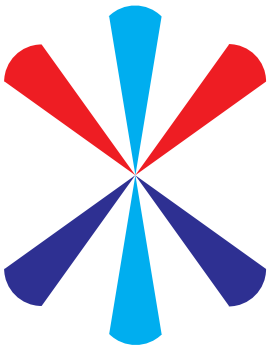

Calterah Alps Radar Firmware Reference Manual

Release 1.1

Calterah Semiconductor

Feb 28, 2020

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER

ONE

OVERVIEW

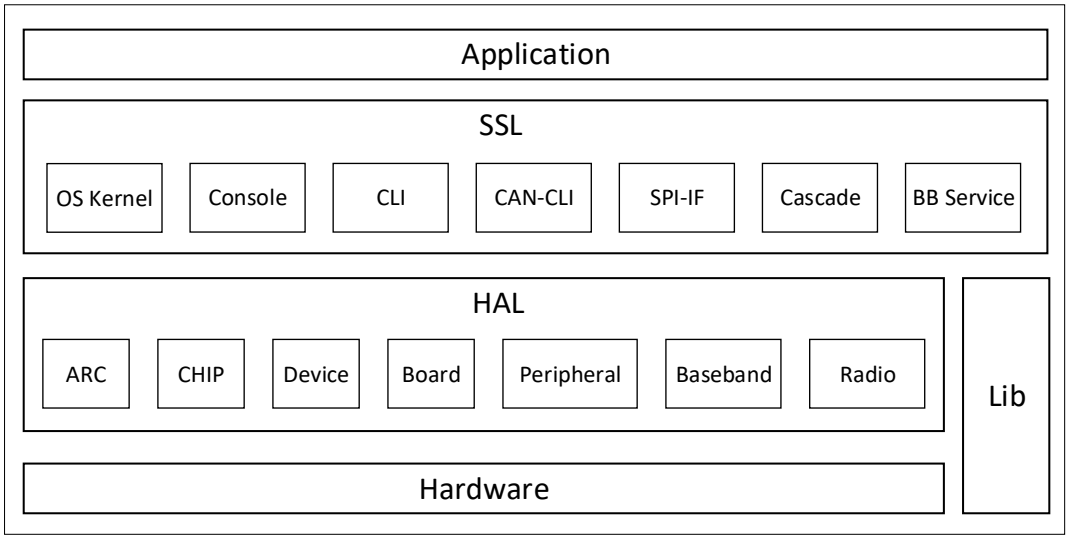

Fig. 1.1: Radar System Hierarchical Structure

Fig. 1.1 illustrates the hierarchical structure of the system.

Above the hardware layer, software is split into three layers.

- The lowest layer is hardware abstract layer (HAL), which includes ARC, chip, device, board, peripherals, baseband, and radio. HAL mainly controls the hardware and access, and offers hardware-independent APIs to the upper layers.
- The middle layer is system service layer (SSL), which includes OS Kernel, console, CLI, CAN-CLI, SOI-IF, cascade, and BBS service. SSL mainly manages data transfer, communication protocols, and data process. The OS kernel module in the SSL offers some basic components for other system service, modules, and application layers, such as MUTEX, semaphore, and queue.
- The top layer is Application, from which users can choose specific radar applications depending on different scenarios. A Kalman filtering demo source code is provided in Calterah software release package to output basic information, including the range, velocity, angle, power, and etc. For more demo code of user cases, contact Calterah field application engineers.

1.1 Build System

The build system of Alps Radar Firmware is based on the make tool.

Alps Radar Firmware supports two different toolchains offered by Synopsys, GNU tool and MetaWare tool. The default toolchain of Alps Radar Firmware is the GNU tool.

- The GNU tool is maintained by Synopsys but free. Everyone can download it from its official website.
- The MetaWare tool is chargeable and also maintained by Synopsys. It can be used for developing safety-related software to meet ISO 26262 requirements, with certified versions of Development Toolkit and Developer Compiler.

The toolchains firstly compile the radar system into different libraries, and then link these parts in an executable file based on a link script file.

1.1.1 Using Compilation Rules

Users can input toolchain options and compilation options such as chip version, as below:

```
Make CHIP_VER=MP TOOLCHAIN=mw --- (1)
```

Table 1.1 describes all the compilation rules.

Table 1.1: Compilation Rules

Rule	Description
all	Build $(APPL_FULL_NAME).elf
build	Clean workspace, and then build $(APPL_FULL_NAME).elf
dump	Display the contents of all headers. This rule will generate $(APPL_FULL_NAME).dump. The real command is `Arc-elf32-objdump -x $(APPL_FULL_NAME).elf > $(APPL_FULL_NAME).dump`.
dasm	Display assembler contents of all sections. This rule will generate $(APPL_FULL_NAME).dasm. The real command is, `Arc-elf32-objdump -d $(APPL_FULL_NAME).elf > $(APPL_FULL_NAME).dasm`.
bin	Generate the binary file $(APPL_FULL_NAME).bin and the ini file $(APPL_FULL_NAME).ini. The real command is `Arc-elf32-objcopy -o binary $(APPL_FULL_NAME).elf $(APPL_FULL_NAME).bin`.
hex	Generate intel hex file $(APPL_FULL_NAME).hex. The real command is `Arc-elf32-objcopy -o ihex $(APPL_FULL_NAME).hex`.
help	Display help information for the build system.
cfg	Display the current configuration.
opt	Display the options of toolchain.
info	Display the current build output information.
spopt	Display the configurable items in the current build system.
infodirs	Display all directory information.
infosrcs	Display all source files.
infoobjs	Display all object files.
size	Dispaly the size information of executable output files .
clean	Clean workspace.
boardclean	Clean board library workspace.
distclean	Delete all output files.
run	Build $(APPL_FULL_NAME).elf and download it to the target board through debugger.
gui	Build $(APPL_FULL_NAME).elf and download it to the target board through debugger.

- If you want to enable all rules, run the following command:

```
Make CHIP_VER=MP TOOLCHAIN=mw all --- (2)
```

Note: `all` is the default rule, which means that the compilation command (1) have the same effect with the compilation command (2).

- If you want to remove all the output files that are generated during compilation and connection, run the following

command:

```
Make distclean --- (3)
```

Certainly, some options can still be indicated in the command like shown below:

```
Make CHIP_VER=MP TOOLCHAIN=mw distclean --- (4)
```

But with the `disclean` rule, CHIP_VER and TOOLCHAIN are not used by the build system.

- If the system needs to be programmed on to external non-volatile memory, run the following command:

```
Make CHIP_VER=MP TOOLCHAIN=mw bin --- (5)
```

1.1.2 Using Macro Variables

The build system defines some macro variables that can be used during compilation. Table 1.2 gives detailed descriptions of these macro variables.

The following gives an example of how to use the macro variable `FLASH_XIP_EN` in a system program:

```
#ifdef FLASH_XIP_EN
<code section>
#endif
```

Note: Users are not recommended to use these macro variables.

