY PROTECTION .....	22
<b>4</b>	<b>SOFTWARE DEVELOPMENT AND PORTING.....</b>	<b>23</b>
4.1	DEVELOPMENT TOOLCHAIN .....	23
4.2	CHIP STARTUP.....	24
4.3	EXCEPTION AND INTERRUPT HANDLING .....	25
4.4	PERIPHERAL ACCESS .....	26
<b>5</b>	<b>AUTOSAR ARCHITECTURE PORTING.....</b>	<b>27</b>
5.1	AUTOSAR ARCHITECTURE .....	27

5.2	MCAL PORTING.....	27
5.3	AUTOSAR OPERATING SYSTEM PORTING .....	30
<b>6</b>	<b>SUMMARY .....</b>	<b>32</b>
<b>7</b>	<b>APPENDICES.....</b>	<b>32</b>
7.1	APPENDIX 1 REFERENCE MATERIALS.....	32
7.2	APPENDIX 2 TERMINOLOGY AND ABBREVIATIONS .....	34

# 1 OVERVIEW

## 1.1 About this Manual

The automotive industry is becoming smarter and more electric, and as a result MCU controller applications in automotive electronics systems are also becoming more ubiquitous. MCUs based on Arm architectures are now widely used in automotive controllers. At the same time, MCUs based on the TriCore architecture also account for a significant percentage of the automotive controller market. Considerations such as demand, cost, development efficiency, and risk management have resulted in multi-architectural development and applications being on the rise in many practical use cases. Furthermore, increased chip performance has also come with more complex core architectures, upgraded instruction sets, and more numerous multi-core architecture applications. In other words, chips are becoming more complex, and this has resulted in more requirements during workload migration between different CPU architectures. The purpose of this manual is to compare and contrast the TriCore and Arm architectures while analyzing, from a software porting perspective, the elements that should be considered when porting software developed for the TriCore architecture to the Arm architecture.

This manual discusses porting from the TriCore architecture to the Arm architecture from two main aspects:

- The first aspect is the differences between the TriCore and Arm architectures, which are explored primarily from the perspectives of CPU architecture and functional safety.
- The second aspect is software development and porting, describing in detail the methods for porting software development for the TriCore architecture to the Arm architecture, as well as analyzing the porting process based on the AUTOSAR architecture.

## 1.2 CPU Architecture

### 1.2.1 Arm Architecture

In the automotive MCU controller field, the Arm architecture is a 32-bit CPU architecture. The Arm architecture is widely used in embedded system designs. It has low power consumption and is suitable for mobile communications and consumer electronics, such as mobile phones, multimedia players, handheld gaming devices, computers, and computer peripherals. It is also applicable to the industrial, automotive, and aerospace fields.

Arm CPUs cores are divided into three series: Cortex-A, Cortex-R, and Cortex-M.

- Cortex-A series

The Cortex-A series of application processors includes the Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A5x, Cortex-A7x, and Cortex-A71x processors, which are based on the ARMv7-A, ARMv8-A, and ARMv9-A architectures. This series provides solutions for devices running complex operating systems (such as Linux, Android, and iOS). The Cortex-A series is widely applicable to a variety of use cases, ranging from low-cost handheld devices to smartphones, tablets, set-top boxes, and enterprise network equipment. It is capable of processing massive amounts of data and high-performance computing. This type of processor generally runs at very high clock speeds (generally over 1GHz). It supports memory management units (MMUs) required by Linux, Android, Windows, and mobile operating systems.

- Cortex-R series

The Cortex-R series of real-time processors includes the Cortex-R4, Cortex-R5, Cortex-R7, Cortex-R8, and Cortex-R52 processors, which are based on the ARMv7-R and ARMv8-R

architectures. As real-time microcontroller cores, the Cortex-R series is specifically designed for embedded systems that require high safety and performance. It provides fast and deterministic response times, making it an ideal choice for applications that have high requirements for real-time responses and safety, such as automotive, industrial, and aerospace systems. Although real-time processors cannot run complete versions of the Linux and Windows operating systems (apart from the Cortex-R82), they are capable of supporting large number of real-time operating systems (RTOS).

### ➤ Cortex-M series

The Cortex-M series of microcontroller processors include the Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, Cortex-M33, Cortex-M52, Cortex-M55 and Cortex-M85 processors, which are based on the ARMv6-M, ARMv7-M, ARMv8-M, and ARMv8.1-M architectures. The Cortex-M series processors feature cores that have low power consumption, high performance, and scalability, and include numerous features that make them suitable for embedded systems. They are designed to be easy to use, which is why they have been highly successful in the microcontroller, IoT, and embedded systems markets. The Cortex-M series is widely used in applications ranging from consumer electronics to industrial control systems, including the microcontroller market, IoT, embedded systems, and automotive controllers.

This manual uses the ARMv7-M architecture as the target architecture for porting. The ARMv7-M architecture includes the Cortex-M3, Cortex-M4, and Cortex-M7 processor architectures. Unless otherwise specified, the content of this manual will use the ARMv7-M architecture for reference.

### 1.2.2 TriCore Architecture

The TriCore architecture is a core architecture for the AURIX family of architectures released by Infineon. The AURIX TriCore architecture unites a RISC processor core, a microcontroller, and a digital signal processor (DSP) in a single MCU. Controller products based on the TriCore architecture are widely used in the automotive industry, including in applications such as powertrains, body control, safety applications, and advanced driver assistance systems (ADAS). They are driving the automotive industry toward greater automation, electrification, and connectivity. Currently, Infineon has launched AURIX 1G and AURIX 2G products, which are the TC2xx series and TC3xx series, respectively.

### ➤ AURIX 1G TC2xx series

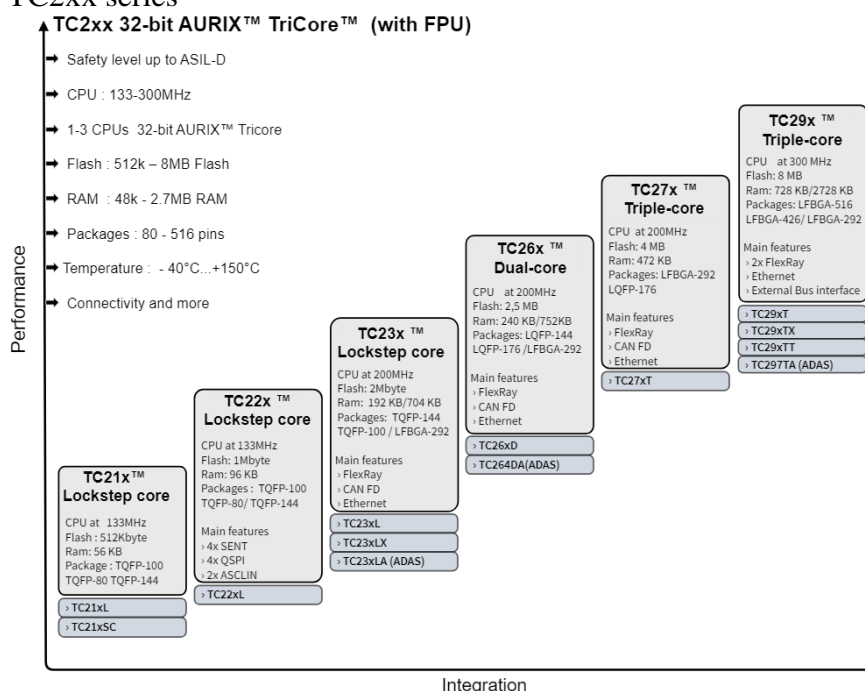
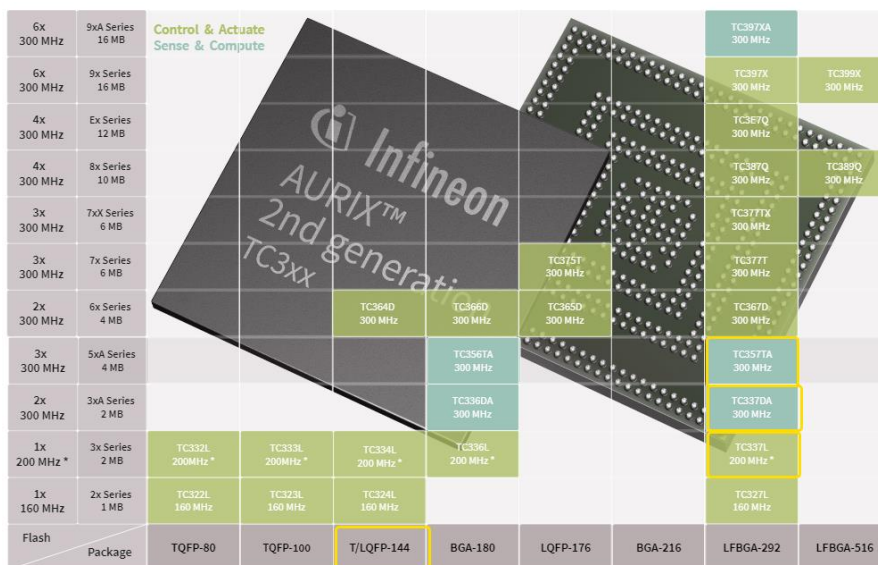


Figure 1.2.2-1 TC2xx chip series

The AURIX™ TC2XX series processors are based on single-core and multi-core 32-bit TriCore™ architecture. It can meet the highest safety requirements while also providing high-level performance. The AURIX™ architecture is used in automotive powertrains, including electric and hybrid vehicles, as well as safety systems such as steering, braking, airbags, and ADAS. The core architecture of the AURIX™ TC2xx series uses the TriCore TC1.6P and TC1.6E architectures. The TriCore TC1.6P core features a high-performance system architecture, capable of reaching a maximum clock speed of 300MHz. The TC1.6E core features a system architecture that has high performance and low power consumption. It can reach a maximum clock speed of 200MHz.

