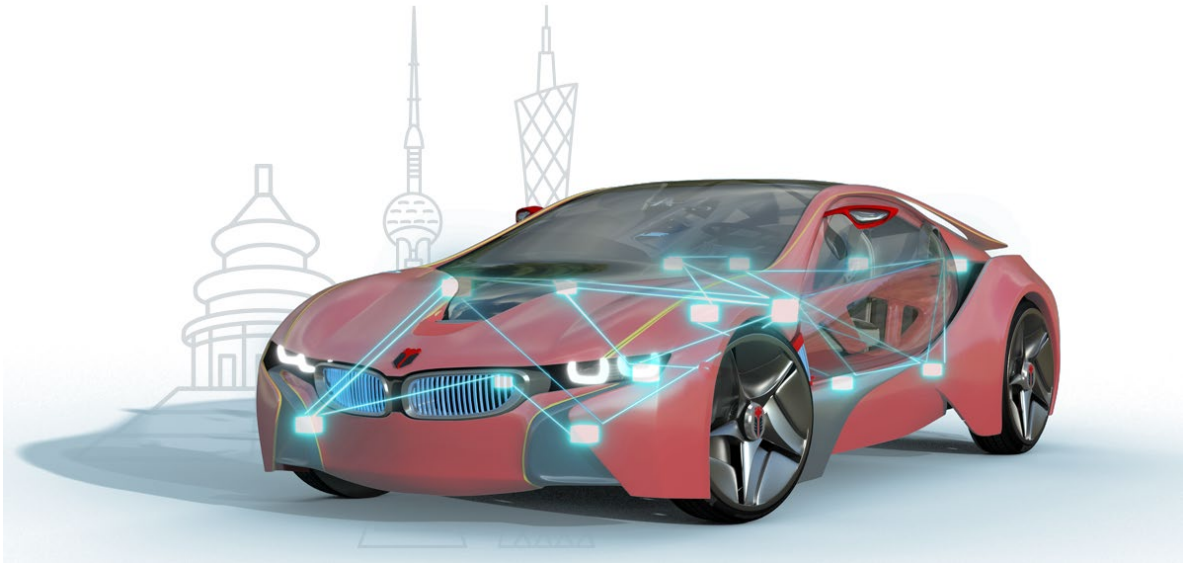
# MIGRATION GUIDE FROM TRICORE PLATFORM TO ARM CORTEX-R52/R52+

ZC® ENGINEERING SERVICE



# MIGRATION GUIDE FROM TRICORE PLATFORM TO ARM CORTEX-R52/R52+

ZC® ENGINEERING SERVICE

## CHAPTER 1 OVERVIEW

This document provides a guide for migrating from the Tricore platform to the ARM Cortex-R52/R52+.

### 1.1 Overview of Arm Architecture

ARM is a set of related computing technology utilizing the RISC architecture, originally developed by Advanced RISC Machines Limited. The ARM architecture is extensively utilized in embedded systems design and is known for its low power consumption for use in the mobile communication fields, consumer electronics (e.g., mobile phones, multimedia players, handheld video game consoles, computers, and computer peripherals). It is also applicable in industries, automotive, aerospace, and other fields.

The ARM processor cores are divided into three families: Cortex-A family, Cortex-R family, and Cortex-M family.

- Cortex-A family

Application Processors, including processors such as Cortex-A5, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A3x, Cortex-A5x, Cortex-A7x, and Cortex-A71x, which are based on ARMv7-A, ARMv8-A, and ARMv9-A architectures. These architectures provide solutions for devices running complex operating systems such as Linux, Android, and iOS.

These processors are widely used in a variety of applications, ranging from low-cost handheld devices to smartphones, tablets, set-top boxes, and enterprise networking equipment. They are capable of handling massive data processing and high-performance computing tasks. Typically, these processors operate at very high clock speeds (generally over 1GHz) and support features like Memory Management Unit (MMU), which are required by operating systems such as Linux, Android, Windows, and mobile operating systems.

- Cortex-R family

Real-time processors, including Cortex-R4, Cortex-R5, Cortex-R7, Cortex-R8, Cortex-R52/R52+ and Cortex-R82 are based on the ARMv7-R and ARMv8-R architectures. The Cortex-R family represents a line of real-time microcontroller cores specifically designed for high-safety and high-performance embedded systems. These processors are intended to offer rapid and deterministic response times, making them particularly suited for applications that require high levels of real-time performance and safety, such as automotive, industrial, and aerospace systems. While these real-time processors generally do not support full versions of operating systems such as Linux and Windows (with the exception of the Cortex-R82), they do support a wide range of Real-Time Operating Systems (RTOS).

- Cortex-M family

Microcontroller Processors are specialized computing components designed for low-power, high-performance, and scalable applications. This category includes processors such as Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, Cortex-M33, Cortex-M55, and Cortex-M85 which are based on the ARMv6-M, ARMv7-M, and ARMv8-M architectures. The Cortex-M family is a series of low-power, high-performance, and extensible processor cores that include many features particularly suited for embedded systems. Their user-friendly nature has greatly contributed to their success across the microcontroller, IoT, and embedded systems markets. The Cortex-M family is widely used in a range of applications from consumer electronics to industrial control systems, including the microcontroller market, IoT, embedded systems, and automotive controllers.

This document takes ARM's Cortex-R52/R52+as the target platform for porting, and its corresponding ARM core architecture is ARMv8-R.

## 1.2 Overview of TriCore Architecture

The TriCore architecture, developed by Infineon, is the foundational technology behind the AURIX series of platforms. The AURIX TriCore platform integrates RISC processor cores, and digital signal processors (DSPs) into a unified MCU solution. These controllers are pivotal in advancing automation, electrification, and connectivity in modern vehicles. Infineon has launched two generations of AURIX products: the TC2xx series and the TC3xx series, referred to as AURIX 1G and AURIX 2G respectively.

- AURIX 1G TC2xx family

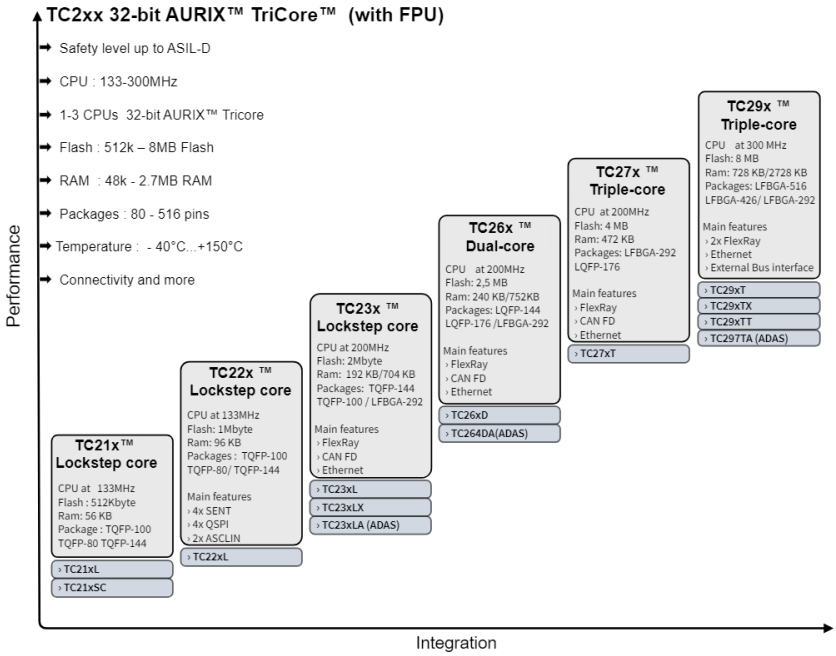



Figure 1.2-1 TC2xx chip family

The AURIX™ TC2xx family is a line of processors based on the single-core and multi-core 32-bit TriCore™ architecture, designed to meet the highest safety standards while offering enhanced performance. The AURIX™ platform is utilized in automotive powertrain systems, including electric and hybrid vehicles, as well as in safety systems such as steering, braking, airbags, and advanced driver assistance systems (ADAS). The core architecture of AURIX™ TC2xx employs the TriCore TC1.6P and TC1.6E architectures. The TriCore TC1.6P core is designed for high-performance applications, offering a maximum clock frequency of up to 300 MHz. Conversely, the TriCore TC1.6E core is optimized for efficient low-power operation, with a maximum clock frequency of up to 200 MHz.

- AURIX 2G TC3xx family

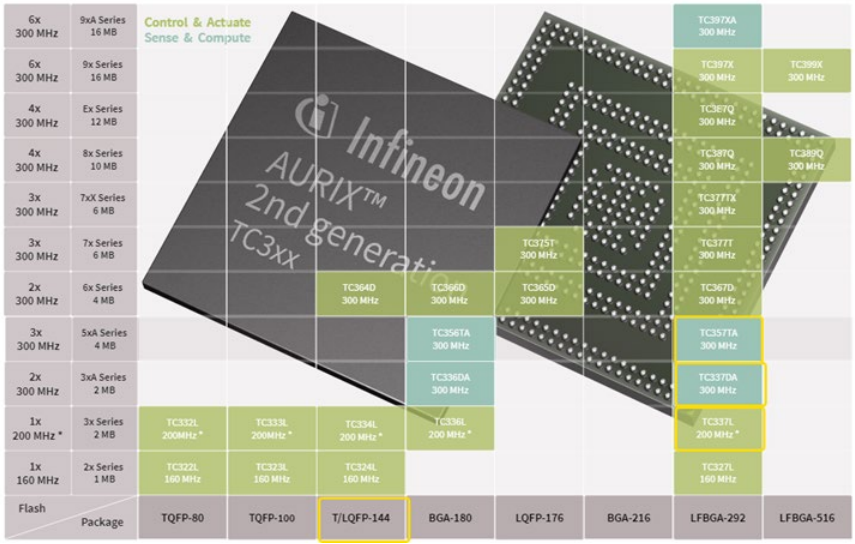



Figure 1.2-2 TC3xx chip family

The AURIX™-TC3xx series combines a high-performance and high-safety architecture, with up to six cores, making it suitable for next-generation autonomous driving domain control and data fusion applications. The AURIX™ TC3xx microcontrollers are also suitable for safety-critical applications such as airbag, braking, and power steering systems, as well as sensor systems for radar, lidar, or camera technologies. The AURIX™ TC3xx cores utilize the TriCore TC1.6.2P architecture. The TriCore TC1.6.2P architecture is similar to the TC1.6P architecture, but but offers enhanced memory access performance and expanded memory protection compared to the TC1.6P architecture.