Table 1.2: Macro Definitions

Macro	Description
CPU_ARC	processor architecture
$(CHIP_ID)	chip ID
PLAT_ENV_$(PLAT_ENV)	platform environment
HW_VERSION	board version
BOARD_$(BOARD)	board name
USE_DW_AHB_DMA	Indicate whether to use AHB DMA.
USE_DW_UART_0	Indicate whether to use UART0.
USE_DW_UART_1	Indicate whether to use UART1.
USE_DW_SPI_0	Indicate whether to use SPI0.
USE_DW_SPI_1	Indicate whether to use SPI1.
USE_DW_SPI_2	Indicate whether to use SPI2.
USE_DW_QSPI	Indicate whether to use QSPI.
FLASH_XIP_EN	Indicate whether to layout executable file onto flash memory space.
USE_CAN_0	Indicate whether to use CAN0.
USE_CAN_1	Indicate whether to use CAN1.
USE_IO_STREAM	Indicate whether to use IO Stream service.
IO_STREAM_PRINTF	Indicate whether to use IO Stream service to print information.
CHIP_ALPS_$(VALID_CHIP_VER)	chip type and version
CURRENT_CORE=$(VALID_CUR_CORE)	current core version
LIB_CLIB	Standard C library. If it's not used, the heap space needed by its memory API can be removed.
MID_COMMON	middleware type
EMBARC_TCF_GENERATED	Indicate that the build of EMBARC is based on TCF file.
PLATFORM_EMBARC	platform name
_HAVE_LIBGLOSS_	The library for GNU low-level OS support. Contains the startup code, the I/O support for gcc and newlib (the C library), and the target board support packages that users need to port the GNU tools to an embedded execution target.
__MW__	Indicate that the current toolchain is metaware.
__GNU__	Indicate that the current toolchain is GNU.
TOOLCHAIN	the toolchain name
_STACKSIZE	stack size.
ENABLE_OS	Indicate whether to use an OS kernel.
$(OS_ID)	OS name.
OS_HAL	Reserved.
FREERTOS_HEAP_SEL	FreeRTOS head type select
INCLUDE_uxTaskGetStackHighWaterMark	Indicate whether the task can get the stack's high watermark.

Continued on next page

Table 1.2 – continued from previous page

Macro	Description
configGENERATE_RUN_TIME_STATS	Indicate whether to generate the task's runtime statistics.

1.1.3 Adding a Board-Level Module

All files of a board-level module should be placed under the peripheral directory, including the make file.

Note: The name of the make file should be the same with that of the board-level module, because the build system searches source and header files based on the make file.

The build system processes the modules that are indicated by the variable *EXT_DEV_LIST*, which is defined in the project make file (by default, `sensor.mk`). The value of *EXT_DEV_LIST* is the name list of a module. For example:

```
EXT_DEV_LIST += flash PMIC
```

To add a board-level module to the build system, users can add the module name to EXT_DEV_LIST. For example:

```
EXT_DEV_LIST += flash PMIC can_transceiver
```

During compilation, the build system checks whether the module exists under the peripheral directory. Each module must have its own sub-directory. If the module exists, its make file is then added to the build system. The make file defines which file should be compiled and adds some other necessary information to the build system.

1.1.4 Adding a System-Level Module

In the radar system, each system-level module is compiled to a separate library according to its own compilation rule. During linking, the build system links all system-level libraries and other object files, and generates an executable image file.

Adding a system-level module to the system is more complicated than adding a board-level module, because users need to define the rule of compiling and the ways of adding it to the options of the linker.

The process can be simplified by adding a new system-level module based on an existing system-level module. The steps are as follows:

1. Copy and paste the directory of an existing module.
2. Modify the name of the new directory.
3. Remove the original source and header files from the new directory.
4. Modify the name of the make file.
5. Add the source of the new module and header files to the new directory.
6. Modify the make file, and add it to the build system.

Note: Most system-level modules are placed under the `calterah/common` directory, except the CLI (FreeRTOS command line) module.

1.2 Debugging

In the radar system, there are two debuggers available, MetaWare Debugger and GDB Debugger.

- If you choose MetaWare Debugger, make sure of the following:
  - The license file is placed on the local computer or a global server in advance.

- The JTAG port is available on the radar board, so that the debugger on the host side can connect to the radar board.

Source-Level Debugging

To debug your program at the source level, compile it with the option -g.

The option -g makes sure that the debugging information is put into the file being compiled. This option also provides the linker with the required information for linking files and placing debugging information into the executable file.

The debugging information, which is stored in the object or executable file, describes the data type of each variable and function, and the correspondence between source-line numbers and executable-code addresses.

Programs compiled without debugging information can also be debugged by MetaWare Debugger, but the debugging operations would be limited.

Assembly-Level Debugging

To debug your program at the assembly level, the program does not need to be compiled with the option -g.

- If you choose GDB Debugger, a scheme that transforms protocols from USB to JTAG on the radar board is a must.

  To start GDB Debugger on the host side, run the following command:

```
Make CHIP_VER=MP gui
```

  The GDB Debugger can also be integrated to ARC GNU IDE Eclipse. For detailed information, see *How to develop and debug radar sensor firmware on ARC GNU IDE Eclipse* under the `Document` directory in the software release package.

  With GDB Debugger, a program must be compiled with debugging information.

1.3 Boot

In the radar system, there are many boot modes. When defining the boot modes, XIP feature, CAN OTA feature, and the system boot time must be considered. For example, the system performance may be different when XIP feature is enabled at different phases.

By default, after power-on, ARC CPU fetches instructions from ROM space. Then ROMCODE, the program on ROM space, reads and performs the firmware. If the system needs to enable CAN OTA feature, users can add a bootloader between ROMCODE and firmware to fulfill CAN OTA feature or boot firmware.

Additionally, users can choose whether to enable NOR flash XIP feature in ROMCODE or bootloader.

1.3.1 Boot Without Bootloader

NOR Flash XIP Feature Disabled

If NOR flash XIP feature is disabled, the boot sequence of the radar system is as shown in Fig. 1.2.

Firstly, ROMCODE reads the flash header from the external flash device.

A lot of boot information is saved in the flash header, such as whether to enable the XIP feature in ROMCODE and the configuration parameters of analog PLL.

- If the PLL_ON flag in the flash header is set, ROMCODE configures analog PLL and switches the clock of the ARC CPU to the PLL clock.