### ➤ AURIX 2G TC3xx series



6x 300 MHz	9xA Series 16 MB	Control & Actuate Sense & Compute				TC397XA 300 MHz	
6x 300 MHz	9x Series 16 MB					TC397X 300 MHz	TC399X 300 MHz
4x 300 MHz	Ex Series 12 MB					TC367Q 300 MHz	
4x 300 MHz	8x Series 10 MB					TC387Q 300 MHz	TC389Q 300 MHz
3x 300 MHz	7xX Series 6 MB					TC377TX 300 MHz	
3x 300 MHz	7x Series 6 MB					TC377T 300 MHz	
2x 300 MHz	6x Series 4 MB					TC367D 300 MHz	
3x 300 MHz	5xA Series 4 MB					TC357TA 300 MHz	
2x 300 MHz	3xA Series 2 MB					TC337DA 300 MHz	
1x 200 MHz *	3x Series 2 MB	TC333L 200 MHz *	TC333L 200 MHz *	TC334L 200 MHz *	TC336L 200 MHz *	TC337L 200 MHz *	
1x 160 MHz	2x Series 1 MB	TC322L 160 MHz	TC323L 160 MHz	TC324L 160 MHz		TC327L 160 MHz	
Flash	Package	TQFP-80	TQFP-100	T/LQFP-144	BGA-180	LQFP-176	BGA-216
							LFBGA-292
							LFBGA-516

Figure 1.2.2-2 TC3xx chip series

The architectures of the AURIX™-TC3xx series feature both high performance and high safety. It can have up to six cores and is suitable for next generation applications that fuse self-driving domain controllers and data. AURIX™ TC3xx microcontrollers are applicable to safety-related use cases, such as airbags, braking and power steering systems, radar, lidar, and camera sensors and systems. The AURIX™ TC3xx uses the TriCore TC1.6.2P core architecture. The TriCore TC1.6.2P architecture is similar to the TC1.6P architecture, but features better memory access performance and memory protection.

This manual uses the AURIX TriCore TC1.6P architecture as the porting architecture. The AURIX TriCore TC1.6P architecture supports the TC2xx and TC3xx series. Unless otherwise specified, the content of this manual will use the TriCore TC1.6P architecture for reference.

## 1.3 ARMv7-M Architecture

The Cortex-M processors include the ARMv6-M, ARMv7-M, ARMv8-M and ARMv8.1-M architectures. The Cortex-M0 and Cortex-M0+ are based on the ARMv6-M architecture, the Cortex-M3, M4, and M7 are based on the ARMv7-M architecture, and the Cortex-M23, Cortex-M33, Cortex-M52, Cortex-M55 and Cortex-M85 are based on the ARMv8-M architecture. The ARMv7-M architecture shares many similarities with previous Arm architectures, and it has been specially designed to support deeply embedded and lower cost real-time microcontrollers. By removing many features of the old architectures while adding new ones, a program design model that is more similar to a microcontroller has been created. For example, the number of operating modes has been greatly reduced from seven or more to two: Handler mode and Thread mode. Furthermore, the ARMv7-M Porting from the TriCore Architecture to the Arm Architecture

architecture only supports Thumb instruction sets and features a brand-new system-level programming model.

The key features of the ARMv7-M architecture are as follows:

- Power, performance, and footprint constraints that are the most stringent in the industry  
Straightforward pipeline design delivers industry-leading system performance in a wide range of markets and applications
- Highly deterministic operations  
Supports single or low cycle count execution  
Minimal interrupt latency and shorter pipeline design  
Can perform cacheless operations
- Excellent support for C/C++ applications and maintains consistency with Arm's programming standards in this field  
Exception handling routines are standard C/C++ functions that use standard calling conventions
- Designed specifically for deep embedded systems  
Supports devices with lower pin counts
- Supports debugging and software analysis, as well as event-driven systems

#### 1.4 TC1.6P Architecture

The TriCore architecture is the first unified single-core 32-bit microcontroller DSP architecture that is optimized for real-time systems. The TriCore instruction set architecture (ISA) combines the real-time capabilities of microcontrollers, the computational power of DSPs, and the cost-effectiveness of RISC's load-store architecture into a compact and programmable core.

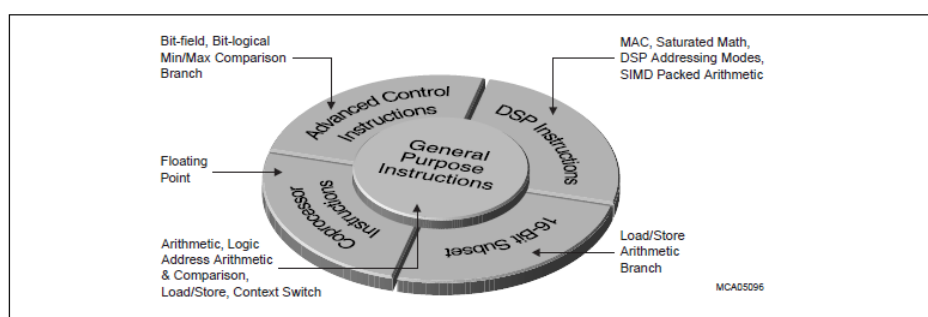


Figure 1.4-1 TriCore Architecture

The ISA supports a unified 32-bit address space, optional virtual addressing, and memory-mapped input/output. The ISA simultaneously supports 16-bit and 32-bit instruction formats. All instructions have a 32-bit format, of which 16-bit instructions are a subset. 16-bit instructions are selected because of their usage frequency. These instructions greatly reduce code space, thereby lowering memory requirements and system and power consumption.

Real-time responsiveness is mainly dependent on interrupt latency and context switching time. High-performance architectures minimize interrupt latency by avoiding long multi-cycle instructions and providing flexible hardware support for interrupts. The TriCore architecture also supports fast context switching.

## 2 COMPARISON OF CORE ARCHITECTURE

This manual analyzes the differences between the Arm and TriCore architectures in the following aspects to more clearly establish the differences:

- **Programmer's model**  
The differences in the programmer's models for the two architectures are explored by introducing the features of the chips from a developer's perspective. The analysis is mainly based on their data type formats, modes of operation, and other aspects.
- **Instruction set**  
The differences in the instruction sets of the two architectures are explored.
- **General Purpose Registers**  
The differences in the general-purpose registers (GPRs) of the two architectures are explored.
- **Exceptions and interrupts**  
The interrupts and exceptions used in the two architectures are explored, including their interrupt priorities, interrupt and exception handling methods, and the differences between their interrupt and exception vector tables.
- **Memory**  
The memory models of the two architectures are explored, including address space, addressing method, caches, etc.
- **Floating Point Unit**  
Both TriCore and Arm support floating point computing, and the differences in floating point computing between the two are explored.
- **Debug**  
The differences between the debug systems of the two architectures are explored.

### 2.1 Programmer's Model

The programmer's model is the interface that developers use during chip development. Developers must ensure they are familiar with the programmer's model to increase development efficiency. This chapter introduces the differences between the two architectures through the data types supported by the chips, the byte orders, and the modes of operation.

#### 2.1.1 Data Type

The data types supported by the TriCore architecture include: Boolean, bit string, byte, signed fraction, address, signed and unsigned integers, IEEE-754 single-precision floating-point number.

The data types supported by the Arm architecture include: byte, halfword, word, 32-bit pointers, unsigned or signed 32-bit integers, unsigned 16-bit or 8-bit integers, signed 16-bit or 8-bit integers, unsigned or signed 64-bit integers.

Developers using the C language must pay close attention to the data types of the compiler, as the compiler is responsible for translating the defined data types into the target file data types supported by the hardware. This manual takes the commonly-used HighTec compiler for TriCore and the commonly-used Arm Compiler for Arm as examples to perform a comparative analysis.



Type	Hightec (Tricore)	Arm Compiler (Arm)
char	8bit	8bit
short	16bit	16bit
int	32bit	32bit
long	32bit	32bit
long long	64bit	64bit
float	32bit	32bit
double	64bit	64bit
long double	64bit	64bit
pointer	32bit	32bit
enum	8bit-32bit	8bit-32bit

Porting tip: Both compilers use generic data types, so variables defined as specific data types can be directly used during porting. If other compilers are used during porting, the methods described here can still be used for comparative analysis.

### 2.1.2 Byte Ordering and Alignment

The TriCore and Arm architectures both use 4-byte alignment (word aligned) for address alignment. The data alignment methods of the two architectures are generally based on the size of the data.

The data storage and CPU register data storage in the TriCore architecture both use the little-endian format (where the least significant byte of data is stored at the low memory address), as shown in the following figure:

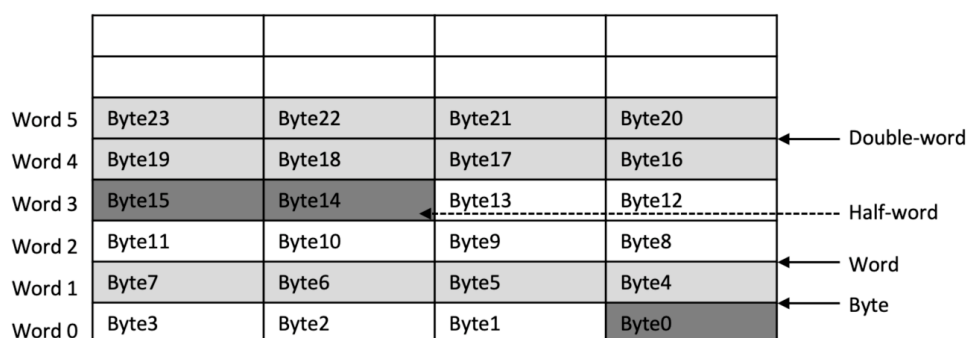


Figure 2.1.2-1 TriCore Byte Ordering

On the other hand, the Arm architecture can be configured to use either the little-endian format (where the least significant byte of data is stored at the lowest memory address) or the big-endian format (where the most significant byte of data is stored at the lowest memory address) for data storage.