This document focuses on the AURIX TriCore TC1.6P platform architecture as the target for porting. The AURIX TriCore's TC1.6P platform architecture supports both the TC2xx and TC3xx series. Unless specifically stated otherwise, the content of this document is based on the TriCore's TC1.6P platform architecture as a reference.

### 1.3 Introduction to ARMv8-R Architecture

Early Cortex-R processors, such as the Cortex-R5, were based on the Armv7-R architecture. The Cortex-R52 and Cortex-R52+ processors implement the Armv8-R architecture, designed to address the increasing complexity of automotive real-time software and the transition from discrete dedicated controllers to more centralized and integrated controllers. The Armv8-R architecture adds support that enables better control of software within a single processor, providing code isolation and supporting repeatable and understandable behavior, including virtualization capabilities in real-time processors.

- Early Cortex-R processors, such as the Cortex-R5, were based on the Armv7-R architecture. The Cortex-R52 and Cortex-R52+ processors have implemented the Armv8-R architecture, which aids in addressing the increasing complexity of automotive real-time software and facilitates the transition from discrete specialized controllers to more centralized and combined controllers. The Armv8-R architecture enhances support, enabling better software control within a single processor, providing code isolation, and supporting repeatable and understandable behavior, including virtualization capabilities in real-time processors. It supports both the T32 and A32 instruction sets.
- Provides backward compatibility with the Armv7-R instruction set.
- Enables programs built on Armv7-R to be directly compiled and built in the Armv8-R environment.
- Supports three levels of exception levels EL0/EL1/EL2, and a two-stage Memory Protection Unit (MPU). The EL1 MPU is typically managed by the operating system to achieve isolation between the operating system and applications, as well as between different execution units within an application. The EL2 MPU is programmed by code running at EL2 to achieve isolation control at the virtual machine level.
- Supports GIC V3.0, offering efficient interrupt response and inter-core distribution mechanisms, supporting up to 960 shared interrupts (SPIs) among cores.

## 1.4 TC1.6P Architecture Overview

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

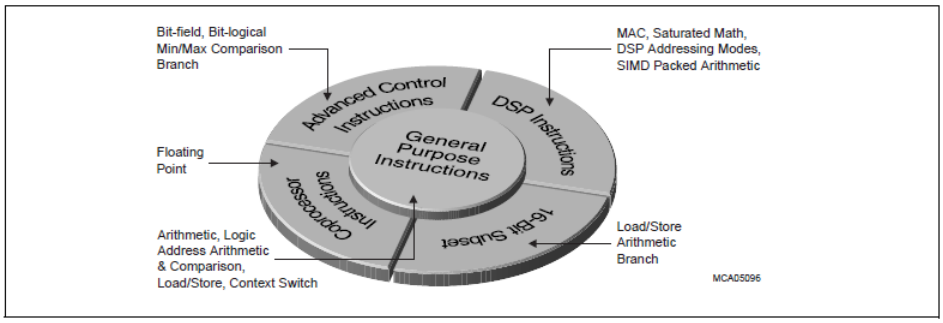



Figure 1.4-1 TriCore Architecture

The ISA supports a unified 32-bit address space, with optional virtual addressing and memory-mapped input/output. The architecture also supports both 16-bit and 32-bit instruction formats. All instructions have a 32-bit format. The 16-bit instructions, a subset of the 32-bit instructions, are optimized for frequently used operations, significantly reducing code space and minimizing memory, system, and power consumption.

Real-time responsiveness is primarily determined by interrupt latency and context-switching time. A high-performance architecture minimizes interrupt latency by avoiding long multi-cycle instructions and providing flexible hardware support for interrupts. The TriCore architecture also supports rapid context switching.

# CHAPTER 2 ARCHITECTURE COMPARISON

This article aims to elucidate the distinctions between the ARM and TriCore platforms by analyzing the following aspects:

- Programming Models

The programming models of the two architectures are introduced, highlighting the differences from the perspective of developers. This mainly includes an analysis of data type formats and operating modes.

- Instruction Sets

The differences in the instruction sets of the two architectures are discussed.

- General-Purpose Registers

The distinctions in the general-purpose registers used in each architecture are explained.

- Exceptions and Interrupts

The mechanisms for handling interrupts and exceptions are compared, including interrupt priorities, interrupt and exception processing methods, and interrupt and exception vector tables.

- Memory Models

The differences in memory models are outlined, covering aspects such as address spaces, addressing modes, and memory protection units.

- Debugging Systems

The differences in the debugging systems available for each architecture are described.

Programming models are the interfaces for developers when developing with microcontrollers. Familiarity with the chip's programming model is essential for improving development efficiency. This section will discuss the differences between the two architectures in terms of the data types supported by the chip, byte order, and the operational modes of the chip.

### 2.1.1 Data Type

The ARM Cortex-R52/R52+ supports data types including: Byte (8 bits), Halfword (16 bits), Word (32 bits) and Doubleword (64 bits). It also supports half-precision, single-precision, and double-precision floating-point data types.

The TriCore architecture supports data types including: Boolean, Bit String, Byte, Signed Fraction, Address, Signed and Unsigned Integers, and IEEE-754 Single-Precision Floating-Point Number.

As a C language developer, the focus is on the data types within the compiler. This document will compare and analyze the commonly used Hightec compiler for TriCore and the Arm Compiler for ARM as examples.

Table 2.1.1-1: Data Types Supported by Hightec and Arm Compiler

| 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 Tips 1: Both compilers have general types that can be directly used when defining variables during porting. Additionally, if users are porting with other compilers, they can compare and analyze in a similar way.

Porting Tips 2: The length of each data type is compiler-dependent, and it is necessary to confirm the length of the data types under a specific compiler during porting. For C language developers, it is recommended to use the 'sizeof' keyword to obtain the length of data types in programming, rather than writing in fixed length values.

Porting Tips 3: Minimize the use of bit fields (Bit types) as it can reduce the portability of the code.

### 2.1.2 Byte order and data alignment

The ARM Cortex-R52/R52+ support either little-endian format (where the least significant byte is stored at the lower memory address) or big-endian format (where the most significant byte is stored at the lower memory address).

- Little-endian format of ARM Cortex-R52/R52+

The storage of a word (4 bytes) and a half-word (2 bytes) in memory is illustrated in the following figure:

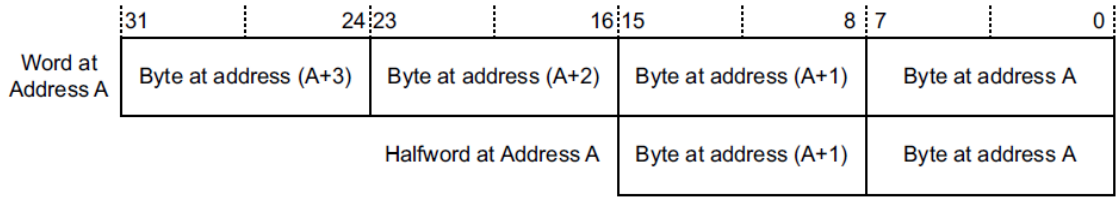



Figure 2.1.2-1 Little-endian format of ARM Cortex-R52/R52+

- Big-endian format of ARM Cortex-R52/R52+

The storage of a word (4 bytes) and a half-word (2 bytes) in memory is illustrated in the following figure:

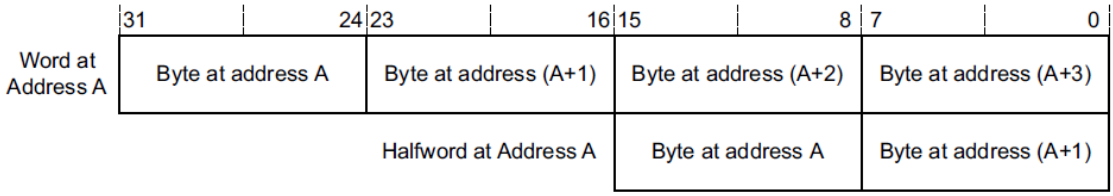



Figure 2.1.2-2 Big-endian format of ARM Cortex-R52/R52+

The byte order of ARM Cortex-R52/R52+ can be selected between little-endian and big-endian modes based on the control input at reset, with little-endian mode being the default. Both TriCore and ARM architectures use 4-byte alignment (Word aligned) for addressability. The data alignment method for both architectures generally aligns data according to its size.