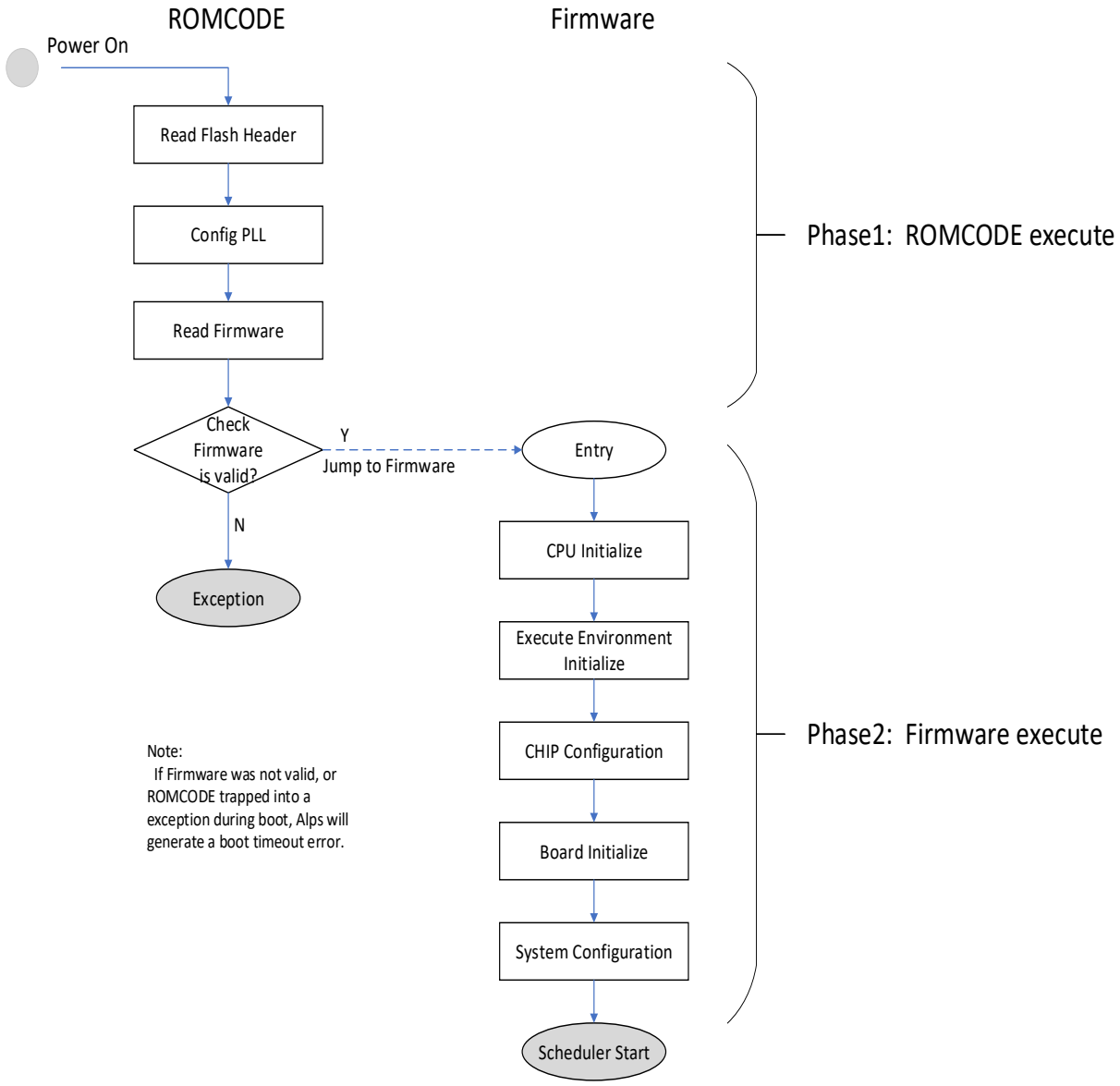

Fig. 1.2: Boot Sequence Without Bootloader (XIP Disabled)

- If the PLL_ON flag in the flash header is not set, the clock of ARC CPU is chosen as the default. This clock is also known as the boot clock, with the value being 50 MHz.

After reading firmware from the external flash device, ROMCODE checks whether the firmware is valid or not.

- If the firmware is valid, ROMCODE executes the firmware and enters the second boot phase.
- If the firmware is invalid, ROMCODE enters a while loop, and waits until the boot watchdog generates a timeout error.

NOR Flash XIP Feature Enabled

If NOR flash XIP feature is enabled, the boot sequence of the radar system is as shown in Fig. 1.3.

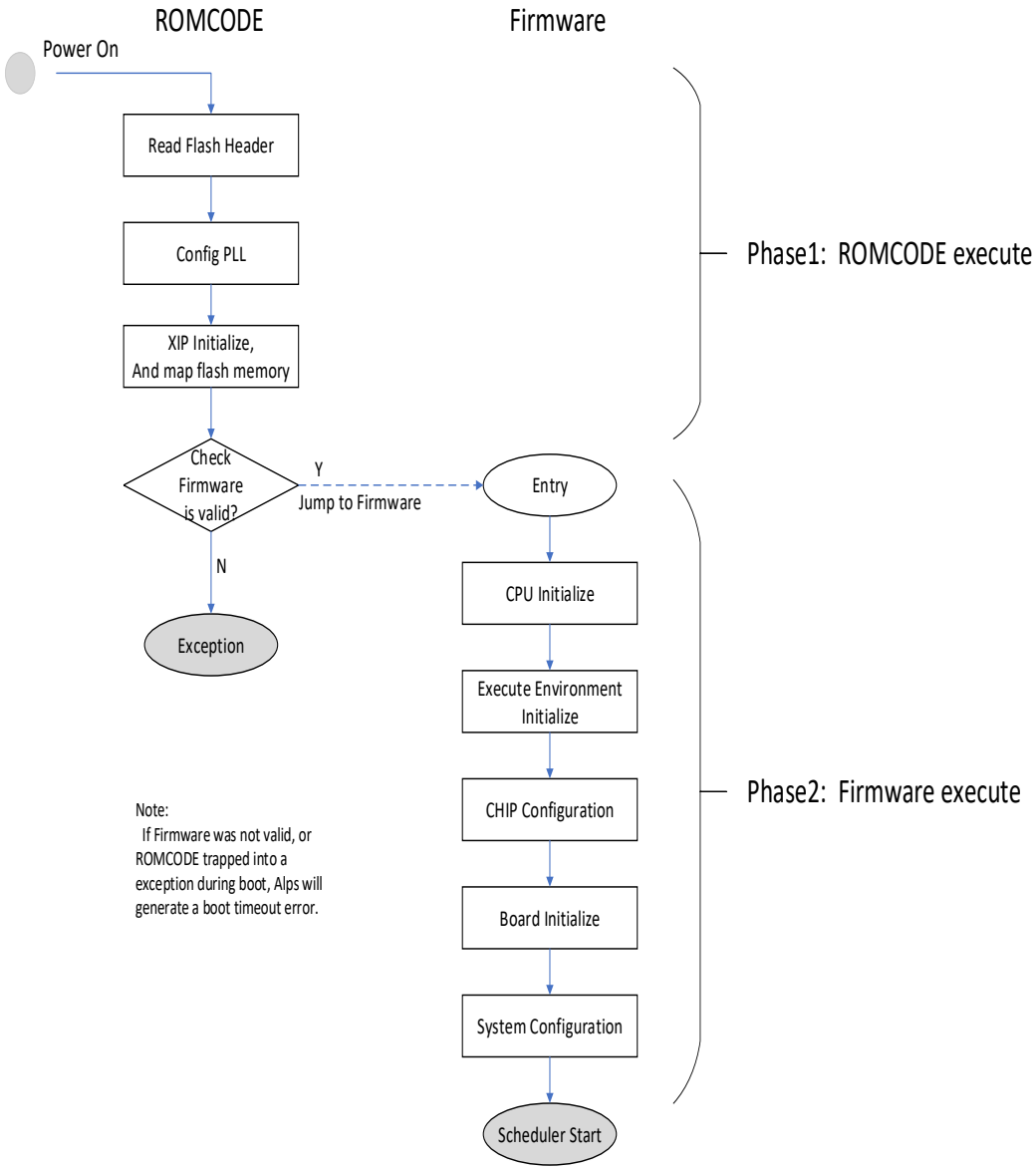

Fig. 1.3: Boot Sequence Without Bootloader (XIP Enabled)

In this mode, ROMCODE must configure analog PLL, and switch the clock of the ARC CPU to the PLL clock. Otherwise, the XIP feature will not operate normally,because the system clock frequency is too low.