#### ➤ Little-endian format of the Arm architecture

Words (4 bytes) and half-words (2 bytes) of data are stored in memory as follows:

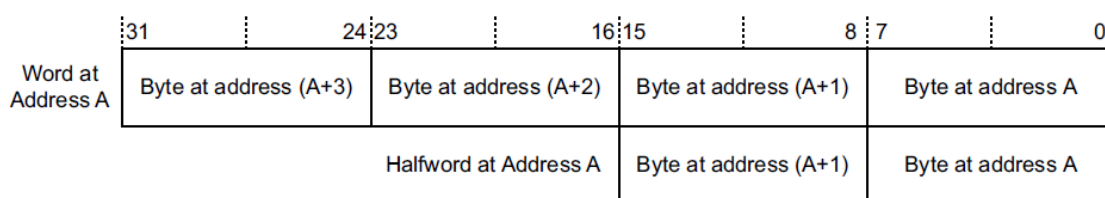


Figure 2.1.2-2 Little-endian format of the Arm architecture

➤ Big-endian format of the Arm architecture

Words (4 bytes) and half-words (2 bytes) of data are stored in memory as follows:

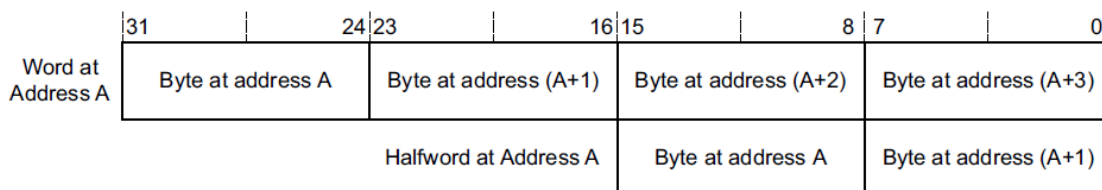


Figure 2.1.2-3 Big-endian format of the Arm architecture

Either big- and little-endian format can be selected for the byte order in the Arm architecture according to the control input during reset, the default is little-endian format.

Porting tip: Developers must consider the data defined in the program during the porting process. In particular, the byte order needs to be taken into account when using variables defined with custom structures. During porting, if the Arm architecture uses the big-endian format, any structures that involve byte order must be changed to the big-endian format.

### 2.1.3 Operating Modes

The I/O privilege levels of the TriCore architecture are divided into 3 levels: User-0 mode, User-1 mode, and Supervisor mode.

I/O Privilege Level	Description
User-0 mode	In this mode, tasks do not have access to external peripherals and cannot enable or disable interrupts.
User-1 mode	In this mode, tasks are used to access normal and unprotected peripherals, such as reading and writing to serial ports, accessing timers, and accessing the status registers of most I/O components.
Supervisor mode	In this mode, tasks can access the system register and peripherals, and can enable or disable interrupts.

The Arm architecture has two operating modes: Handler mode and Thread mode.

- Handler mode: Exception handling is performed in handler mode. In handler mode, the processor always has a privileged access level.
- Thread mode: Normal application code is executed in thread mode. In thread mode, the processor can be in a privileged access level or an unprivileged access level.

Operating Mode	Privilege	Use Cases
Handler	Privileged	Exception handling
Thread	Privileged	In this mode, when executing a privileged process or thread, only privileged access is supported.
	Unprivileged	In this mode, when executing an unprivileged process or thread, only unprivileged access is supported. Access to some resources will be restricted.

Porting tip: During the porting process, the developer needs to consider the differences in privilege levels for resources between the two architectures, especially when porting operating systems. It is necessary to consider the restrictions of privileged access. The User mode/Supervisor mode processing method for TriCore needs to be changed to Handler/Thread.

## 2.2 Instruction Set

The instruction sets of the TriCore and Arm architectures both support 16-bit and 32-bit reduced instruction sets.

The instruction sets of the TriCore architecture include: Arithmetic, address arithmetic, comparison, address comparison, logical, MAC, shift, coprocessor, bit logical, branch, bit field, load/store, packed data, system.

Most TriCore architecture instruction sets can be completely executed within one machine cycle.

The instruction sets of the Arm architecture are Thumb instruction sets based on Thumb-2, which are compatible with 16-bit and 32-bit instructions. Arm instruction sets include: Branch, Data-processing, Status register access, load/store, Miscellaneous, Exception-generating, Coprocessor, Floating-point.

A comparison of TriCore and Arm instruction sets is as follows:

TriCore architecture instruction sets	Arm architecture instruction sets
mov d3, d1	MOV R8, R7
add d3, d1, d2	ADD R1, R1, R3
j foobar	B label
CMPSWAP.W e0, [a0+4]	CMP R6, R7

Porting tip: Due to the significant differences between the two CPU architectures' instruction sets, developers may need to rewrite assembly code used in a program during the porting process based on actual functionality of the program.

## 2.3 General Purpose Registers

The core registers of the TriCore architecture are divided into 2 types: General Purpose Registers (GPRs) and Core Special Function Registers (CSFRs).

### ➤ General Purpose Registers (GPRs)

GPRs include 16 address general purpose registers A[0] to A[15] and 16 data general purpose registers D[0] to D[15].

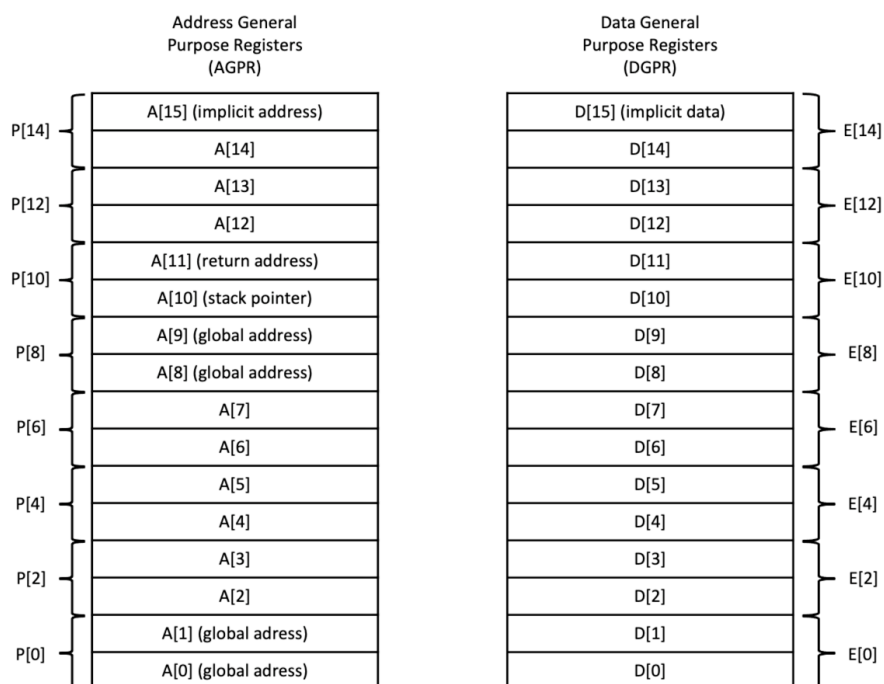


Figure 2.3-1 TriCore GPRs

The general-purpose registers A[10], A[11], A[15], and D[15] also have special functions:

A[10]: Stack Pointer (SP) register

A[11]: Return Address (RA) register

A[15]: Implicit Address register

D[15]: Implicit Data register

### ➤ Core Special Function Registers (CSFRs)

CSFRs include system registers, such as Program Counter (PC) registers, Program Status Word (PSW) registers, and Previous Context Information register (PCXI), which have key function in context switching for tasks.

CSFRs also include Compatibility Mode Register (COMPAT), Access Control Registers, Interrupt Registers, Memory Protection Registers, Trap Registers, Memory Configuration Registers, Core Debug Controller Registers, and Floating Point Registers.

The Arm architecture includes 16 registers, 13 of which are general purpose registers (R0 to R12), and 3 are special purpose registers.

#### ➤ General purpose registers

R0 to R12 are general purpose registers. R0 to R7 are known as low registers. Due to space limitations, many 16-bit instructions can only access low registers. R8 to R12 are high registers and can be used for 32-bit and some 16-bit instructions.

#### ➤ Special purpose registers

R13 is a Stack Pointer (SP) register, used as a pointer to stacks.

Porting from the TriCore Architecture to the Arm Architecture

R14 is a Link Register (LR), used to save the return link when functions or subroutines are called.  
R15 is a Program Counter (PC).

➤ Other special purpose registers

Application Program Status Register (ASPR) and exception and interrupt registers: PRIMASK, FAULTMASK, BASEPRI, control registers, floating point registers, etc.

Porting tip: During the porting process, developers must perform configuration according to the differences between the two architectures' core registers. In the TriCore architecture, when exceptions are encountered, analysis must be performed based on the D[15] register, which stores the Trap Identification Number (TIN), and the A[11] register, which records the entry address of the trap. On the other hand, in the Arm architecture, It needs to be analyzed based on the exception, interrupt-related registers PRIMASK, FAULTMASK, and BASEPRI. For the Stack Pointer (SP) register in the TriCore architecture, the A[10] register should be read to confirm the stack location. In the Arm architecture, the R13 register should be read to confirm the stack location.

## 2.4 Exceptions and Interrupts

The interrupt system of the TriCore architecture supports multiple interrupt sources, including chip peripheral interrupt and external interrupt. An interrupt request can either be serviced by the CPU or the DMA. Each interrupt source is assigned a unique interrupt priority level. Traps in the TriCore structure occur as a result of events such as non-maskable interrupts (NMI), instruction exceptions, memory management exceptions, or illegal access. Traps are always active and cannot be masked by the software.

The Arm architecture provides the Nested Vectored Interrupt Controller (NVIC) module for handling interrupts. The NVIC supports multiple interrupt requests, non-maskable interrupts (NMI), SysTick timer interrupts, and multiple system exceptions.

This manual describes the differences between the two architectures through interrupt priority level, interrupt and exception handling, and interrupt and exception vector tables.

### 2.4.1 Interrupt Priority Level

Interrupt requests in the TriCore architecture are handled according to the priority level, and interrupt nesting is supported. The rules for interrupt priority are as follows:

- High priority interrupts can interrupt low priority interrupts already running.
- Interrupts cannot be interrupted by another interrupt of the same priority level.
- The Interrupt Control Unit (ICU) determines which interrupt to handle according to the priority level.

All service requests are assigned a Service Request Priority Number (SRPN). Each interrupt handling process has its own SRPN. Different interrupt service requests must be assigned different SRPNs. There are up to 255 interrupt priority levels, with the priority level number 0 being the lowest interrupt priority. Exceptions with the highest priority level cannot be masked by the software.

The Arm architecture exception and interrupt handling is determined by the priority of the exception and the current priority of the processor. Interrupt nesting is supported. High priority interrupts can interrupt low priority interrupts already running. When two exceptions with the same

Porting from the TriCore Architecture to the Arm Architecture

priority level are triggered, the exception with the lower interrupt ID number will be handled first. Some exceptions (RESET, NMI, and HardFault) have fixed priority levels. Their priority levels are negative, so they will have higher priority than other exceptions. The priority levels of other exceptions can be configured between a range of 0 to 255. The smaller the value, the higher the interrupt priority level.