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

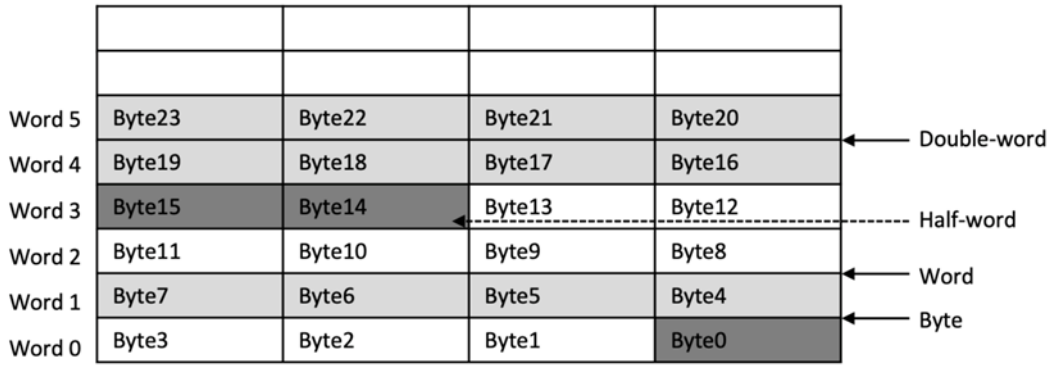



Figure 2.1.2-3 TriCore Byte Ordering

Porting Tip: As a developer, during the porting process, it is necessary to consider the data defined in the program, especially the variables defined using custom struct definitions, and take into account the endianness of the byte sequence. When porting, if the ARM architecture is set to big-endian mode, structures involving byte order should be modified to accommodate big-endian mode.

### 2.1.3 Operating Modes

The ARM Cortex-R52/R52+ can execute in various modes, each associated with an Exception Level (EL). An exception will cause the processor to switch to the corresponding mode. The User mode is associated with Exception Level EL0. The System, FIQ (Fast Interrupt Request), IRQ (Interrupt Request), Supervisor, Abort, and Undefined modes are associated with Exception Level EL1. The Hypervisor mode is associated with Exception Level EL2. After booting, the system defaults to EL2 mode.

The ARM Cortex-R52/R52+ can switch from User mode to EL1 using the Supervisor Call (SVC) instruction. Can switch from EL1 to EL2 using the Hypervisor Call (HVC) instruction. HVC instruction can be only called from EL1.

The TriCore architecture has three levels of I/O Privilege: User-0 mode, User-1 mode, and Supervisor mode.

Table 2.1.3-1 TriCore architecture I/O Privilege Levels.

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

Porting Tip: During the porting process, developers need to consider the differences in permissions between the two architectures when accessing resources that require privileges. This is especially important when porting operating systems, as the restrictions on privileged access must be taken into account.

## 2.2 Instruction Sets

The ARM Cortex-R52/R52+ is compliant with the Armv8-R AArch32 architecture and has two instruction set states:

- A32: Executes 32-bit, word-aligned A32 instructions.
- T32: Executes 16-bit and 32-bit, half-word-aligned T32 instructions.

Both the TriCore and ARM architectures support reduced instruction set sizes of 16-bit and 32-bit.

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

Most TriCore architecture instructions are completed within a single machine cycle.

For a comparison of the TriCore and ARM architecture instruction sets, refer to the following:

Table 2.2-1 Comparison of TriCore and ARM Architecture Instruction Sets

| 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 1: Due to the significant differences in the instruction sets between the two platforms, developer should rewrite assembly code during the porting.

Porting Tip 2: The format of assembly code is related to the compiler. Please refer to the corresponding compiler manual.

## 2.3 General-Purpose Registers

The ARM Cortex-R52/R52+ is based on the Armv8-R architecture. The general-purpose registers of the Armv8-R architecture are illustrated in the figure below:

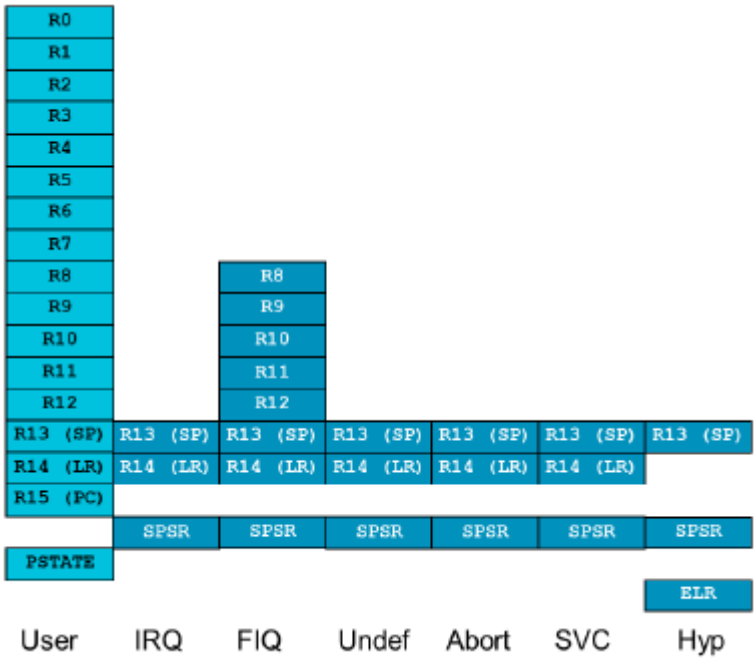



Figure 2.3-1 The general-purpose registers of the Armv8-R architecture

In the Armv8-R architecture, various modes have been introduced, including User, IRQ, FIQ, Undef, Abort, SVC, and Hyp. Registers R0-R7 are general-purpose registers shared across all modes. Except for User mode, all other modes have their banked registers. In IRQ, FIQ, Undef, Abort and SVC mode, R13 (SP) and R14 (LR) registers are banked. In Hyp mode, R13 (SP) is banked, R14 is not banked. R13 holds the stack pointer (SP) and R14 holds the return address (LR). The FIQ mode has its own separate R8-R12 registers.

When the ARM Cortex-R52/R52+ operates, if there is a switch in the current operating mode, SP and LR will use the corresponding R13 and R14 of that mode as their actual running values.

The TriCore architecture's core registers are divided into two categories: General-Purpose Registers (GPRs) and Core Special Function Registers (CSFRs), which include:

- General-Purpose Registers (GPRs):

The General-Purpose Registers (GPRs) consist of 16 general-purpose address registers A[0] to A[15] and 16 general-purpose data registers D[0] to D[15].

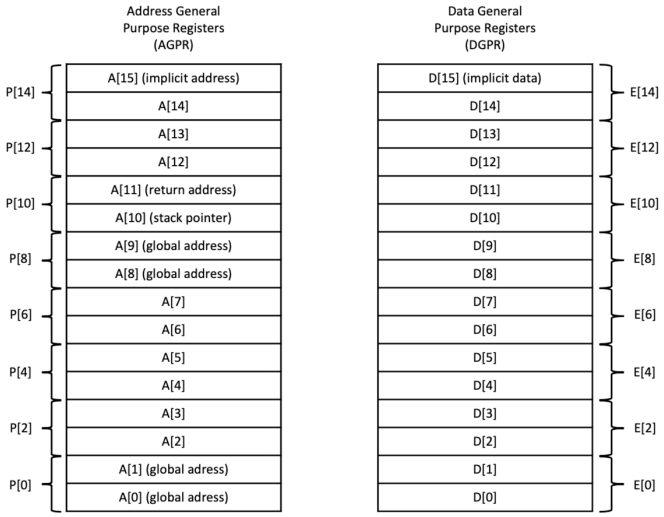



Figure 2.3-2 TriCore GPRs

Four of the general-purpose registers 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: PC(Program Counter)/PSW(Program Status Word)/PCXI( Previous Context Information register). These registers play a key role in task context switching.

Additionally, include: Compatibility Mode Register (COMPAT)、Access Control Registers、Interrupt Registers、Memory Protection Registers、Trap Registers、Memory Configuration Registers、Core Debug Controller Registers、Floating Point Registers.

Porting Tip: Developer need  pay close attention to the differences in registers between the two cores during the porting process. When encountering exceptions, in the TriCore architecture, it is necessary to analyze the TIN stored in register D[15] and the trap entry address recorded in A[11]. It is also important to note the Stack Pointer (SP) register; in the TriCore architecture, check register A[10] to determine the stack location, while in the ARM architecture, check register R13 to determine the stack location.

## 2.4 Exceptions and Interrupts

The interrupt controller for the ARM Cortex-R52/R52+ is based on the ARM GIC architecture. The ARM Cortex-R52/R52+ supports the following types of interrupts:

Table 2.4.1-1: Interrupt Types Supported by ARM Cortex-R52/R52+

| Interrupt Type | Expound | Number of supports |
|---|---|---|
| Private Peripheral Interrupts (PPIs) | These are interrupts that are private to each processor core. | 16 |
| Shared Peripheral Interrupts (SPIs) | These are interrupts generated by peripherals that can be routed to a specific processor core through software configuration. | 960 |
| Software Generated Interrupts (SGIs) | These are interrupts that are triggered by software writing to the Software Generated Interrupt (SGI) generation system register. | 16 |

The ARM Cortex-R52/R52+ features a processor-level GIC Distributor, as well as multiple core-level GIC Redistributors (GIC CPU Interface Per Core). The GIC logic is depicted in the following figure:

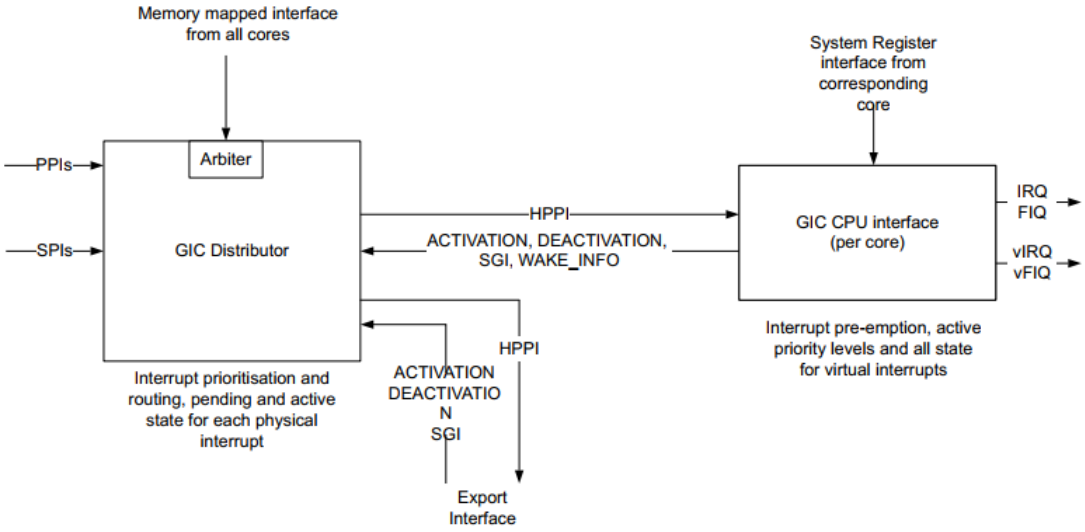



Figure 2.4.1-1 GIC Logic Block Diagram

The interrupt controller of the ARM Cortex-R52/R52+ supports grouping, for example: Group 0 is used to receive FIQ interrupts, while Group 1 is used to receive IRQ interrupts. The interrupt controller registers are memory-mapped, with the physical base address for different processors specified by the CFGPERIPHBASE register.

The ARM Cortex-R52/R52+ has a two-level exception vector table, with separate exception vector tables for EL1 mode and EL2 mode. The structure of the exception vector table is shown in the following table:

Table 2.4.1-2: Interrupt Vector Tables for EL1 and EL2 Modes

| Address Offset | EL1 Vector Table | EL2 Vector Table |
|---|---|---|
| 0x00 | Reset.<br><br>Note that ARM Cortex-R52/R52+ always reset from EL2, do not have a 'real reset' in EL1 | Reset |
| 0x04 | Undefined Instruction | Undefined Instruction (From Hypervisor Mode) |
| 0x08 | Software Interrupt | HVC (from Hypervisor Mode) |
| 0x0C | Prefetch Abort | Prefetch Abort (from Hypervisor Mode) |
| 0x10 | Data Abort | Data Abort (from Hypervisor Mode) |
| 0x14 | Reserved | Hypervisor Trap/Hypervisor mode entry |
| 0x18 | IRQ | IRQ |
| 0x1C | FIQ | FIQ |

The ARM Cortex-R52/R52+ exception model defines three Exception Levels (EL0-EL2), where:

- EL0: The lowest software execution privilege level, which is the user mode;
- EL1: Privileged mode, an enhanced exception level;
- EL2: Privileged mode, provides support for virtualization.

The ARM Cortex-R52/R52+ enters the exception handling process after capturing specific events and returns the program execution process to the state when the exception occurred using the exception return instruction.

The TriCore architecture's interrupt system supports multiple interrupt sources, including on-chip peripheral interrupts and external interrupts. The service providers for interrupt requests can be either the CPU or the DMA. Each interrupt source is assigned a unique interrupt priority. The exceptions (Traps) in the TriCore architecture include non-maskable interrupts (NMI), instruction exceptions, memory management exceptions, or exceptions caused by illegal access. Once an exception occurs, it cannot be masked by software.

The interrupt requests in the TriCore architecture are processed based on interrupt priority and support interrupt nesting, with the following interrupt priority rules:

- High-priority interrupts can preempt the processing of low-priority interrupts.
- Interrupts with the same priority will not interrupt each other.
- The Interrupt Control Unit (ICU) decides which interrupt to handle based on priority arbitration.

All service requests are assigned a Priority Number (SRPN). Each interrupt handler has its own Priority Number. Different interrupt service requests must be assigned different Priority Numbers. There can be up to 255 interrupt priorities, with Priority Number 0 being the lowest interrupt priority. Exceptions have the highest priority and cannot be masked by software.

The TriCore architecture defines 8 types of exceptions. Each type of exception is distinguished by a Trap Identification Number (TIN), and the value of TIN is assigned to the D[15] register before entering the exception service routine. Additionally, exceptions are categorized as synchronous exceptions, asynchronous exceptions, hardware exceptions, software exceptions, and non-recoverable exceptions.

| TIN | Name | Synch. / Asynch. | HW / SW | Definition |
|---|---|---|---|---|
| **Class 0 - MMU** | | | | |
| 0 | VAF | Synch. | HW | Vitrual Address Fill. |
| 1 | VAP | Synch. | HW | Virtual Address Protection. |
| **Class 1 – Internal Protection Traps** | | | | |
| 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 Accress. |
| 6 | MPN | Synch. | HW | Memory Protection Null Address. |
| 7 | GRWP | Synch. | HW | Global Register Write Protection. |
| **Class 2 – Instruction Errors** | | | | |
| 1 | IOPC | Synch. | HW | Illegae 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. |
| **Class 3 – Context Management** | | | | |
| 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 Contex 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 |
|---|---|---|---|---|
| **Class 4 – System Bus and Peripheral Errors** | | | | |
| 1 | PSE | Synch. | HW | Program Fetch Sychronous 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 Intetrity Error. |
| 6 | DIE | ASynch. | HW | Data Memory Integrity Error. |
| 7 | TAE | ASynch. | HW | Temporal Asynchronous Error. |
| **Class 5 – Assertion Traps** | | | | |
| 1 | OVF | Synch. | SW | Arithmetic Overflow. |
| 2 | SOVF | Synch. | SW | Sticky Arithmetic Overflow. |
| **Class 6 – System Call** | | | | |
| 4 | SYS | Synch. | SW | System Call. |
| **Class 7 – Non-Maskable Interrupt** | | | | |
| 0 | NMI | ASynch. | HW | Non-Maskable Interrupt. |

Figure 2.4-2 TriCore Exception Classification

In the TriCore architecture, there are two vector tables: one for interrupts and one for exceptions.

- **Interrupt Vector Table Method**

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

When an interrupt occurs, the CPU calculates the entry point of the corresponding interrupt service function from the contents of the PIPN (Pending Interrupt Priority Number) and BIV registers. There are two vector table spacing options available: 32 bytes or 8 bytes. The vector table spacing is determined by the VSS bit in the BIV register. The specific calculation method 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};

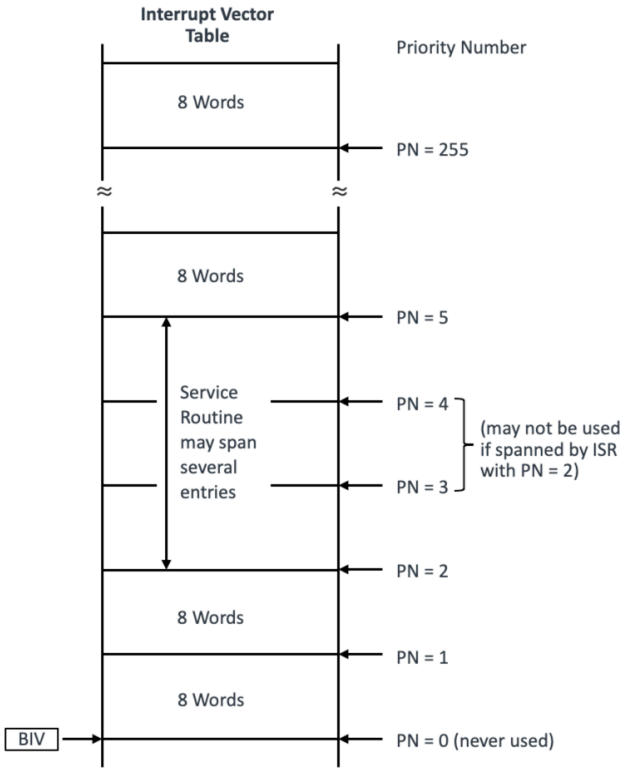



Figure 2.4-3 TriCore Interrupt Vector Table

- **Exception Vector Table**

The base address of the exception vector table is stored in the Base Trap Vector Table Pointer (BTV) register. When an exception occurs, the entry address of the exception handling function is obtained by performing a bitwise OR operation between the Trap Class shifted left by 5 bits and the value of the BTV register. Shifting the Trap Class left by 5 bits creates a 32-byte interval between the entry points of different exception handling functions; therefore, the value in the BTV register must be aligned on at least a 256-byte boundary.