After XIP initialization, ROMCODE should firstly map the external flash memory space to that of ARC CPU, according to the configuration parameters in the flash header, and then run the firmware.

1.3.2 Boot With Bootloader

NOR Flash XIP Feature Disabled

If bootloader is added to the system and the XIP feature is disabled, the boot sequence of radar system is as shown in Fig. 1.4.

During the execution of bootloader, if the CAN OTA flag has been set, ROMCODE enters CAN OTA flow and waits for the message from CAN bus.

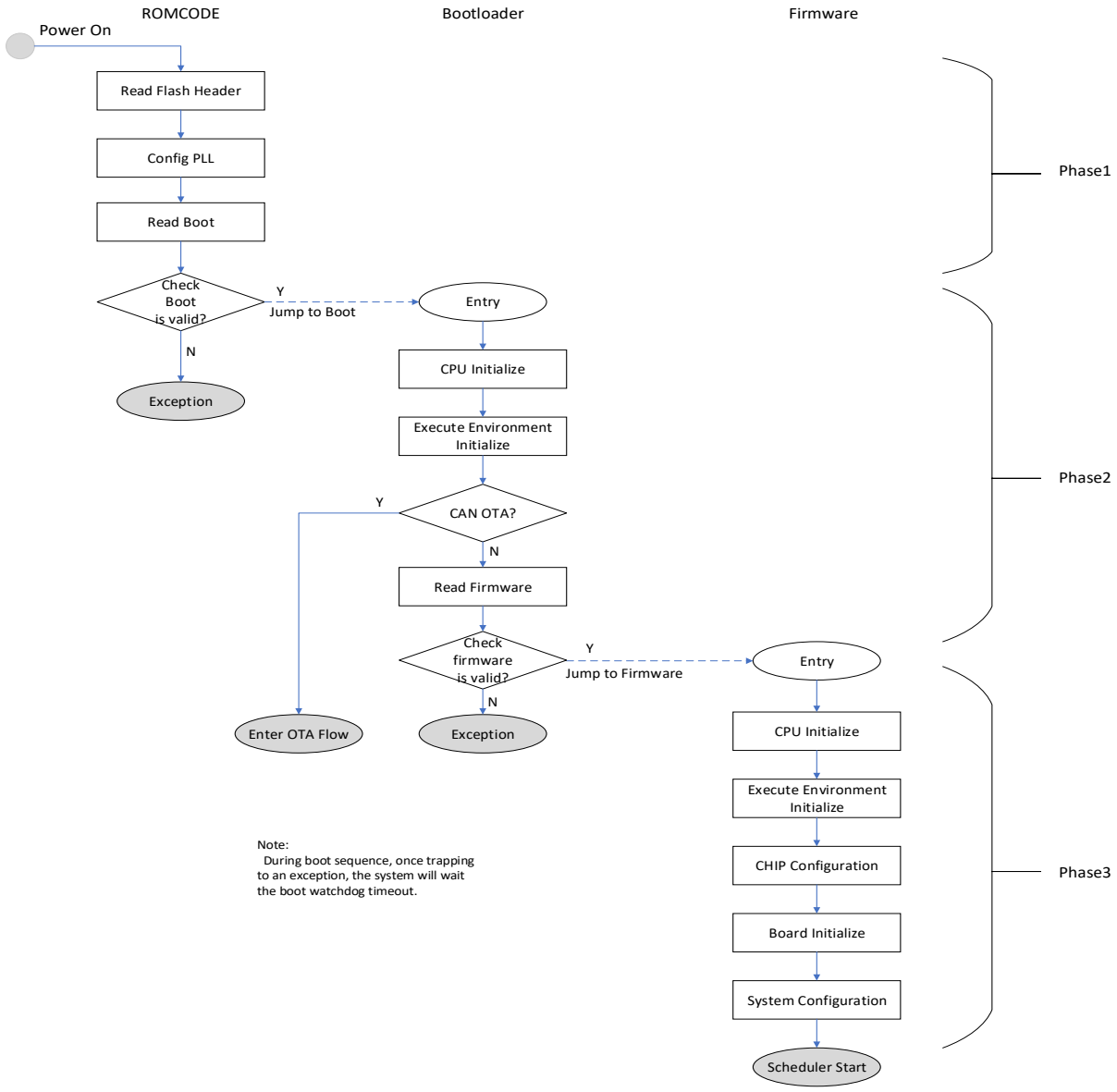

Fig. 1.4: Boot Sequence With Bootloader (XIP Disabled)

NOR Flash XIP Feature Enabled

If bootloader is added to the system and the XIP feature is enabled, the boot sequence of radar system is as shown in Fig. 1.5.

XIP feature can also be enabled at Phase 1. If CAN OTA flag is valid, bootloader has to disable the XIP feature and reprograms itself or the firmware that is stored in external flash memory. Otherwise, the program cannot run properly and the performance will be affected.

The existing bootloader can be replaced by a new bootloader from the host. If so, CAN OTA progress will cost much more time.