### 2.4.2 Interrupt and Exception Handling

The TriCore architecture defines 8 types of traps. Each type of trap is identified by a Trap Identification Number (TIN). Before entering the trap service process, the TIN value is assigned to the D[15] register. Traps are divided into synchronization traps, asynchronous traps, hardware traps, software traps, and unrecoverable traps.

TIN	Name	Synch. / Asynch.	HW / SW	Definition
<b>Class 0 - MMU</b>				
0	VAE	Synch.	HW	Virtual Address Fill.
1	VAP	Synch.	HW	Virtual Address Protection.
<b>Class 1 – Internal Protection Traps</b>				
1	PRIV	Synch.	HW	Privileged Instruction.
2	MPR	Synch.	HW	Memory Protection Read.
3	MPW	Synch.	HW	Memory Protection Write.
4	MPX	Synch.	HW	Memory Protection Execution.
5	MPP	Synch.	HW	Memory Protection Peripheral Access.
6	MPN	Synch.	HW	Memory Protection Null Address.
7	GRWP	Synch.	HW	Global Register Write Protection.
<b>Class 2 – Instruction Errors</b>				
1	IOPC	Synch.	HW	Illegal Opcode.
2	UOPC	Synch.	HW	Unimplemented Opcode.
3	OPD	Synch.	HW	Invalid Operand specification.
4	ALN	Synch.	HW	Data Address Alignment.
5	MEM	Synch.	HW	Invalid Local Memory Address.
<b>Class 3 – Context Management</b>				
1	FCD	Synch.	HW	Free Context List Depletion (FCX = LCX).
2	CDO	Synch.	HW	Call Depth Overflow.
3	CDU	Synch.	HW	Call Depth Underflow.
4	FCU	Synch.	HW	Free Context List Underflow (FCX = 0).
5	CSU	Synch.	HW	Call Stack Underflow (PCX = 0).
6	CTYP	Synch.	HW	Context Type (PCXI. UL wrong).
7	NEST	Synch.	HW	Nesting Error: RFE with non-zero call depth.

TIN	Name	Synch. / Asynch.	HW / SW	Definition
<b>Class 4 – System Bus and Peripheral Errors</b>				
1	PSE	Synch.	HW	Program Fetch Synchronous Error.
2	DSE	Synch.	HW	Data Access Synchronous Error.
3	DAE	ASynch.	HW	Data Access Asynchronous Error.
4	CAE	ASynch.	HW	Coprocessor Trap Asynchronous Error.
5	PIE	Synch.	HW	Program Memory Integrity Error.
6	DIE	ASynch.	HW	Data Memory Integrity Error.
7	TAE	ASynch.	HW	Temporal Asynchronous Error.
<b>Class 5 – Assertion Traps</b>				
1	OVF	Synch.	SW	Arithmetic Overflow.
2	SOVF	Synch.	SW	Sticky Arithmetic Overflow.
<b>Class 6 – System Call</b>				
4	SYS	Synch.	SW	System Call.
<b>Class 7 – Non-Maskable Interrupt</b>				
0	NMI	ASynch.	HW	Non-Maskable Interrupt.

Figure 2.4.2-1 TriCore trap categories

Arm architecture exceptions are divided into the following:

- Reset: Reset, including power on and hot start
- NMI: Non-maskable interrupt
- Hard Fault: Can be triggered by all faults
- MemManage Fault: Memory management error, memory protection unit (MPU) error, or unauthorized access
- Bus Fault: Bus error, prefetch error, memory access error, or other address access error
- Usage Fault: Usage error, such as executing undefined instructions or illegal state transitions
- SVC: Supervisor call typically used for operating systems
- Debug Monitor: Debug monitoring, in which breakpoints, watchpoints, and other debugging exceptions are monitored during software debugging
- PendSV: Suspend service call, typically used by the operating system for context switching.
- SysTick: An exception generated by the system timer, it can be used as a timer for OS environments
- External Interrupt: Used for peripheral interrupts

The Arm architecture provides several programmable registers for interrupt and exception management. These registers are mostly located in the NVIC and system control block (SCB). The Arm architecture also provides registers for interrupt masking, such as PRIMASK, FAULTMASK, and BASEPRI. Once the chip is reset, all interrupts will be in the disabled state, and the default priority level will be 0. Before using an interrupt, the priority level of the required interrupt needs to be configured, and the interrupt of the peripheral modules needs to be enabled, as well as the NVIC interrupt.



### 2.4.3 Interrupt and Trap Vector Tables

There are 2 vector tables in the TriCore architecture, the interrupt vector table and the trap vector table.

#### ➤ Interrupt vector table

The Base of Interrupt Vector Table Register (BIV) stores the base address of the interrupt vector table. Before the interrupt is enabled, the MTCR instruction can be used to modify the BIV register during system initialization. The interrupt vector table base address in the BIV register must be aligned to an even byte address (half-word address).

When an interrupt is generated, the CPU will calculate the corresponding interrupt service function entry using the contents of the PIPN and BIV registers. There are two vector table spacings available. One is 32 bytes and the other is 8 bytes. The vector table spacing is determined by the VSS bit of the BIV register. The specific formula is as follows:

if (BIV.VSS == 1'b0)

ISR\_Entry\_PC = {BIV[31:1], 1'b0} | {PIPN << 5};

Else

ISR\_Entry\_PC = {BIV[31:1], 1'b0} | {PIPN << 3};

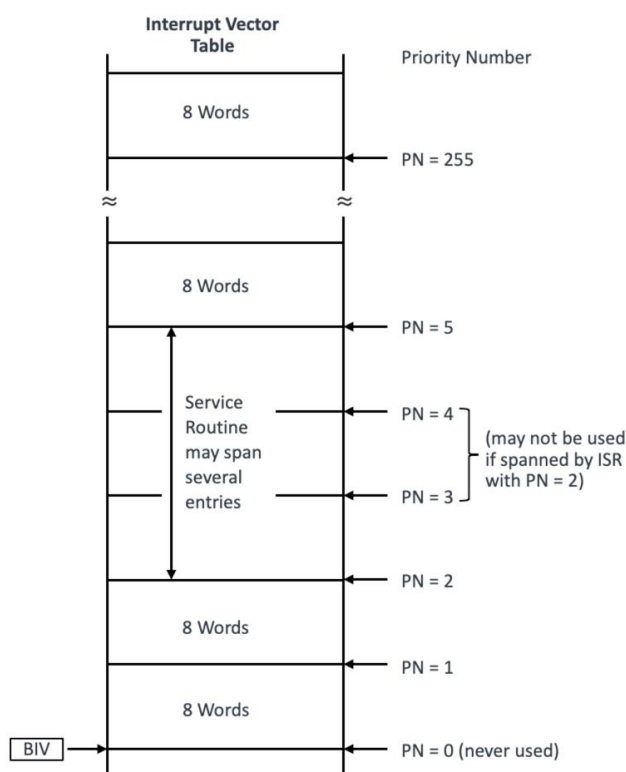


Figure 2.4.3-1 TriCore interrupt vector table

#### ➤ Trap vector table

The Base Trap Vector Table Pointer (BTV) stores the base address of the trap vector table. When a trap is generated, the entry address of a trap handling function is determined by left-shifting the Trap Class by 5 bits and performing an OR operation with the value of the BTV register. Left shifting the Trap Class by 5 bits will result in 32 byte spacing between the different trap handling function entries. Therefore, the BTV register value has to be at least 256 byte edge aligned.



Vector tables in the Arm architecture start from address 0 by default. The vector address is the exception number multiplied by 4. When the chip is enabled, the initial memory (flash or ROM) address (normally 0x00000000) will be used as the initial value of SP\_main, and it cannot be changed during execution. However, in some usage scenarios, the vector table may need to be modified or relocated. Therefore, the Arm architecture includes the ability to relocate the vector table. The interrupt vector table relocate function is achieved through the use of the programmable register of the Vector Table Offset Register (VTOR). The interrupt vector table must be aligned to addresses that are powers of 2, with a minimum alignment requirement of 128 bytes.

Porting tip: The interrupt and exception structures of the two architectures are not consistent. During porting, the Arm architecture interrupt and exception handling method must be completely followed.

## 2.5 Memory

Memory is used for the addressing space in a chip core. The data stored in the memory is used for logic processing in the chip core. The design of memory partitions can directly impact the performance of the core when it is processing tasks. This chapter will describe the differences between the two core architectures in terms of the memory address space, addressing mode, and cache.

### 2.5.1 Memory Address Space

Addresses in the TriCore architecture are 32 bits wide, which allows them an access range of up to 4GB. The address space is divided into the 16 segments [0H - FH]. Each segment is 256MB and can act as peripheral space, cache space, or non-cache space.

The physical memory attributes of the [0H - 7H] memory segments depend on the specific implementation requirements. If MMU is enabled, the [0H - 7H] memory segments are treated as virtual addresses. They must be converted when they are accessed. If MMU is not used, the access characteristics will depend on the specific implementation requirements. Unauthorized access may lead to traps.

The Scratch Pad RAM (SRAM) of the TriCore architecture supports program segments and data segments in the C segment (PSPR) and D segment (DSPR) respectively. In a multi-core architecture, each CPU's data memory (DSPR) and program memory (PSPR) access different memory regions by utilizing mirror image regions for DSPR and PSPR. These mirror image regions are distributed within the [0H - 7H] memory segments.

Segment	Properties
D <sub>H</sub>	DSPR region
C <sub>H</sub>	PSPR region
7 <sub>H</sub>	CPU-0 PSPR and DSPR memory image region
6 <sub>H</sub>	CPU-1 PSPR and DSPR memory image region
5 <sub>H</sub>	CPU-2 PSPR and DSPR memory image region
4 <sub>H</sub>	CPU-3 PSPR and DSPR memory image region
3 <sub>H</sub>	CPU-4 PSPR and DSPR memory image region
2 <sub>H</sub>	CPU-5 PSPR and DSPR memory image region
1 <sub>H</sub>	CPU-6 PSPR and DSPR memory image region
0 <sub>H</sub>	CPU-7 PSPR and DSPR memory image region

Figure 2.5.1-1 TriCore SRAM memory segments

The [8H - DH] memory segments are used for defining non-volatile storage spaces and special storage regions such as program flash memory (PFLASH), data flash memory (DFLASH), chip firmware (BROM), and local memory (LMU). Furthermore, the [8H - 9H] memory segments allow cache access, but the [AH - DH] segments do not.