Porting Tip: The ARM Cortex-R52/R52+ defaults to using the exception vector table corresponding to EL2 mode after startup. After the completion of the Reset Handler in the EL2 mode, can configure the EL1 mode exception vector table and jump to the Reset Handler of the EL1. Note that ARM Cortex-R52/R52+ always reset from EL2, do not have a 'real reset' in EL1. An example reference process is as follows:

- Example of EL2 mode exception vector table:

```
EL2_Vectors:
        LDR PC, EL2_Reset_Addr
        LDR PC, EL2_Undefined_Addr
        LDR PC, EL2_HVC_Addr
        LDR PC, EL2_Prefetch_Addr
        LDR PC, EL2_Abort_Addr
        LDR PC, EL2_HypModeEntry_Addr
        LDR PC, EL2_IRQ_Addr
        LDR PC, EL2_FIQ_Addr
```

Figure 2.4-4 ARM Cortex-R52/R52+ EL2 Mode exception Vector Table

- EL2 Mode Reset Handler Example:

```
EL2_Reset_Handler:
        LDR r0, =EL2_Vectors
        MCR p15, 4, r0, c12, c0, 0      //  Write to HVBAR

        // Init HSCTLR
        LDR r0, =0x30C5180C             // See TRM for decodi
        MCR p15, 4, r0, c1, c0, 0       // Write to HSCTLR

        // Enable EL1 access to all IMP DEF registers
        LDR r0, =0x7F81
        MCR p15, 4, r0, c1, c0, 1       // Write to HACTLR

        // Change EL1 exception base address
        LDR r0, =EL1_Vectors
        MCR p15, 0, r0, c12, c0, 0      // Write to VBAR

        // Go to SVC mode
        MRS r0, cpsr
        MOV r1, #Mode_SVC
        BFI r0, r1, #0, #5
#ifdef __THUMB__
        ORR r0, r0, #(0x1 << 5)         // Set T bit
#endif
        MSR spsr_cxsf, r0
        LDR r0, =EL1_Reset_Handler
        MSR elr_hyp, r0
        DSB
        ISB
        ERET
```

Figure 2.4-5 ARM Cortex-R52/R52+ EL2 Mode Reset Handler Example

To switch to the EL1 mode exception vector table, set the base address of the EL1 mode exception vector table (EL1_Vectors) to the VBAR register, and then use the ERET instruction to jump to the Reset Handler of the EL1 mode.

- EL1 Mode Exception Vector Table Example:

```
EL1_Vectors:
        LDR PC, EL1_Reset_Addr
        LDR PC, EL1_Undefined_Addr
        LDR PC, EL1_SVC_Addr
        LDR PC, EL1_Prefetch_Addr
        LDR PC, EL1_Abort_Addr
        LDR PC, EL1_Reserved
        LDR PC, EL1_IRQ_Addr
        LDR PC, EL1_FIQ_Addr
```

Figure 2.4-6 ARM Cortex-R52/R52+ EL1 Mode Exception Vector Table

## 2.5 Memory Models

The Armv8-R AArch32 architecture used by the ARM Cortex-R52/R52+ defines the PMSAv8 memory model. Memory access permissions and attributes are determined by the Memory Protection Unit (MPU). In the ARM Cortex-R52/R52+, the physical address is always identical to the virtual address. The default memory view of Armv8-R is illustrated in the following figure:

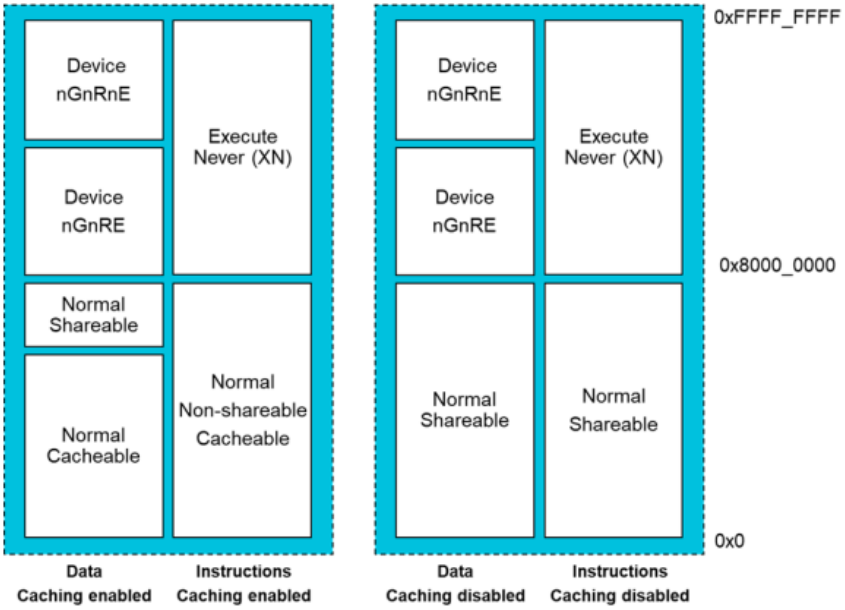



Figure 2.5.1-1 ARM Cortex-R52/R52+ Memory View

Armv8-R includes the following types of memory:

- Normal memory

Normal memory is a conventional memory area suitable for various types of memory storage, such as ROM, RAM, Flash, and SDRAM. This area allows programs to read from and write to it.

- Device memory

Device memory is designed for peripheral and I/O access. This area does not support caching but still allows data to be read and written through buffering.

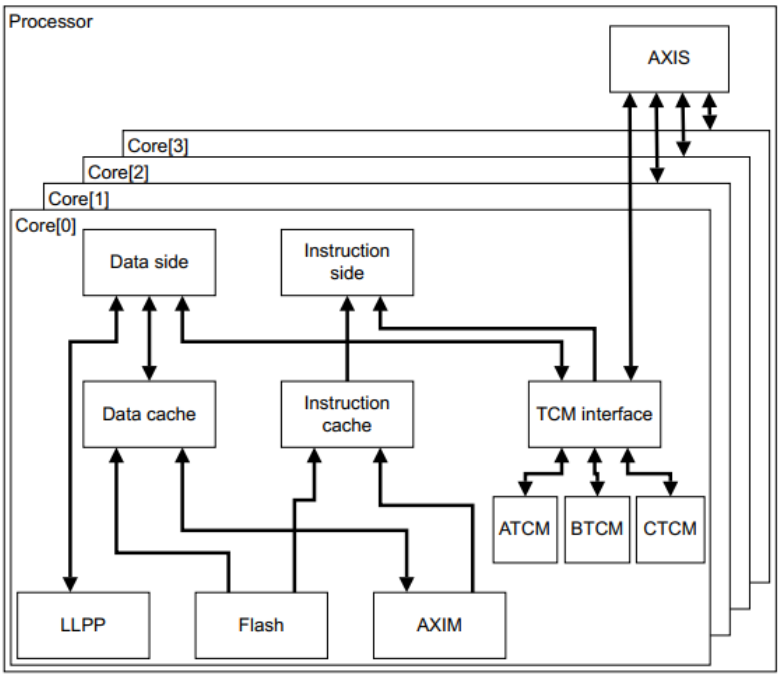



Figure 2.5.1-2 ARM Cortex-R52/R52+ Memory System Block Diagram

The ARM Cortex R52/R52+ features a multi-level memory system. Each core has its own independent data cache and instruction cache. Additionally, each core also has its own TCM (Tightly Coupled Memory), which can be used by software for fast data read/write and instruction execution. The AXIM interface is the primary interface for accessing external memory. The Flash interface is designated for accessing external read-only memory, such as Flash. The LLPP interface is for accessing peripherals and specific external memories.

The ARM Cortex-R52/R52+ provides two programmable Memory Protection Units (MPUs), one for EL1 mode and one for EL2 mode. Each MPU can cover a 4GB address space and is configured with elements including start address, end address, access permissions, and memory attributes.

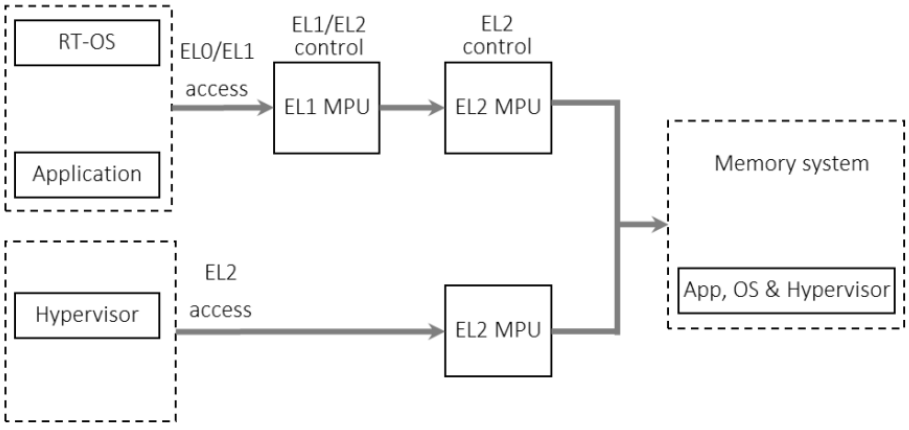



Figure 2.5-3 ARM Cortex R52/R52+ MPU Block Diagram

ARM Cortex R52/R52+ supports virtualization. Applications and RTOS access the EL1 MPU to achieve isolation between applications and between applications and the RTOS. The virtual machine controls the EL2 MPU to achieve isolation between virtual machines. The ARM Cortex R52/R52+ MPU uses 2 bits to control access permissions, as shown in the table below:

Table 2.5-1: Access Permission Control of the ARM Cortex R52/R52+ MPU

| Permission Bit Position | EL0&EL1 | EL2 |
|---|---|---|
| 00 | No access | Read/write |
| 01 | Read/write | Read/write |
| 10 | No access | Read-only |
| 11 | Read-only | Read-only |

The TriCore architecture uses a 32-bit address width, providing access to a maximum addressable range of 4GB. The address space is divided into 16 memory segments [0H - FH], each segment being 256MB in size. Each segment can serve as peripheral space, cached space, or non-cached space.

The physical memory attributes of the segments [0H - 7H] depend on the specific implementation requirements. If the Memory Management Unit (MMU) is enabled, segments [0H - 7H] are considered virtual addresses and must be translated upon access. If the MMU is not used, the access characteristics depend on the specific implementation requirements, and illegal access may result in an exception.