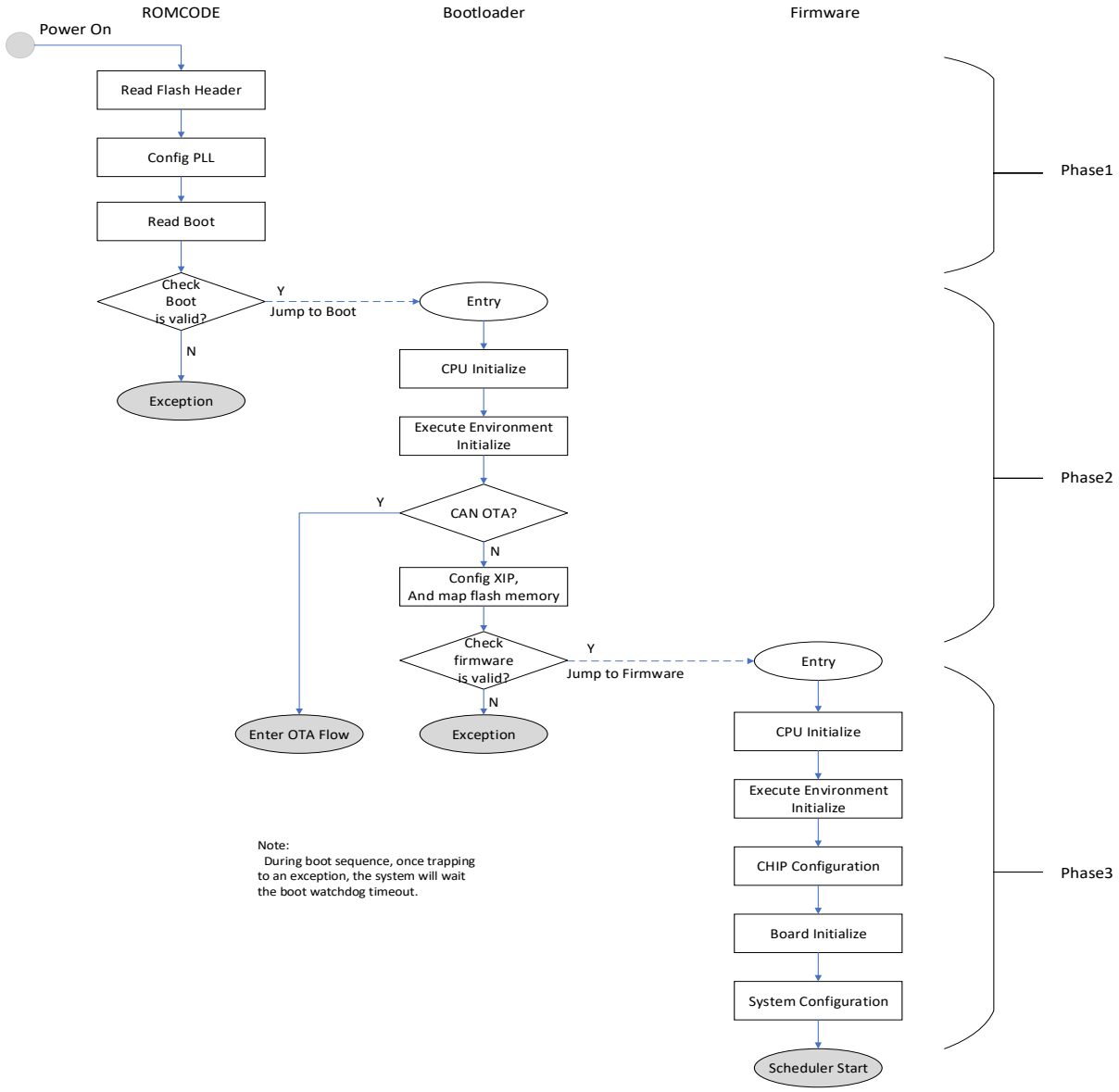

Fig. 1.5: Boot Sequence With Bootloader (XIP Enabled)

CHAPTER

TWO

HAL

2.1 ARC EM6

The ARC EM6 is a processor based on the ARCv2 instruction-set architecture in a highly efficient three-stage pipeline. It integrates the modules that are close to the core, such as ARC timer, ARC watchdog, and cache.

In the radar system, ARC timer has been configured to count system ticks, and hardware exact delay functions are based on the ARC RTC counter. If the system or an application program needs an exact delay, it can call hardware delay functions offered by ARC driver.

The ARC EM6 has 256 exceptions, as defined in Table 2.1. In the radar system, a function pointer array has been defined to describe the exception vector table. Users can register their own exception handlers by calling the API offered by ARC driver. To reduce the complexity of interrupt handlers, an extra function pointer array has been defined. Interrupt handlers registered to this array need not save the context at first or restore the context at last.

Table 2.1: Exception Vectors

Name	Vector Offset	Vector Number	Exception Types
Reset	0x00	0x00	Exception
Memory Error	0x04	0x01	Exception
Instruction Error	0x08	0x02	Exception
EV_MachineCheck	0x0C	0x03	Exception
EV_TLBMissI	0x10	0x04	Exception
EV_TLBMissD	0x14	0x05	Exception
EV_ProtV	0x18	0x06	Exception
EV_PrivilegeV	0x1C	0x07	Exception
EV_SWI	0x20	0x08	Exception
EV_Trap	0x24	0x09	Exception
EV_Extension	0x28	0x0a	Exception
EV_DivZero	0x2C	0x0b	Exception
EV_DCError	0x30	0x0c	Exception
EV_Misaligned	0x34	0x0d	Exception
EV_VecUnit	0x38	0x0e	Vector unit Exception
Unused	0x3C	0x0f	/
IRQ16-255	0x40-0x3FC	0x10-0xff	Interrupts

ARC driver offers many useful methods for users.

- Methods of accessing memory in an un-cacheable manner by ARC CPU.
  - *Inline uint32_t _arc_read_uncached_32(void *ptr)*

  - *Inline uint32_t raw_readl(uint32_t addr)*
  - *Inline void _arc_write_uncached_32(void *ptr, uint32_t data)*
  - *Inline void raw_writel(uint32_t addr, uint32_t val)*

  The first argument in each method is the memory address.

- Methods of accessing memory in a cacheable manner by ARC CPU.
  - *Inline uint32_t _arc_read_cached_32(void *ptr);*
  - *Inline void _arc_write_cached_32(void *ptr, uint32_t data);*

  The first argument in each method is the memory address.

- Methods of reading and writing auxillary registers by ARC CPU.
  - *_arc_aux_read(aux);*
  - *_arc_aux_write(aux, val);*

  The first argument in each method is the register address.

- ARC CPU exception or interrupt methods.
  - *int32_t exc_handler_install(const uint32_t excno, EXC_HANDLER handler);*
  - *int32_t int_handler_install(const uint32_t intno, INT_HANDLER handler);*
  - *int32_t int_disable(const uint32_t intno);*
  - *int32_t int_enable(const uint32_t intno);*
  - *int32_t int_sw_trigger(const uint32_t intno);*
  - *uint32_t cpu_lock_save(void);*
  - *void cpu_unlock_restore(const uint32_t status);*

2.2 Chip

In the radar system, the system-level chip module specifically refers to the CPU, such as ARC EM6. The global resources on the CPU, the drivers of global chip-level modules, and the global information of non-global chip-level modules all belong to the chip module.

- The global resources include interrupt signals, DMA request signals, and global registers. In the radar system, the assignment of the interrupt logic IDs (IRQ16-255, see Table 2.1), DMA request logic IDs, global registers, and the mapping of CPU memory space are all described in the chip module.
- Typically, global chip-level modules include the clock tree consisting of clock source selectors, clock dividers, and clock gates, the debug management unit (DMU) and the error management unit (EMU).
  - The clock tree has its own driver. The driver offers clock operation methods, including source selection, clock division, and clock frequency getting.
  - The DMU driver fulfills operation methods for digital miscellaneous features, which include multiplexed pins (IOMUX), the debug bus, radio SPI controller, and global interrupt controller.

    Multiplexed pins have been described in an array table, which is placed in the board module because different boards may have different configuration. During the system boot, all multiplexed pins will be configured based on this table. If users need to dynamically change the function of a group of pins, the inline functions are available in the header file.
  - The EMU driver encapsulates the lowest chip-level operation methods for system safety features.

- The global information of non-global chip-level modules includes the base address and the interrupt logic IDs. Each non-global chip-level module defines the structure of its own module descriptor to describe its global information and point to its operation method set. An instance of the descriptor structure is placed in the chip module, and a method to get this structure is offered together. An upper layer program can call this method to get the global information of a non-global chip-level module.

Note: All operation methods offered by the chip module are at the physical layer. Re-entrancy must be taken into consideration when using these methods.

2.3 Device

The system-level device module includes the drivers of both chip-level and board-level modules. For example, the UART controller is a chip-level module. The flash is a board-level module, and it is also regarded as a peripheral device. This chapter discusses only chip-level modules and the associated driver models. For board-level modules, see *Peripheral*.

On chip-level modules, the physical layer encapsulates registers into the lowest operation methods as a feature. Before using these modules, the upper drivers install these operation methods to the associated module descriptors, so the upper layer program can find these methods through module descriptor and call these methods.