The [EH - FH] memory segments are used to define the access region of peripherals. The memory regions for chip peripherals, such as the peripheral register, are allocated to these segments.

The Arm architecture is similar to the TriCore architecture, both adopting 32-bit address widths and capable of accessing up to 4GB of addressing range. Unlike the TriCore Architecture, the Arm architecture divides the 4GB addressing space into 8 segments. The segments are each 0.5GB in size and used for the following storage functions:

- Code
- SRAM
- Peripheral
- Two RAM regions
- Two Device regions
- System

A comparison of TriCore and Arm addressing spaces is as follows:

Addressing Space	Arm Architecture	TriCore Architecture
0x00000000-0x1FFFFFFF	Code, used for Flash, ROM, and other non-volatile memory regions.	Addressing space divided into DSPR for the CPU data memory and PSPR for the program memory.
0x20000000-0x3FFFFFFF	SRAM, used for the memory region in the chip.	
0x40000000-0x5FFFFFFF	Peripheral, used for the addressing region for chip peripherals, such as the peripheral register addressing region.	

Porting from the TriCore Architecture to the Arm Architecture

0x60000000-0x7FFFFFFF	Two RAM regions, used for the internal RAM region addressing of chip peripherals. The write-back (WB) method is used as the cache strategy for this addressing region.	
0x80000000-0x9FFFFFFF	Two RAM regions, used for the internal RAM region addressing of chip peripherals. The write-through (WT) method is used as the cache strategy for this addressing region.	PFLASH, DFLASH, BROM, and LMU, cache access is allowed.
0xA0000000-0xBFFFFFFF	Two Device regions, used for the shared device regions.	PFLASH, DFLASH, and BROM, cache access is not allowed.
0xC0000000-0xDFFFFFFF	Two Device regions, used for the non-shared device regions.	Scratch Pad RAM (SRAM), supports program and data segments.
0xE0000000-0xFFFFFFFF	System, used for the Private Peripheral Bus (PPB) addressing range and supplier-defined memory addressing.	Access regions of peripherals and the peripheral memory regions of the chip, such as the peripheral registers.

Porting tip: The addressing spaces of the two architectures are different, therefore it is necessary to consider the data and program storage spaces when porting. In the Arm architecture, data segments need to be stored in the SRAM, while program segments and constant data segments need to be stored in Code when compiling links.

### 2.5.2 Addressing Modes

The TriCore addressing mode accesses memory data through the load and store commands. The length of the accessed data can 8 bits, 16 bits, 32 bits, or 64 bits. The TriCore architecture supports seven addressing modes:

Addressing Mode	Description
Absolute Addressing	Generally used for peripheral registers and global data access. Absolute addressing uses the 18-bit constant defined by the command as the memory address. The complete 32-bit address is formed by taking the upper 4 bits of the 18-bit constant and placing them in the upper 4 bits of the 32-bit address, while filling the remaining empty positions with 0.
Base + Short Offset	The effective address in this addressing mode is the sum of the base address register and a 10-bit offset of the sign extended bit.
Base + Long Offset	In contrast to Base + Short Offset, the offset is a 16-bit sign-extended value, and any location in the memory can be addressed using a two-instruction sequence.

Pre-increment	Commonly used for stack push operations, it utilizes the sum of an address register and an offset as the effective address and then writes the result back to the address register.
Post-increment	This addressing mode is typically used for stack data pop (pop from stack) operations. It uses the value in an address register as the effective address, then adds the sign-extended 10-bit offset to the previous value, and finally updates the address register.
Circular	Typically used to access data values in a circular buffer when performing filtering calculations.
Bit-reverse	Used for Fast Fourier Transform (FFT) algorithm calculations.

The 8 types of addressing modes in the Arm architecture include:

- Register addressing
- Immediate addressing
- Register offset addressing
- Register indirect addressing
- Base addressing
- Stack addressing
- Relative addressing
- Multi-register addressing

Porting tip: For developers, the addressing modes of the two architectures differ significantly but in typical development scenarios, addressing modes are not a significant focus. However, for specific use cases, such as those with high-performance requirements, different addressing modes and instruction sets may be used. For more information related to instruction sets, please refer to Chapter 2.2 [Instruction Set](#).

### 2.5.3 Cache

The TriCore architecture supports caching, including both data caching and instruction caching. If instruction caching is enabled, the CPU can perform prefetching when fetching instructions from memory. Furthermore, if data caching is enabled, the CPU can also perform prefetching when fetching data from memory. In the addressing space of the TriCore architecture, the region for caching is the [8H - 9H] segments, which is very limited, as shown in the following figure:

Segment	Attributes
F <sub>H</sub>	Peripheral Space.
E <sub>H</sub>	Peripheral Space.
D <sub>H</sub>	Non-cacheable Memory.
C <sub>H</sub>	Non-cacheable Memory.
B <sub>H</sub>	Non-cacheable Memory.
A <sub>H</sub>	Non-cacheable Memory.
9 <sub>H</sub>	Cacheable Memory.
8 <sub>H</sub>	Cacheable Memory.
7 <sub>H</sub> - 0 <sub>H</sub>	Non-cacheable Memory.

Figure 2.5.3-1 TriCore architecture caching regions

The caching supported by the TriCore architecture also has the following limitations:

- The addressing space for peripherals cannot be cached.
- Local DSPR data cannot be saved in the local data cache.
- Local PSPR data cannot be saved in the local instruction cache.

The Arm architecture supports caching operations, including both data access caching and instruction caching. The general cache architecture is shown in the figure below:

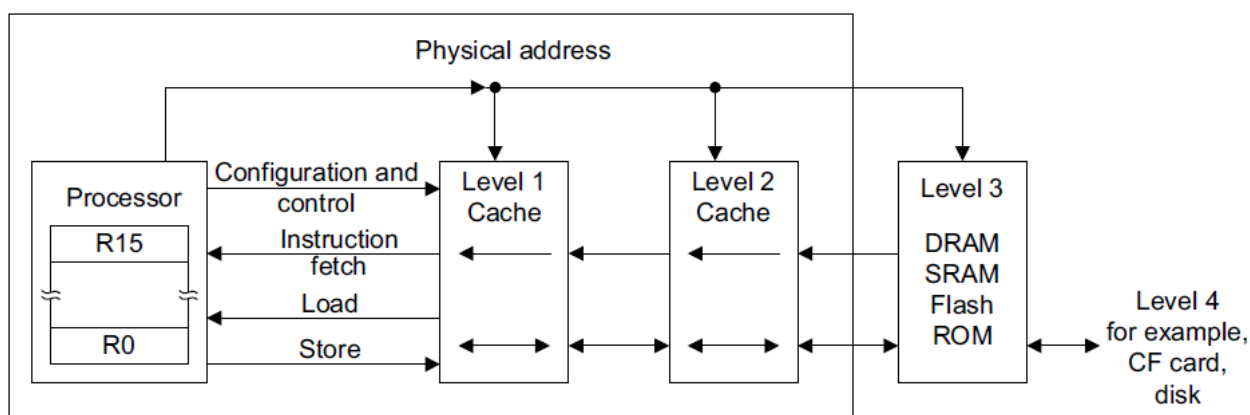


Figure 2.5.3-1 Multi-level cache architecture

The Arm architecture used in the ARMv7-M series cores and the TriCore architecture are both based on the load/store architecture. The Cortex-M7 core of the ARMv7-M series is equipped with a local (L1) instruction cache (I-Cache) and data cache (D-Cache). The Arm architecture provides the memory prompt capabilities Preload Data (PLD) and Preload Instruction (PLI). This function allows the software to notify the hardware of the memory position expected to be used. The memory system can quickly respond to data access.

Porting tip: For developers, while caching can improve data access performance, it can also create some issues for users. A typical issue with caching is data access consistency. For example, when using DMA to read data from the CPU's data cache, if the CPU writes new data into the data cache while the

Porting from the TriCore Architecture to the Arm Architecture

DMA reads old data that is still stored in the data cache, inconsistencies in the data may occur. To avoid such issues, developers must use caches carefully.

## 2.6 Floating Point Unit

The TriCore architecture supports Floating Point Unit (FPU) and IEEE-754 single-precision floating-point arithmetic instructions, which include floating point addition, subtraction, multiplication, division, multiply-accumulate (MAC), etc. The TriCore architecture supports the IEEE-754 single-precision format as well as conversions between signed and unsigned integers and 32-bit signed fractions in the TriCore architecture. In addition, the TriCore architecture also supports the comparison of floating point values and the four rounding modes of IEEE-754.

The FPU of the TriCore architecture has some limitations, for example it only supports the IEEE-754 single-precision format. The FPU does not support arithmetic operations on IEEE-754 denormalized values. Furthermore, because the FPU of the TriCore architecture does not support denormalized floating point numbers, it does not completely comply with the IEEE-754 standard. Remainder functions that conform to the IEEE-754 standard cannot be implemented using FPU instructions, and Fused MACs do not support the IEEE-754 standard.

The Arm architecture also supports floating point operations, which include the two versions of floating point FPUv4-SP and FPUv5. Both versions support single-precision 32-bit floating point operations. However, FPUv5 provides additional instructions and supports double-precision 64-bit floating point operations. The Arm architecture also supports the IEEE-754 standard.

Porting tip: When using floating point operations, developers must consider the level of support for floating point operations provided by both architectures. Unlike the TriCore architecture, the Arm architecture supports double-precision floating point operations.

## 2.7 Debug

The debug system of the TriCore architecture performs core debug through the Core Debug Controller (CDC), which allows access to the memory space of the core and chip. The CDC mainly provides support for the software development environment, including real-time control of core operations and restart, access to and update of internal registers and memory data, and configuration of the breakpoints and watchpoints of complex trigger conditions.

The debug system of the Arm architecture accesses the core and memory regions through the Debug Access Port (DAP). The functions of the debug system include core operations and suspension, single-step operations, register and memory access, etc.

The debug systems of the TriCore architecture and Arm architecture support standard JTAG connectors and trace functions. The TriCore architecture also supports a two-wire Device Access Port (DAP). Compared to a JTAG, a DAP provides faster debug speed and fewer debug pins. Similarly, the Arm architecture supports a two-wire debug interface called Serial Wire Debug (SWD). Furthermore, the Arm architecture supports custom coresight functions, providing additional debug and trace functions. It is able to perform debug on an entire System-on-Chip (SOC).