The TriCore architecture's SRAM (Scratchpad RAM) supports program segments and data segments, located 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 achieve distinct memory non-access through the access to mirror areas of DSPR and PSPR, with these mirror areas distributed within the segments [0H - 7H].

| Segment | Properties |
|---|---|
| $D_H$ | DSPR region |
| $C_H$ | PSPR region |
| $7_H$ | CPU-0 PSPR and DSPR memory image region |
| $6_H$ | CPU-1 PSPR and DSPR memory image region |
| $5_H$ | CPU-2 PSPR and DSPR memory image region |
| $4_H$ | CPU-3 PSPR and DSPR memory image region |
| $3_H$ | CPU-4 PSPR and DSPR memory image region |
| $2_H$ | CPU-5 PSPR and DSPR memory image region |
| $1_H$ | CPU-6 PSPR and DSPR memory image region |
| $0_H$ | CPU-7 PSPR and DSPR memory image region |

Figure 2.5-4 TriCore SRAM Memory Segment.

The memory of the segments [8H - DH] is used for defining non-volatile storage spaces and special storage areas, such as the program flash memory (PFLASH), data flash memory (DFLASH), on-chip firmware BROM, and local memory unit (LMU). Additionally, the memory in segments [8H - 9H] allows cached access, while the memory in segments [AH - DH] does not allow cached access.

The memory of segments [EH - FH] is used to define the access areas for peripherals, with the on-chip peripheral memory areas, such as peripheral registers, allocated to this segment.

The TriCore architecture's addressing mode accesses memory data through load and store instructions, with data access widths of 8-bit, 16-bit, 32-bit, or 64-bit. The TriCore architecture supports seven addressing modes, including:

Table 2.5-2: The Seven Addressing Modes Supported by the TriCore Architecture

| Addressing Modes. | Description |
|---|---|
| Absolute Addressing | This is generally used for accessing peripheral registers and global data. In absolute addressing, the 18-bit constant specified in the instruction is used as the memory address. The complete 32-bit address is formed by shifting the high 4 bits of the 18-bit constant to the high 4 bits of the 32-bit address, with the remaining bits filled with 0. |
| Base + Short Offset | The effective address in this mode is the sum of the base address register and the sign-extended 10-bit offset. |
| Base + Long Offset | Compared to Base + Short Offset, the offset is a 16-bit sign-extended value, allowing any location in memory to be addressed with a two-instruction sequence |

| | |
|---|---|
| Pre-increment | Typically used for stack operations when pushing data onto the stack. This addressing mode uses the sum of the address register and the offset as the effective address and writes the result back to the address register. |
| Post-increment | Typically used for stack operations when popping data off the stack. This addressing mode uses the value in the 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 | Generally used for accessing data values in a circular buffer during filter computations. |
| Bit-reverse | Used for calculations in Fast Fourier Transform (FFT) algorithms. |

The TriCore architecture supports caching and features both data and instruction caches. If the instruction cache is enabled, the CPU can perform speculative processing when fetching instructions from memory. Similarly, if the data cache is enabled, the CPU can also perform speculative processing when retrieving data from memory. For the TriCore architecture's addressable space, the area available for caching is within the segments [8H - 9H], which is a limited range, as shown in the following figure:

| **Segment** | **Attributes** |
|---|---|
| $F_H$ | Peripheral Space. |
| $E_H$ | Peripheral Space. |
| $D_H$ | Non-cacheable Memory. |
| $C_H$ | Non-cacheable Memory. |
| $B_H$ | Non-cacheable Memory. |
| $A_H$ | Non-cacheable Memory. |
| $9_H$ | Cacheable Memory. |
| $8_H$ | Cacheable Memory. |
| $7_H$ - $0_H$ | Non-cacheable Memory. |

Figure 2.5-5 TriCore Architecture Cache Area

Additionally, the caching in the TriCore architecture has the following limitations:

- The address space of peripherals cannot be cached.
- Data from the local DSPR (Data Scratch Pad RAM) cannot be stored in the local data cache.
- Data from the local PSPR (Program Scratch Pad RAM) cannot be stored in the local instruction cache.

Porting Tip 1: For developers, while caching can enhance the performance of data access, it can also introduce issues, such as data consistency problems. A typical issue arises when a DMA (Direct Memory Access) operation reads memory from the CPU's data cache. If the CPU writes new data to the data cache but DMA reads the old data still stored in the cache, data consistency problems can occur. To avoid such issues, developers need to use caches cautiously.

Porting Tip 2: For developers, there may be significant differences in addressing modes between the two architectures. Generally, addressing modes are not a major concern in typical development scenarios. However, in special application scenarios, such as those with high performance requirements, different addressing modes and instruction sets may be utilized.

## 2.6 Debugging Systems

The ARM Cortex R52/R52+ supports debugging through JTAG (Joint Test Action Group) or SWD (Serial Wire Debug) protocols. Debugging can be facilitated by connecting a debug device, such as the ARM DSTREAM, to the target device.

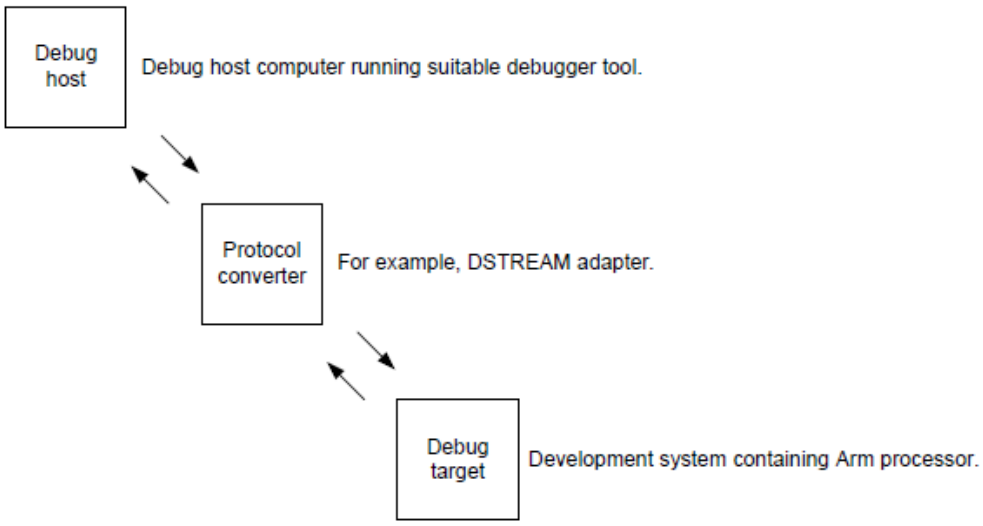



Figure 2.6-1 ARM Cortex R52/R52+ Debug Model

The TriCore architecture's debugging system is implemented through the Core Debug Controller (CDC), which facilitates core debugging and allows access to the core and the chip's memory space. The CDC primarily supports the software development environment by providing real-time control over the core's execution and restart, access to and updating of internal registers and memory data, and setting complex breakpoint and watchpoint conditions.

Both the TriCore and ARM architectures' debugging systems support the standard JTAG interface and also feature trace functionality. The TriCore architecture additionally supports a two-wire Device Access Port (DAP), which offers higher debugging speeds and requires fewer pins compared to JTAG. ARM architecture, on the other hand, supports a two-wire debugging interface known as Serial Wire Debug (SWD). Furthermore, ARM architecture supports the proprietary CoreSight feature, which provides additional debugging and tracing capabilities. CoreSight allows for the debugging of the entire System on Chip (SoC), offering a comprehensive solution for system-wide debug and trace needs.

# CHAPTER 3 FUNCTIONAL SAFETY DESIGN COMPARISON.

The TriCore architecture and the ARM Cortex-R series architectures are widely used in the automotive field. With the development of the automotive industry, the requirements for the safety and stability of automotive systems are becoming increasingly stringent. The functional safety standard ISO 26262 has also set higher requirements for automotive systems. In the ISO 26262 standard, different ASIL / SIL levels based on risk assessment analysis are proposed, along with specific target indicators that need to be achieved. ASIL D represents the highest level of potential risk and requires the strictest methods for fault management. To meet functional safety requirements, chips supporting the TriCore architecture and ARM architecture have proposed their respective solutions for functional safety. This section introduces the specific chip series, with the TriCore architecture focusing on the TC3xx series chips and the ARM architecture focusing on the Cortex-R52/R52+ series chips.

## 3.1 Core Safety Mechanisms.

The core safety of the TC3xx chip is implemented through a core lockstep mechanism. This is achieved by using a master core and a checker core to perform lockstep functionality. While the master core carries out logical processing, the checker core also processes the inputs of the master core. After both cores have completed processing, a logic comparator compares the results from both cores to verify consistency. To prevent common cause failures, there is a delay of two clock cycles in the input to the checker core.

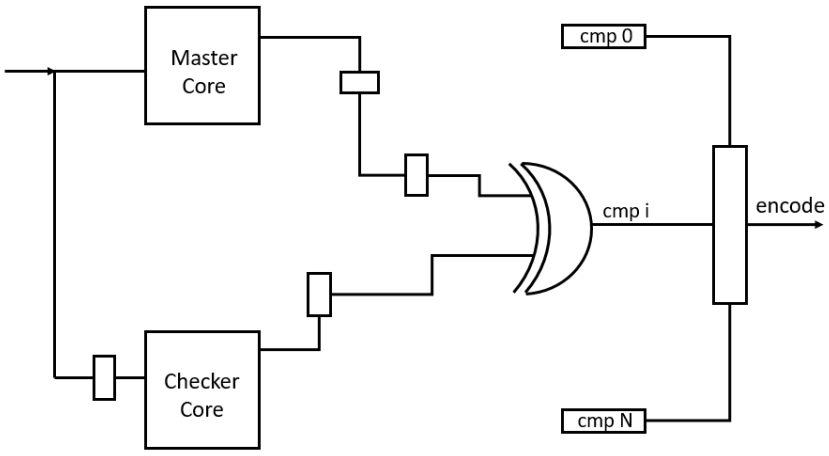



Figure 3.1-1 TC3xx Core Comparator

The safety mechanisms of the ARM Cortex-R52/R52+ depend on the specific implementation by the chip manufacturer, and you can refer to the corresponding chip manual.