In most cases, an IP, such as DW_SSI, is implemented into multiple modules. Many chip-level modules like SPI0, SPI1, QSPI are its instances. Modules have their own module descriptors, but share one physical layer driver. As mentioned in the last paragraph, the operation methods offered by a physical layer driver have been installed in different module descriptors. Therefore, the HAL program can gain a module descriptor by only a logic module identifier. It then can control and access the module.

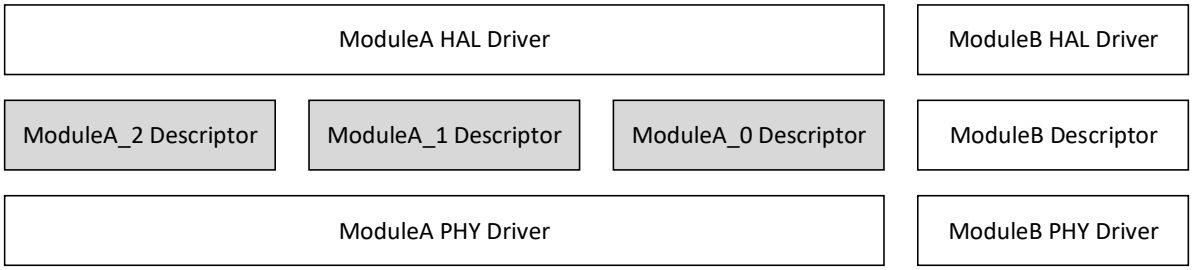

Fig. 2.1: Chip-Level Driver Model

Of course, board-level or system-level modules can directly get the operation methods offered by physical layer driver.

2.3.1 GPIO Driver

The GPIO driver provides several operation methods for the system, including initialization, setting data direction, writing indicated data to hardware lines and reading sampled data from hardware lines.

If users want to start work by a GPIO interrupt, the GPIO interrupt callback function must be installed at first.

To initialize the GPIO module, follow these steps:

1. Get the GPIO descriptor and install a GPIO physical driver.
2. If the GPIO exists, enable the clock gate of the GPIO controller. Otherwise, return an error to the caller.

3. Register the interrupt service routine for each GPIO.

Before operating a GPIO, set up its data direction.

Theoretically, each GPIO with the interrupt feature, that is, the GPIO with a separate interrupt signal logic ID, can have its own interrupt service routine. However, GPIO driver provides a common interrupt service routine for all GPIOs for convenience. The upper layer program can register a callback function for the used GPIO. The callback will be executed in the common interrupt service routine.

Note: Not every GPIO has the interrupt feature. In the radar system, the GPIO with a logic ID from 16 to 31 cannot trigger interrupt signals to ARC CPU.

2.3.2 DMA Driver

DMA can be used by the system or application program as a replacement of Memcpy to copy data. It can also be used by the drivers of other modules to read data from the hardware FIFO, or write data to the hardware FIFO. For example, DMA can be used by SPI driver to transfer data from the memory to the hardware FIFO of the SPI controller, and then trigger SPI to send the data onto SPI data lines.

The DMA controller has eight logic channels, and each of them can be configured by users separately. The configuration parameters can be split into three parts.

- Parameters about the source of transfer connection, including:
  - source addresses
  - addressing modes, which include:
    - increment mode
    - decrement mode
    - fixed mode
  - handshake types between the source and the destination of transfer connection, which include:
    - hardware handshake
    - software handshake

  If the feature of source gathering is enabled, the Source Gather Count and Interval also need to be configured.
- Parameters about the destination of transfer connection, including:
  - addressing modes
  - handshake types
  - destination addresses

  The Destination Scatter Count and Interval are optional, depending on whether the destination scatter feature is enabled or not.
- Global parameters of the logic channels, which are allocated to a transfer, including:
  - channel priority
  - transfer types, which include:
    - memory-to-memory
    - memory-to-peripheral
    - peripheral-to-memory

DMA driver defines a DMA channel descriptor for each channel. The descriptor includes the state and the configuration parameters of the channel.

Before starting a DMA transfer, the user applies a DMA logic channel by calling the method offered by the DMA driver. If all DMA logic channels are busy, the user has to wait until other users release one. Once a channel is applied successfully, its state will change to the *allocated* status, which means that the channel is locked for other users in the system to reapply it. In this case, the user must configure the channel.

Once the transfer has finished, the user must release the channel.

To initialize DMA, follow these steps:

1. Get the DMA descriptor and install DMA physical driver.
2. If the DMA exists, enable the clock gate of DMA controller. Otherwise, return an error to the caller.
3. Allocate memory for each DMA channel descriptor and initialize all channels.
4. Register the interrupt service routine.

2.3.3 SPI Driver

Serial Peripheral Interface (SPI) is a full duplex synchronous serial communication interface used for short range communications, especially for connecting MCU to a board-level module. Generally, an SPI master can communicate with multiple SPI slaves. These slaves are supported through individual slave select lines.

The SPI master sets the baud rate for an SPI transfer connection. So even if an SPI slave has sent the data to the master, the master still needs to trigger data in the hardware FIFO of the slave side to data lines, by sending the clock to the slave.

SPI supports four different clock modes. The selection of the mode depends on the combination of clock polarity and phase. Before starting any transfer, SPI clock mode should be set up.

- Clock polarity is the state of the clock, i.e. idle or active. If idle state is 0, active state will be 1, and vice versa.
- Clock phase selects the relationship of the clock with the slave select signal.
  - If the clock phase is 0, data is captured on the first edge of the clock.
  - If the clock phase is 1, the clock starts toggling one cycle after the salve select line is activated, and data is captured on the second edge of the clock.

SPI driver offers a uniform transfer API for different communication types, which include full duplex, half duplex, and simplex. That is, no matter which communication type is selected, the upper layer program always calls the same function. By default, SPI driver initializes the SPI controller to make it work in half duplex communication mode.

SPI driver offers extra operation methods that use DMA to transfer data. These methods are optional. To reduce the size of ROM space of the system, users can choose not to compile and link the methods into the executable image file. The ROM space includes the code section and the read-only-data section.

SPI driver shares the DW_SSI physical layer driver with other modules like QSPI. Therefore, the DW_SSI must be reentrant for different HAL drivers.

In DW_SSI physical driver, if the SPI master sends data, the data is split into multiple parts. Once a part is padded into the hardware FIFO, DW_SSI physical driver triggers a transfer to output them to data lines. The biggest advantage of this scheme is that the CPU has time to process other system tasks when the hardware is sending data, and at the same time, SPI slave on the other side has enough time to read data. To ensure the accuracy of the data, an extra delay between each part can be added. But the extra delay will reduce the throughput rate of the connection.

In DW_SSI physical driver, if the SPI slave sends data, the data is also split into multiple parts. Once a part is filled into the hardware FIFO, the SPI slave waits for the SPI master on the other side to send a clock for reading the data, and at the same time, the CPU on the slave side can handle other system tasks.

To raise the efficiency of data transfer and ensure the accuracy of the transferred data, the upper layer program needs to achieve an extra synchronization mechanism between the master and the slave. By default, SPI connects the cascade master with the slave. The cascade interface module has achieved a synchronization mechanism. For detailed information, see *Cascade Interface*.