### 3 COMPARISON OF FUNCTIONAL SAFETY DESIGN

The TriCore architecture and Arm Cortex-M series are widely used in the automotive field. As the automotive industry continues to advance, the safety and stability requirements for automotive systems also become stricter. The ISO 26262 standard for functional safety for road vehicles lists higher requirements for automotive systems. Within the ISO 26262 standard, different ASIL/SIL levels are defined based on risk assessment and analysis, and specific target indicators that need to be achieved are provided. ASIL D represents the highest level of potential risk, requiring the most rigorous approaches to fault management. In order to satisfy the functional safety requirements, chips that support the TriCore architecture and Arm architecture have proposed different solutions for functional safety, which will be described using specific chip series. The TriCore architecture will be analyzed based on the TC3xx series chips, while the Arm architecture will be analyzed based on the Cortex-M7 series chips. Both series of chips meet ASIL D requirements.

#### 3.1 Core Safety

Core safety in the TC3xx chips is implemented using a “lockstep” approach. The lockstep function is implemented through a master core and checker core. When the master core is executing logical processing, the checker core similarly performs logical processing on the inputs to the master core. After both cores have completed processing, a logical comparator compares the processed results of the two cores to verify if they are consistent. To prevent common-cause failures, the inputs to the checker core are delayed by 2 clock cycles.

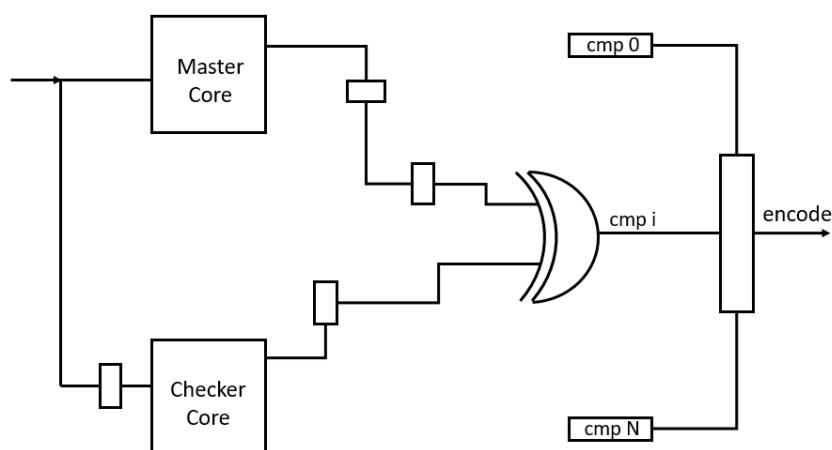


Figure 3.1-1 TC3xx core comparator

Core safety in the Cortex-M7 series chips is also implemented through a lockstep approach. The Cortex-M7 provides a configurable option to implement dual-core lockstep, which involves designing a duplicate of the computational core. This function can effectively enable the necessary fault detection, meeting the ASIL D hardware standard for the computational core. Cortex-M7 redundant logic runs 2 clock cycles behind the main logic to prevent common-cause failures.



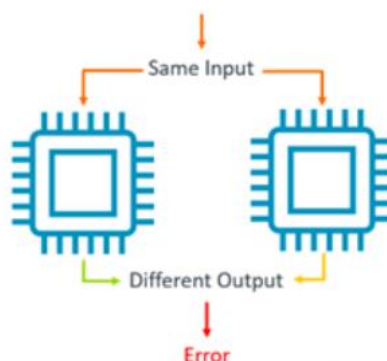


Figure 3.1-2 Cortex-M7 lockstep mechanism

For single cores without lockstep mechanisms, additional mechanisms need to be implemented to meet functional safety requirements, such as a Software-Based Self-Test (SBST) for the core. The SBST is a software-based testing method that checks the logical processing functionality of the core. It requires real-time monitoring of the core's execution state while the core is running. However, even with additional safety mechanisms, non-lockstep cores can only meet requirements as high as ASIL B.

### 3.2 Memory Protection

In addition to core safety, ISO 26262 also includes the concept of Freedom From Interference (FFI). It requires that when data interchange occurs between modules with different ASIL levels, isolation and protection measures must be adopted in the memory space to prevent lower ASIL level modules from affecting higher ASIL level modules. Therefore, the Memory Protection Unit (MPU) is a necessary safety mechanism for the chip.

Memory protection in the TC3xx chip is implemented based on address range, providing protection for both program and data regions. The memory protection of the TriCore architecture supports up to 6 protection sets, with up to 18 data protection segments and 10 program protection segments.

The TC3xx chips support the memory protection functions of bus MPU. Compared to core memory protection, bus MPU can provide restrictions on access from slave memory to the bus master.

ARMv7-M chips support the protected memory system architecture PMSAv7. The system address space implemented by PMSAv7 is protected by the MPU. The MPU divides the memory into several regions, with the Cortex-M7 supporting up to 16 protected regions. The location and size of each region are configurable. The size of each region must be a power of 2 but cannot be smaller than 32 bytes.

For developers, memory protection functions are generally implemented by the operating system. Memory protection is configured through context switching within the operating system. When the operating system is ported, the differences between the two CPU architectures need to be considered for memory protection porting.



## 4 SOFTWARE DEVELOPMENT AND PORTING

The software for porting described in this chapter is typically developed using high-level languages, such as the C language. The C language has features such as easy compilation and cross-architecture compatibility, making it widely used in embedded systems. In the actual porting process, there may be assembly code dependencies specific to the architecture, especially in the boot code and exception handling programs. However, each architecture provides its own programs for users to reference, making the process more convenient to users.

The process of porting software from the TriCore architecture to the Arm architecture is described through the following aspects:

- Development toolchain
- Chip startup process
- Exception and interrupt handling
- Peripheral access

### 4.1 Development Toolchain

The most prominent development environments for the TriCore architecture include Tasking and HighTec, and Infineon's miniwiggler debugger or professional debuggers like Lauterbach and iSYSTEM can also be used. For the Arm architecture, the development environments primarily include the Arm Compiler, IAR, GreenHills, KEIL, and GNU GCC. JLINK or professional debuggers such as Lauterbach and iSYSTEM can be used for debugging.

Developers must consider the differences between the compilation environments when porting TriCore architecture software to the Arm architecture, which will be analyzed based on the following aspects:

- Assembly code  
Typically, to improve the execution efficiency of programs, the program startup code and exception handling program will be written with assembly code. During porting, the differences between the assembly instructions for different architectures need to be considered. For details, refer to Chapter 2.2 [Instruction Set](#). Typically, the IDE provides examples related to different architectures to the user. The startup code, exception handling, and other programs of the ported architecture can be replaced with the corresponding programs of the target architecture.
- Data types  
Different compilers support different data types for various architectures, and the specific information can be found in the compiler's type definitions. For details, refer to Chapter 2.1.1 [Data Type](#).
- Compilation and linking options  
The options available in different compilers need to be considered during the porting process, such as each compiler's support for various C language standards (C90, C99, etc.), adherence to different ANSI standards, and optimization options for compilation and linking. Different compilation and linking options will affect the execution results of the generated executable files.

➤ **Linker file**

During the linking process of the compiler, the programs are assigned to different addressing spaces according to the linker file, such as the SRAM, PFLASH, and stack regions. Different compilers use different linker file formats. Typically, the IDE will provide users with linker file templates for different architectures. During the porting process, users need to replace the code sections of the ported architecture with the target architecture's corresponding sections.

## **4.2 Chip Startup**

When the chip is powered on, it is typically necessary to initialize the internal registers and other chip components before executing the user-developed program (main function). The program can only run after the chip has been fully initialized.

Using the TC3xx chip as an example for the startup process of the TriCore architecture, normally during startup the boot firmware will determine the code location to jump to based on the startup address defined in the boot mode header (BMHD). When the program jumps to the startup code location, it initializes the chip's core registers.

The initialization process is as follows:

1. Configure the cache mechanism by enabling or disabling data caching and program caching.
2. Initialize address registers A0, A1, A8, and A9, and the system global registers.
3. Initialize global variables and perform clear operations on uninitialized data (e.g. .bss or .sbss segments).
4. Initialize the base address registers BIV and BTV of the interrupt and exception vector tables. The base address register values are configured according to the linker file.
5. Initialize the stack register. The base address of the stack register is configured according to the stack address in the linker file.
6. Initialize the global variables by setting the initial data (such as the data segments) to the actual initial values. Assign the data in the Flash region to the global variables of the RAM region.
7. Initialize the system registers, such as MPU and CSA.
8. Initialize the peripheral modules, such as the clock and I/O.
9. Enter StartOS

Chip bootup for the Arm architecture begins at the reset vector's position in the exception vector table, followed by the initialization process:

1. Configure the cache mechanism by enabling or disabling data caching and program caching.
2. Initialize the CPU core register.
3. Initialize the stack register.
4. Initialize modules such as the MPU and FPU.
5. Initialize the global variables, such as the global variables of the .bss and .data segments.
6. Initialize the peripheral modules, such as the clock.
7. Enter the operating system (OS).

During the porting process, developers need to consider the differences between the two startup processes of the two architectures. Typically, the IDE will provide examples of startup codes for Porting from the TriCore Architecture to the Arm Architecture

different architectures to users. In most cases, the majority of configurations do not require users to make modifications. The configurations can simply be replaced with the startup code for the target architecture. However, some specific modules need to be configured. For example, the MPU module needs to be configured according to the original requirements. The values of stack registers need to be configured based on the pointers in the linker file, and the initialization of global variables needs to be configured according to the partitions in the linker file.

### 4.3 Exception and Interrupt Handling

For developers, during the porting process, the interrupt and exception handling of the Arm architecture must be implemented according to the interrupt and exception handling modes of the Arm architecture. Refer to Chapter 2.4.3 [Interrupt and Trap Vector Tables](#) to configure the interrupt vector table. The interrupt vector table of the IAR compiler as an example, as shown in the following figure:

```
__vector_table
DCD     sfe(CSTACK)
DCD     Reset_Handler

        DCD     NMI_Handler
        DCD     HardFault_Handler
        DCD     MemManage_Handler
        DCD     BusFault_Handler
        DCD     UsageFault_Handler

__vector_table 0x1c
DCD     0
DCD     0
DCD     0
DCD     0
DCD     SVC_Handler
DCD     DebugMon_Handler
DCD     0
DCD     PendSV_Handler
DCD     SysTick_Handler
```

Figure 4.3-1 Arm architecture interrupt vector table

Interrupts and exceptions share a single interrupt vector table in the Arm architecture. The exception handling process is described below.