## 3.2 Memory Protection Unit (MPU)

In addition to core safety, the ISO 26262 standard introduces the concept of Freedom from Interference (FFI), which requires that data exchanges between modules of different ASIL levels be isolated in memory space to prevent lower ASIL level modules from affecting higher ASIL level modules. Therefore, Memory Protection Unit (MPU) functionality is an essential safety mechanism for chips.

The ARM Cortex R52/R52+ provides two programmable MPUs. The EL1 mode and EL2 mode each correspond to their respective MPUs. Each MPU can cover a 4G address space. Each Memory Protection Region includes the following configuration elements: start address, end address, access permissions, and memory attributes.

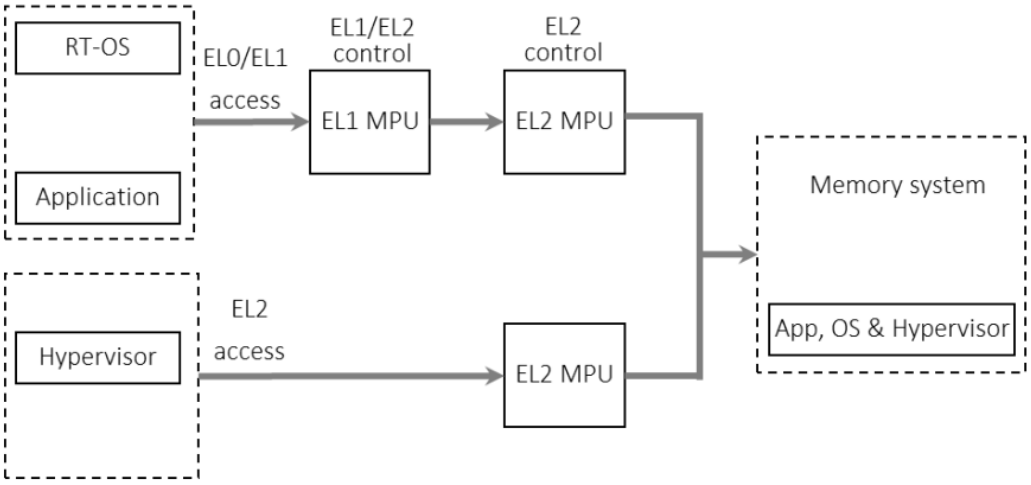



Figure 3.2-1 ARM Cortex R52/R52+ MPU Block Diagram

The ARM Cortex R52/R52+ supports virtualization. Applications and the RTOS access the EL1 MPU, achieving isolation between applications and between applications and the RTOS. The virtual machine controls the EL2 MPU, achieving isolation between virtual machines. The ARM Cortex R52/R52+ MPU uses 2 bits to control access permissions, as shown in the Table 2.5-1.

The memory protection in the TC3xx chip is based on address-range-based memory protection, which implements protection for both the program area and the data area. The TriCore architecture's memory protection supports up to six protection sets, with a maximum of 18 data protection regions and up to 10 program protection regions.

The TC3xx chip also supports bus-level (Bus MPU) memory protection functions. Compared to core-level memory protection, bus-level memory protection provides access restrictions for bus masters to the memory of bus slaves.

Porting Tip: For developers, memory protection features are generally implemented by the operating system, which configures memory protection through context switching. When porting an operating system, the porting of memory protection features should take into account the differences between the two chip architectures.

# CHAPTER 4 SOFTWARE DEVELOPMENT PORTING.

## 4.1 Toolchain.

Embedded software development and runtime environments are typically located on different platforms. The development environment is generally deployed on Windows/Linux computers, while the runtime environment is deployed on the target chip. The compilation and building of runtime environment programs are achieved through a cross-compilation environment. The toolchain for embedded development environments usually includes a compiler, assembler, linker, debugger, etc. With the development of toolchains, editing software, compilation software, assembly software, linking software, debugging software, and functional libraries have all been integrated into one environment, known as the Integrated Development Environment (IDE).

The ARM platform toolchain includes Arm Development Studio and Green Hills MULTI Integrated Development Environment. The development environment for the TriCore architecture mainly includes Tasking, HighTec, and debuggers such as Infineon's miniwiggler or professional debuggers like Lautebach, ISYSTEM, etc.

For developers, when porting software from the TriCore platform to the ARM platform, it is necessary to consider the differences in various compilation environments. The analysis should mainly be conducted from the following aspects:

- Assembly Code

Typically, for performance reasons, the startup code and exception handling routines of a program are written in assembly language. When porting, it is necessary to consider the differences in assembly instructions across platforms, as referenced in section on 2.2 Instruction Sets. Integrated development environments usually provide examples for different platforms, and you can replace the startup code and exception handling routines for the porting platform with the corresponding routines for the target platform.

- Data Types

Different compilers support different data types for different platforms. For specifics, refer to the data type definitions of the compilers, which can be found in the section on Data Type.

- Compilation and Linking Options

During the porting process, it is important to consider the different compiler options. This includes the different standards of the C language supported by the compiler (such as C90/C99), support for different ANSI standards, and optimization options for compilation or

linking. Different compilation and linking options can affect the execution of the generated executable files. For the Arm Compiler, the settings for key compilation options such as `march` are shown in the following table:

Table 4.1-1: Key Compilation Options for Arm Compiler

| Serial Number | Compilation Option | Compilation Option Value |
|---|---|---|
| 0 | -march | armv8-r |
| 1 | -mcpu | cortex-r52 |
| 2 | -mfpu | neon-fp-armv8 (if neon is present) |
| 3 | -mfloat-abi | hard |

- Linker Scripts

During the compiler's linking process, linker scripts are used to allocate the program to different address spaces, such as RAM, FLASH, stack areas, etc. The formats of linker scripts vary among different compilers. Generally, integrated development environments provide users with linker script templates for different platforms. During the porting process, users need to replace the code partitioning in the linker script with the corresponding partitions for the target platform.

## 4.2 Development Toolchain

When a chip is powered on, it is necessary to initialize the internal registers and other components of the chip to establish the necessary runtime environment before executing the user-developed applications. Once the chip initialization is complete, the application can be executed.

The startup process of the ARM Cortex-R52/R52+ begins at the reset vector, typically starting in EL2 mode and executing the Reset Handler of the EL2 mode interrupt vector table. If virtualization is not used, the Reset Handler in EL2 mode can set up the interrupt vector table for EL1 mode and jump to the Reset Handler of EL1 mode. For more details, you can refer to the section on Exceptions and Interrupts in 2.4 Exceptions and Interrupts.

In the Reset Handler, the following initialization processes are typically carried out:

- Configure the caching mechanism to enable or disable data cache and instruction cache functions.
- Initialize CPU core registers.
- Initialize the stack pointer.
- Initialize modules such as the MPU (Memory Protection Unit).
- Initialize global variables in the .bss and .data sections.
- Initialize peripheral modules like clocks.
- Proceed to the entry point of the user application, such as the main function.

The ARM Cortex-R52/R52+ supports multiple Clusters (clusters of cores). A Cluster is a collection of related processor cores. There are typically two methods for synchronizing cores within the same Cluster during startup:

Method 1:

The hardware boots the primary core after reset, while all other secondary cores are held in reset by the hardware. The primary core completes the boot process and then brings the secondary cores out of reset.

Method 2:

All cores in the Cluster exit reset, and the secondary cores are paused by software, waiting for the primary core to signal them to continue running.

As shown in the following figure:

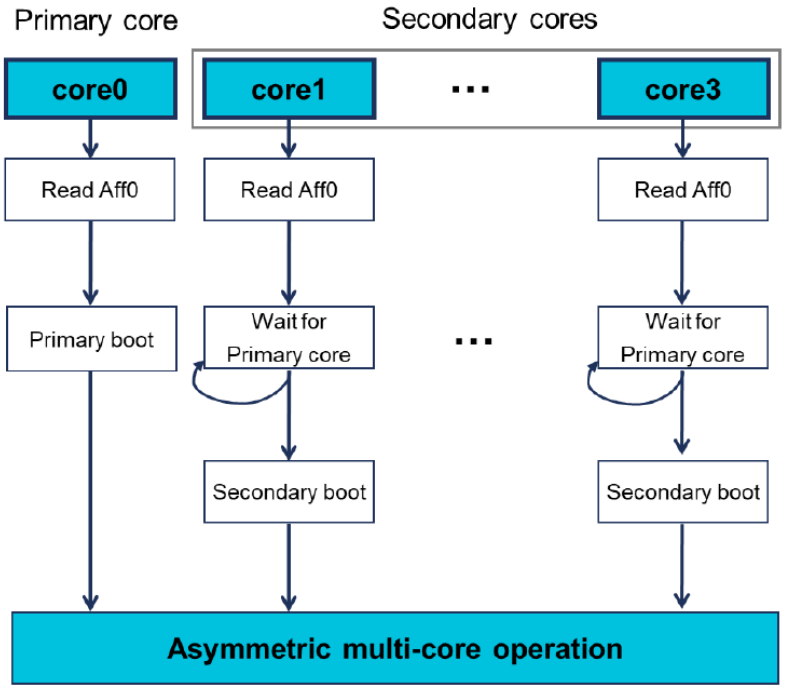



Figure 4.2-1 ARM Cortex R52/R52+ Cluster Boot Process.