For the operation methods offered by SPI driver, see *Peripheral APIs*.

2.4 Board

The system-level board module describes board-level or product-level configuration of the radar system, especially the configuration differences on different boards.

As mentioned in *Chip*, because differences may exist among different boards, the multiplexed pin table is placed in the board module. During the system boot, all multiplexed pins are configured based on this table. This table has two major advantages. First, the functions of the multiplexed pins can be changed easily and statically. Secondly, IOMUX driver compatibility can be easily achieved.

Similarly, the configuration parameters of CAN baud rate may also be different on different boards. So these parameters should also be placed in the board module. CAN driver can get the parameter of an indicated baud rate by calling the associated method.

The mapping or partition of the external flash memory is also placed in this board module. Fig. 2.2 describes the default partition of the external flash memory.

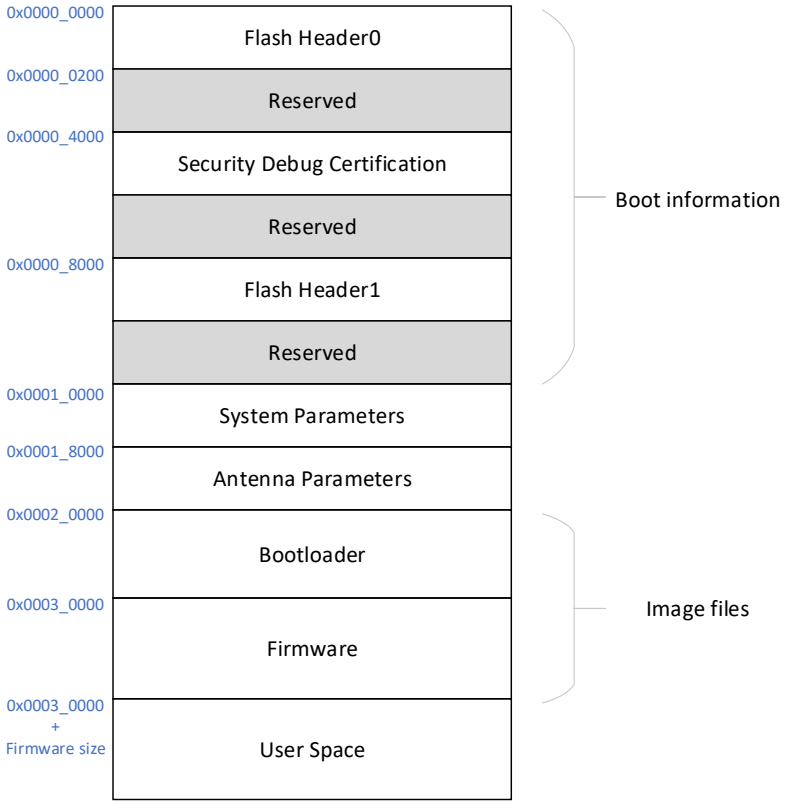

Fig. 2.2: Default Partition of External Flash Memory

Flash Header1 is a back-up partition of Flash Header0. In case the sector in which Flash Header0 is located is damaged, ROMCODE can read from Flash Header1 .

The Security Debug Certification partition is defined for the security debugging feature. This partition is reserved when the system security feature is not enabled, or the system security feature is enabled but the system does not need to enter the security debugging mode.

The System Parameters partition stores system information, such as system startup counter, system power-off counter, the usage information of the user space partition of flash memory, and system error information.

2.5 Peripheral

Peripherals are located on the board, so they are also called board-level modules, which means that these modules are not part of the MCU. Generally, these modules are connected to the MCU through serial or parallel interfaces or GPIOs. The MCU can configure these modules or get the states of them by sending control frames. It can also get the data stored on these modules or store data onto these modules by sending data frames.

In the radar system, NOR flash device is an important board-level device. The system program and parameters are both stored in the NOR flash device. For detailed information, see *NOR Flash*.

If the radar system needs to be connected to a CAN bus, CAN transceiver must be added between the CAN bus and the CAN controller in MCU. In this case, an extra serial interface is required to control the CAN transceiver.

2.5.1 NOR Flash

QSPI interface connects the MCU with the external NOR flash device, so NOR flash driver must have the ability to control and access the QSPI module. Since the system does not achieve HAL driver for QSPI, the NOR flash driver must call the methods offered by its physical layer driver. This design improves the system performance since a layer of software encapsulation is reduced.

The NOR flash driver supports and can dynamically distinguish the devices that are compliant with Common Flash Interface (CFI) Standard or Serial Flash Discoverable Parameter (SFDP) Standard.

- If the system selects to dynamically recognize the NOR flash device in NOR flash initialization function, NOR flash driver first tries to read NOR flash identifier information by sending CFI commands.
  - If the information read from the external NOR flash device matches the CFI query identification string, NOR flash driver continues reading the parameters of the NOR flash device.
  - Otherwise, NOR flash driver tries to send SFDP commands to read NOR flash identifier information until it matches the CFI query identification string.

  Once NOR flash driver gains the correct parameters of the NOR flash device, it parses them and obtains the memory density, memory region information, sector sizes, page sizes, and sector erase instructions.
- NOR flash driver can also statically recognize the NOR flash device before compilation. This scheme reduces the boot time because the NOR flash driver does not need to read the identifier information and parameters of the external NOR flash device.

The access progress of NOR flash memory must be protected. The access operations include reading, programming, and erasing. If an upper layer program is using the hardware resources, another upper layer program must not enter the progress. A semaphore is created by the NOR flash driver to protect the access progress. Any access to NOR flash device should get the semaphore. Otherwise, it should wait until the completion of the access progress.

- When reading NOR flash memory, if the length of the data to be read exceeds the page size of the NOR flash memory, the reading will be split to several times to ensure that each time the length of the data to be read is equal to or less than the page size. Each time before starting reading, the interrupt feature must be disabled. To

ensure that other system tasks can be processed, the interrupt feature must be re-enabled after the data reading is finished. The period when the interrupt feature is disabled should not be too long. Otherwise, some real-time tasks cannot be processed in time.

- The progress of NOR flash memory programming is similar with that of reading, except that the external NOR flash device has to be idle each time before starting programming.

For the operation methods offered by the NOR flash driver, see *Peripheral APIs*.

CHAPTER

THREE

SSL

3.1 OS Kernel

The operation system (OS) kernel of the radar system is FreeRTOS. It supports task preemption and configurable task priorities, queue, MUTEX, semaphore, and etc.

With the FreeRTOS Plus feature and Command Line added, the system can call the registered callback function based on the received messages. For detailed information, see *FreeRTOS Command Line*.

Other RTOS can also be added into the system, but system engineers must achieve an operation system abstraction layer.

3.2 Console

In the system, console is a module in the system service layer (SSL). It mainly manages the input and output data through UART, and reports the received data to the upper later program or other system service or modules.

The console receiver manages the data-receiving progress. Because both the time of the data arrival and the length of the arrival data are not known in advance, it is necessary to allocate a large memory block for the receiver to buffer the received data. The memory size can be statically changed before compilation. Received data will be read from the hardware FIFO, and store in the local buffer. Then the location and length of the data will be inserted a queue, so the upper layer program or other system-level modules can get messages from the queue.

The console transmitter manages the data-transmitting progress. It defines three types of buffer hookers. For detailed information, see Fig. 3.1.