- Trigger conditions for exceptions
  1. The core is in operation.
  2. The exception and interrupt source is in an enabled state.
  3. The priority level of the exception is higher than that of the current exception being handled.
  4. The exception is not masked by the exception masking register.
- Exception entry process
  1. Push onto the stack and save the current registers and return addresses. If the processor is in thread mode, the process stack pointer (PSP) will be used as the stack pointer, and if the processor is in exception mode, the main stack pointer (MSP) will be used as the stack pointer.
  2. Obtain the exception vector
  3. After confirming the initial address of the exception handling function, retrieve the instruction to be executed for exception handling.
  4. Update the NVIC register and processor register.

➤ Exception handling process

Specific handling tasks can be executed in the exception handling function. The processor will be in handler mode. In addition, the MSP is utilized in privileged access mode, and it supports interrupt preemption and interrupt nesting.

➤ Exception return process

Exception return will use special instructions in some processor architectures. This means that exception handling will not be written and compiled like normal C code. For Cortex-M processors, the exception return mechanism is triggered by a special address called EXC\_RETURN. This value is generated at the exception entry point and stored in the link register LR. When this value is written into the PC by a valid exception return instruction, it triggers the exception return process.

#### 4.4 Peripheral Access

The peripherals of Arm core chips will vary depending on the specific implementations by different chip manufacturers. The address space for Arm architecture chip peripherals is typically in the range of 0x40000000 to 0x9FFFFFFF. For specific addressing control, refer to Chapter 2.5.1 [Memory Address Space](#).

Normally, chip manufacturers will provide users with peripheral drivers, therefore developers can use the peripheral drivers provided by chip manufacturers during the porting process. When integrating into the AUTOSAR architecture, developers need to understand the standard interfaces of various peripheral modules within the AUTOSAR architecture. The requirements of AUTOSAR can be met by using standard interfaces for chip packaging. For more details, refer to Chapter 5.2 [MCAL Porting](#).

## 5 AUTOSAR ARCHITECTURE PORTING

### 5.1 AUTOSAR Architecture

The current mainstream software architecture for automotive electronics is the Automotive Open System Architecture (AUTOSAR). It is a development partnership founded in 2003 that pursues the objective of creating a standardized software architecture for automotive electronics (the partners include automotive manufacturers, parts suppliers, and research and service institutions from around the world). AUTOSAR has stipulated a set of dedicated open frameworks and industry standards for automobiles, which will serve as the fundamental infrastructure for managing future applications and standard software modules in the automotive industry.

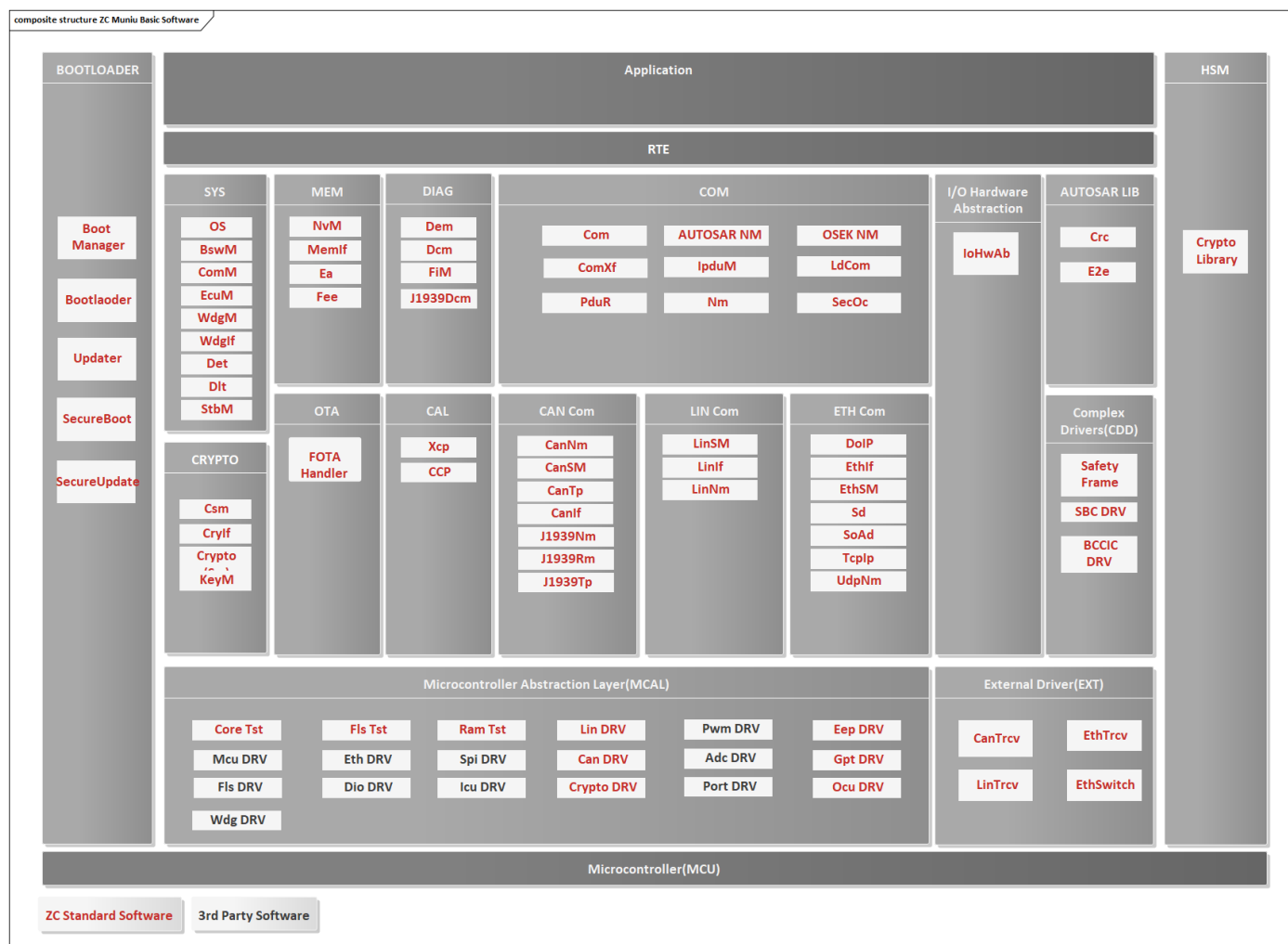


Figure 5.1-1 ZC Muniu AUTOSAR architecture

The layered architecture and unified interfaces of AUTOSAR make porting more convenient. Developers only need to replace the modules dependent on architecture, such as the microcontroller layer (MCAL) and operating system (OS) module, when porting software developed according to the AUTOSAR architecture from the TriCore architecture to the Arm architecture.

### 5.2 MCAL Porting

In the architecture of microcontroller units (MCUs), in addition to the core, there are also chip peripherals, such as the analog-to-digital converter (ADC), general-purpose input/output (GPIO), clock

Porting from the TriCore Architecture to the Arm Architecture

module, CAN communication module, timer module, etc. These peripheral modules must be called to implement the final operating logic functions.

MCAL drivers have standardized the peripheral module drivers by abstracting the functionality and interfaces of the peripheral modules. This standardization facilitates porting between different architectures. MCAL divides the peripheral modules into several driver groups, which include:

- **Microcontroller drivers**  
Microcontroller drivers are responsible for the basic core and peripheral configurations of the MCU. They mainly include the MCU driver (MCU), watchdog driver (WDG), and general purpose timer driver (GPT).
- **Memory drivers**  
Memory drivers provide control functions for the on-chip storage (including internal flash and internal EEPROM). They mainly include the internal flash driver and internal EEPROM driver.
- **Communication drivers**  
Communication drivers provide control functions for ECU communication peripherals and automotive network communication peripherals. They mainly include the SPI driver (SPI), LIN driver (LIN), CAN driver (CAN), FlexRay driver (FR), and Ethernet driver (ETH).
- **I/O drivers**  
I/O drivers are the drivers for MCU on-chip input and output modules. They include the port control driver (PORT), digital I/O pin driver (DIO), analog-to-digital converter driver (ADC), pulse-width modulation output driver (PWM), input capture driver (ICU), and output comparison driver (OCU).
- **Crypto drivers**  
Crypto drivers are the drivers for on-chip encryption modules.

Because the implementation of peripheral devices can vary significantly between different chips, including differences in internal logic circuits and peripheral control registers, porting the MCAL module requires referencing AUTOSAR's requirements for each module. MCU and ADC driver porting are used as examples here.

#### 1. MCU driver porting

MCU drivers mainly implement the clock, reset, and mode management for the MCU. The initialization process of the MCU driver is shown in the figure below:

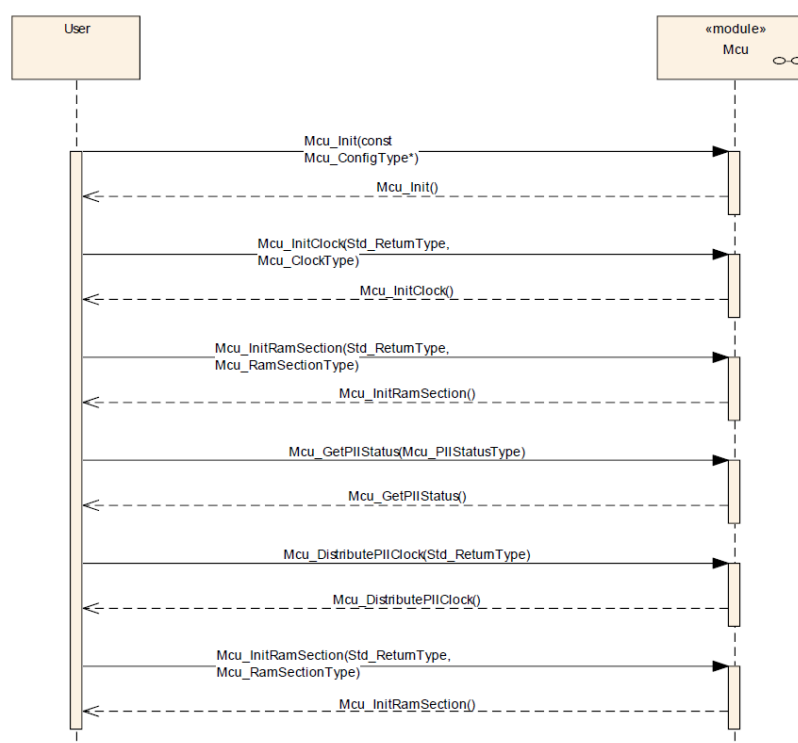


Figure 5.2-1 MCU initialization process

The following points must be considered when porting the MCU:

- When the MCU initializes the clock, the system clock, bus clock, and various peripheral module clocks must be configured according to the internal clock of the MCU. The functions must be implemented in the `Mcu_InitClock()` and `Mcu_DistributePllClock()` interfaces.
- MCU module management must be implemented through the `Mcu_SetMode()` interface and configured according to the different internal modes of the chip.
- For MCU reset management, obtaining the reset reason and chip reset must be performed through the `Mcu_GetResetReason()` and `Mcu_PerformReset()` interfaces.