Porting Tip: During the porting process, developers should account for differences in the boot process between various architectures. Generally, integrated development environments provide users with boot code examples for different platforms. In most cases, most configurations do not require changes by the user; simply replace them with the target platform's boot code. However, some special modules may need to be configured, such as the MPU module, which needs to be adapted according to the original requirements. The values for stack pointers need to be configured based on the linkage script, and the initialization of global variables needs to be configured according to the partitions in the linkage script.

### 4.3 Exceptions and Interrupt Handling.

For information on interrupts and exceptions in the ARM Cortex-R52/R52+, please refer to section on 2.4 Exceptions and Interrupts.

In the ARM Cortex-R52/R52+, interrupts and exceptions share a single interrupt vector table. The following mainly introduces the process of handling exceptions:

1、The processor state is automatically saved to the SPSR register and the current processor state is updated. If in EL1 mode, the link register (LR) is modified for subsequent return from the interrupt. If in EL2 mode, the register ELR will be used.

2、Execution begins from the corresponding entry in the interrupt vector table. This entry is a jump instruction that branches to the top-level handler.

3、The context is saved, and then the second-level exception handler is called.

4、The second-level exception handler typically executes in a C language environment. After execution, it returns to the top-level handler.

5、The top-level handler restores the context and then returns to the address saved in the link register (LR).

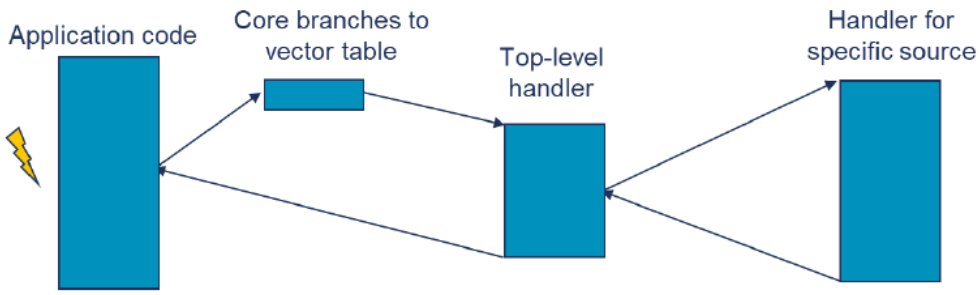



Figure 4.3-1 ARM Cortex R52/R52+ Exception Handling Process.

Porting Tip: For developers, it is necessary to compare the differences in the handling of exceptions and interrupts between the two architectures. Due to these differences, the exception and interrupt handling processes of the two architectures are not reusable. Generally, integrated development environments provide users with boot code examples for different platforms. In these examples, they usually include the processes for exceptions and interrupts, as well as the default interrupt vector table.

### 4.4 Peripheral Access

The peripherals of the ARM Cortex-R52/R52+ are dependent on the implementation provided by different chip manufacturers. Typically, these manufacturers supply peripheral drivers for user implementation. During the porting process, development should be based on the peripheral drivers provided by the chip manufacturer.

### 4.5 Memory Protection Unit

The ARM Cortex R52/R52+ provides two programmable MPUs (Memory Protection Units). The EL1 mode and EL2 mode each correspond to their respective MPUs. Each MPU can cover a 4G address space. Each memory protection region includes the following configuration elements: start address, end address, access permissions, and memory attributes.

An example of the MPU initialization process for the ARM Cortex R52/R52+ platform is as follows:

```
LDM r5!, {r1-r4}    // r5 points to MPU configuration
MCR (H)PRBAR0, r1   // write (H)PRBAR0
MCR (H)PRLAR0, r2   // write (H)PRLAR0
MCR (H)PRBAR1, r3   // write (H)PRBAR1
MCR (H)PRLAR1, r4   // write (H)PRLAR1
… (8 times)
LDM r5!, {r1-r2}    // load (H)MAIR0/1 setting
MCR (H)MAIR0, r1    // write (H)MAIR0
MCR (H)MAIR1, r2    // write (H)MAIR1
```

Figure 4.5-1 ARM Cortex-R52/R52+ MPU Configuration Example

Porting Tip 1: The MPU helps ensure memory isolation between subsystems, enhancing the system's reliability and security. For developers, it is necessary to compare the differences in MPU enabling and configuration between the two architectures.

Porting Tip 2: The export of memory addresses is related to the linker script, and the syntax of linker scripts may vary between different toolchains. For developers, it is necessary to replace the original linker script according to the toolchain being used.

## 4.6 Performance Optimization Suggestions

Performance optimization on the ARM Cortex-R52/R52+ platform primarily focuses on three aspects: Cache, Tightly Coupled Memory (TCM), and fast interrupts (FIQ). Additionally, utilizing the compilation optimization options provided by the ARM platform toolchain can effectively improve software performance.

- Cache

The ARM Cortex R52/R52+ platform has a multi-level memory system. Each core has its own independent data cache and instruction cache. Utilizing these caches can significantly reduce the execution cycle for software accessing data and instructions.

An example of enabling Cache on the ARM Cortex R52/R52+ platform is as follows:

```
MRC     p15, 0, r0, c1, c0, 0     // read System Control Register
ORR     r0, r0, #(0x1 << 12)      // enable I Cache
ORR     r0, r0, #(0x1 << 2)       // enable D Cache
ORR     r0, r0, #0x1              // enable MPU
MCR     p15, 0, r0, c1, c0, 0     // write System Control Register
```

Figure 4.6-1 ARM Cortex-R52/R52+ Cache Enablement Example

- TCM

TCM is characterized by its close coupling with the core, which means that accessing TCM is typically faster than accessing RAM. TCM is often used to store critical code and data that require high-speed access or have low latency requirements.

ARM Cortex R52/R52+ has three TCMs: ATCM, BTCM and CTCM. All of them can be used to store instructions or data. TCM is controlled through CP15 registers, which manage the enablement status and size parameters of TCM.

The size of each TCM on the ARM Cortex R52/R52+ platform can be independently configured as 0KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, or 1MB. TCM is an optional implementation in the processor, with 0KB indicating that TCM is not implemented on that processor.

- FIQ

FIQ is a special type of interrupt mechanism in the ARM Cortex R52/R52+ platform. The FIQ on the ARM Cortex R52/R52+ platform has a high priority, enabling it to quickly respond to and handle urgent interrupt events. Its main features include:

1、Rapid Response: FIQ can preempt processor resources more swiftly than IRQ, thus reducing interrupt latency.

2、Dedicated Registers: The ARM Cortex R52/R52+ platform provides a set of dedicated registers for FIQ (refer to section 2.3 General-Purpose Registers), minimizing the overhead of saving and restoring context during interrupt handling, thereby shortening response time.

During the migration process, if critical events like emergency data from sensors or system failures occur, FIQ enables rapid response, ensuring system stability and reliability.

- Compilation Optimization Options

Compilation optimization options are settings used during the compilation of programs to improve code performance, reduce code size, or enhance other characteristics.

On the ARM Cortex R52/R52+ platform, taking the Arm Compiler as an example, common compilation optimization options include:

1、Code Inlining: Embeds the code of small functions directly into the place where they are called, reducing the overhead of function calls. For instance, if a frequently called and short function is optimized with inlining, it can significantly improve execution efficiency.

2、Loop Optimization: Includes loop unrolling, which duplicates the loop body multiple times to reduce the overhead of loop control. For example, in simple computation-intensive loops, loop unrolling can fully utilize the processor's pipeline to enhance performance.

3、Constant Propagation: Directly replaces expressions that use known constant values with those constants in the code. If there is a constant that is used multiple times in the code, constant propagation can reduce redundant calculations.

Compilation optimization options are related to the toolchain being used. During migration, corresponding compilation optimization options can be set according to the manual of the toolchain to improve program performance.

# CHAPTER 5 SUMMARY

This document primarily explores the differences between the Tricore and ARM Cortex-R52/R52+ platforms, focusing on architecture, functional safety design, and software development porting. It provides porting recommendations for developers during the migration process.

Additionally, the software development porting section includes performance optimization suggestions for developers' reference.

# CHAPTER 6 APPENDIX

## 6.1 References

| ID | References | Version | Date |
|---|---|---|---|
| 1 | Arm Cortex-R52 Processor Technical Reference Manual | Revision: r1p4 | 20 July 2021 |
| 2 | Arm Cortex-R52+ Processor Technical Reference Manual | Revision: r0p1 | 25 August 2022 |
| 3 | Arm Architecture Reference Manual Supplement: Armv8, for the Armv8-R AArch32 architecture profile | Updated EAC release | 06 November 2020 |
| 4 | Arm Architecture Reference Manual for A-profile architecture | 0487K.a | 20 March 2024 |
| 5 | Infineon-AURIX_TC3xx_UserManual | V02_00 | / |
| 6 | Infineon-AURIX_TC3xx_Architecture_UserManual | V01_00 | / |

## 6.2 Terms and Acronyms

| Term/Acronym | Description |
|---|---|
| CPU | Central Processing Unit |
| MCU | Microcontroller Uni |
| MMU | Memory Management Unit |
| IDE | Integrated Development Environment |
| RISC | Reduced Instruction Set Computer |
| ISA | Instruction Set Architecture |
| RTOS | Real Time Operating System |
| DSP | Digital Signal Processing |
| DMA | Direct Memory Access |

| | |
|---|---|
| MPU | Memory Protection Unit |
| ASIL | Automotive Safety Integrity Level |
| IDE | Integrated Development Environment |
| CSA | Context Save Areas |