## 2. ADC driver porting

The ADC driver implements the configuration of the ADC module, including configuring the trigger source for ADC conversion, enabling or disabling ADC conversion, providing a notification mechanism for ADC conversion, and querying the conversion status and results. Additionally, the ADC driver offers a grouping mechanism, where different ADC channels are assigned to different groups, allowing individual ADC channel groups to be managed separately.

For ADC porting, the following points must be considered according to the ADC module of the chip:

- Supports different conversion modes: One-shot conversion and continuous conversion modes

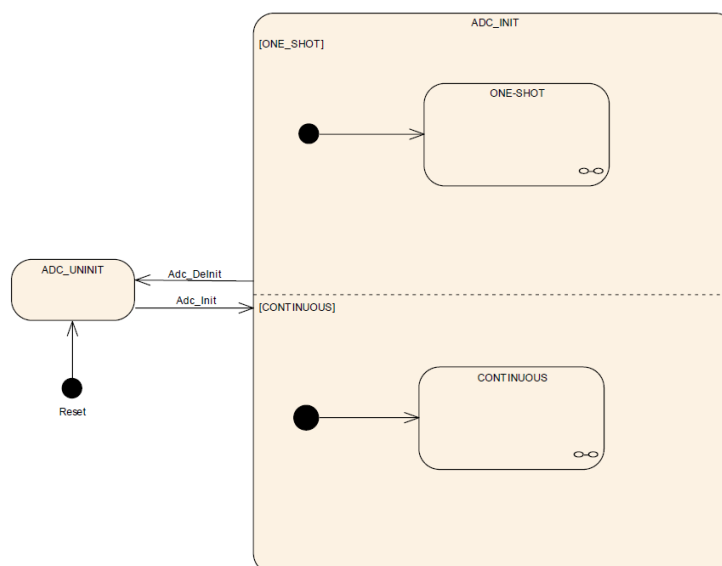


Figure 5.2-2 ADC conversion mode

- Supports different conversion trigger sources: Hardware triggers and software triggers

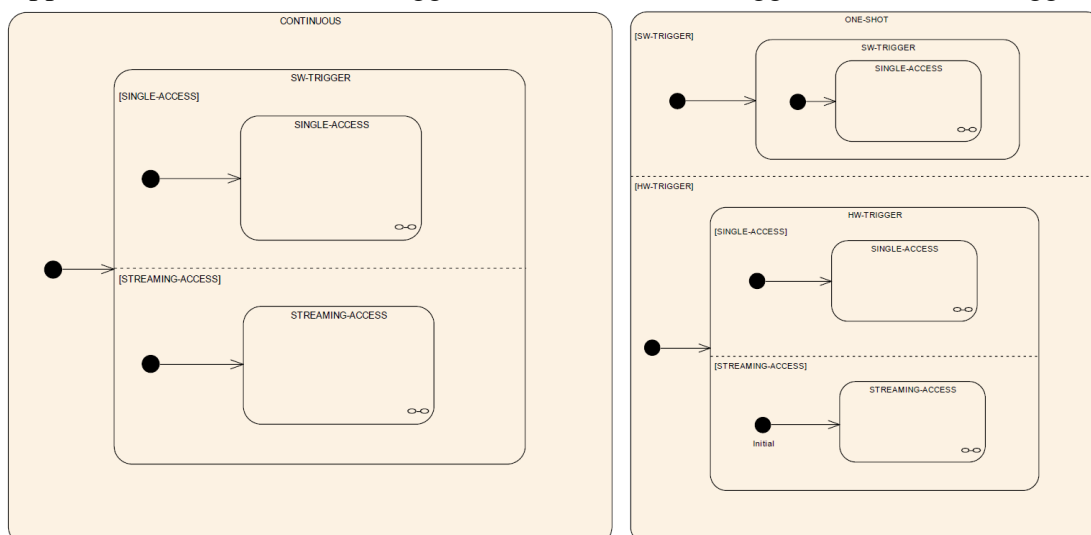


Figure 5.2-3 Triggers under different conversion modes

- Supports the priority level of each channel
- Supports different result access methods

### 5.3 AUTOSAR Operating System Porting

The AUTOSAR operating system is a real-time operating system that provides interrupt handling, task scheduling, system timing, and clock synchronization functions. The AUTOSAR operating system provides some additional features based on the OSEK operating system, such as memory protection and time protection. The AUTOSAR OS is offered in 4 scalability classes: SC1, SC2, SC3, and SC4, as shown in the figure below:



Feature	Described in Section	Scalability Class 1	Scalability Class 2	Scalability Class 3	Scalability Class 4	Hardware requirements
OSEK OS (all conformance classes)	7.1	✓	✓	✓	✓	
Counter Interface	8.4.17	✓	✓	✓	✓	
SWFRT Interface	8.4.18, 8.4.19	✓	✓	✓	✓	
Schedule Tables	7.3	✓	✓	✓	✓	
Stack Monitoring	7.5	✓	✓	✓	✓	
ProtectionHook	7.8		✓	✓	✓	
Timing Protection	7.7.2		✓		✓	Timer(s) with high priority interrupt
Global Time /Synchronization Support	7.4		✓		✓	Global time source
Memory Protection	7.7.1, 7.7.4			✓	✓	MPU
OS-Applications	7.6, 7.12			✓	✓	
Service Protection	7.7.3			✓	✓	
CallTrustedFunction	7.7.5			✓	✓	(Non-)privileged Modes

Figure 5.3-1 AUTOSAR OS Scalability classes

The implementation of the operating system is mainly dependent on the core architecture of the chip. For example, on a TriCore architecture, it utilizes the general purpose registers A0-15, D0-D15, PSW, PCXI, DPR0L, DPR0H, and system-specific registers. Additionally, the operating system employs TriCore's unique context management mechanism known as Context Save Areas (CSA).

The following points have to be considered for Arm architecture operating system porting:

- The special applications of the core general purpose registers R0 to R12 and the special registers R13 to R15 must be considered. For details, refer to Chapter 2.3 [General Purpose Registers](#).
- Exception and interrupt handling must be implemented through the specific mechanisms of the Arm architecture. For details, such as PendSV handling and SVC handling, refer to Chapter 2.4 [Exceptions and Interrupts](#).
- When the operating system handles tasks or exceptions, the operating mode of the Arm architecture needs to be considered. The privilege access modes in handler mode and thread mode are different. Unprivileged mode tasks accessing special registers can be implemented using the Supervisor Call (SVC) instruction.
- The memory protection functions required for functional safety must be configured based on the MPU functions of the Arm architecture. For details, refer to Chapter 3.2 [Memory Protection](#). During memory protection switching, task-specific stack management should also be considered.

## 6 SUMMARY

This manual primarily describes the differences between the TriCore and Arm architectures, and outlined the process of porting embedded software and the AUTOSAR architecture from the TriCore architecture to the Arm architecture, as well key points to take into account.

The layered design and unified interfaces of the AUTOSAR architecture facilitate cross-architecture software porting. The Arm Cortex-M series processors were specifically designed for the microcontroller market and embedded systems. Their real-time performance, memory protection, interrupt handling, debugging capabilities, software tools and libraries, and scalability make the AUTOSAR architecture easier and more convenient to port and develop in.

## 7 APPENDICES

### 7.1 Appendix 1 Reference materials

No.	Description	Version	Date
1	Armv7-M Architecture Reference Manual.pdf <a href="https://developer.arm.com/documentation/ddi0403/latest/">https://developer.arm.com/documentation/ddi0403/latest/</a>	E.e	15 February 2021
2	Arm® Cortex®-M7 Processor Technical Reference Manual.pdf <a href="https://developer.arm.com/documentation/ddi0489/latest/">https://developer.arm.com/documentation/ddi0489/latest/</a>	r1p2	15 November 2018
3	Arm® Compiler armclang Reference Guide.pdf Reference link: <a href="https://developer.arm.com/documentation/100067/0612?lang=en">https://developer.arm.com/documentation/100067/0612?lang=en</a>	6.12	27 February 2019
4	Infineon-AURIX_TC3xx_UserManual	V02_00	
5	Infineon-AURIX_TC3xx_Architecture_UserManual	V01_00	
6	ZC.MuNiu Basic Software Platform Manual.pdf Reference link: <a href="http://www.shzckj.cn/portal/article/product_detail.html?id=41">http://www.shzckj.cn/portal/article/product_detail.html?id=41</a>	/	/
7	AUTOSAR_SWS_ADCDriver.pdf	4.3.1	2017-12-08
8	AUTOSAR_SWS_MCUDriver.pdf	4.3.1	2017-12-08

9	AUTOSAR_SWS_OS.pdf	4.3.1	2017-12-08
---	--------------------	-------	------------

## 7.2 Appendix 2 Terminology and abbreviations

Terminology/Abbreviation	Description
RISC	Reduced Instruction Set Computer
MMU	Memory Management Unit
RTOS	Real Time Operating System
DSP	Digital Signal Processing
ISA	Instruction Set Architecture
DMA	Direct Memory Access
SBST	Software-based Self-test
MPU	Memory Protection Unit
ASIL	Automotive Safety Integrity Level
IDE	Integrated Development Environment
AUTOSAR	Automotive Open System Architecture
SC	Scalability classes
CSA	Context Save Areas