- The first type of buffer hookers is used in the interrupt context to print log, such as errors, warnings, and other information.

  The buffer on this type of buffer hookers is allocated and managed locally.

- The second type is used in the task context to print log.

  The buffer on this type of buffer hookers is also allocated and managed locally.

- The third type is reserved for users, and the buffer is allocated and released by users.

  So once the data in these buffers have been sent, the console transmitter must confirm with the user.

Additionally, to ensure the correct order of messages, for each message console transmitter assigns an order ID while allocating buffer hookers.

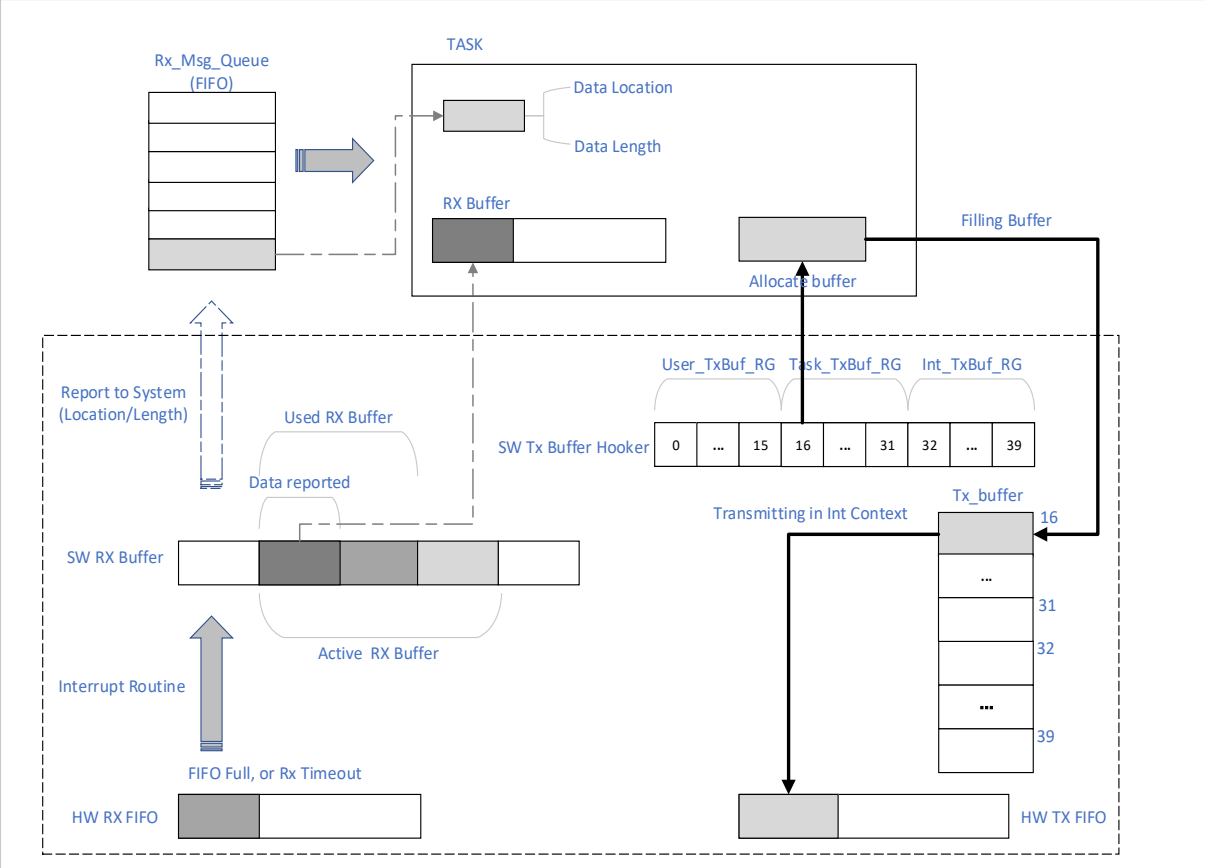

Fig. 3.1: Console Data Stream (Interrupt)

3.2.1 Message-Receiving Progress

When the amount of the data in the hardware RX FIFO exceeds the watermark, or there is no data written into the hardware RX FIFO within a certain period, an interrupt signal will be asserted. Once ARC CPU detects this signal, it enters associated interrupt service routine. In the routine, ARC CPU reads data from hardware RX FIFO, and stores the data at the top of the local buffer, starting from the base address of the free buffer. After all the data has been read, the location and length of the stored data will be inserted a queue.

The local buffer has a writing pointer and a reading pointer. If either of the pointers reaches the end, it will roll back to the head of the local buffer. And as long as new data arrives, ARC CPU continues reading the data onto the local buffer in the interrupt service routine.

A very bad situation is that the data reading of the upper layer program is so slow that the writing pointer reaches the reading pointer. In this case, the error information is recorded in the error trace memory. To avoid this bad situation from happening, the length of the local buffer should be controlled within a reasonable range, and more ideally, the communication should support hardware flow control or break features like UART break signals.

3.2.2 Message-Transmitting Progress

In the interrupt or task context, a caller must apply a buffer hooker before printing log or sending data. If all buffer hookers are busy, the task should be kept in the sleep mode and wait for one to be released. Once the buffer hooker is applied, its state will change into *allocated*. Then, the caller can fill data to the buffer hung on the hooker, change the state of the hooker to *data ready*, and wait for the console transmitter to handle it.

In the interrupt context, it is recommended that users should reduce the log printing as much as possible, because log printing prolongs the execution time of the interrupt service routine.

In the exception context, all messages buffered on local buffer hookers will be printed out. And all exception information should then be directly printed out.

Once the hardware has finished transmission of all data in a buffer hung on a hooker, the console transmitter changes the state of the buffer hooker to *idle*. If the next buffer hooker is in the state of *data ready*, the console transmitter installs the buffer hung on the hooker to the lower layer driver.

If the buffer hooker is reserved for users, once the message buffered on the hooker has been transmitted, the console transmitter should confirm with the upper layer program. Once the upper layer program receive this signal, it can release the buffer.

3.3 Command Line

3.3.1 FreeRTOS Command Line

The radar system also integrates FreeRTOS Command Line Interface (CLI). FreeRTOS CLI provides a simple, extensible, and RAM-efficient method of enabling the radar system to recognize all ASCII characters and process input strings of the command line. For the ease of use, some extra features have been added.

The message processing is achieved in a task, which checks whether the queue received by the console is empty. If the queue is not empty, the task gets the message from the top of the queue, and processes the characters in the message one by one.

The command processor walks through the command list to find the command that matches the input string. If there is a matching command, the command processor executes its routine. In this routine, some output may be filled into the output buffer. After the execution, the output is cleared. If there is no matching command, the command processor outputs a message to inform the user.

To view the history commands that are input through CLI, users can press the up arrow key to find the previous input command strings until the oldest one is reached, and the down arrow key to find the next until the latest one is reached. By default, at most eight commands can be buffered. The upper limit of command strings that can be buffered can be modified depending on the needs of the application.

To modify the history commands that are input through CLI, users can navigate to the position where they want to make changes by moving the cursor through the left, right, up and down keys, and then make modifications. The modified command string can be used as a new input to CLI, and the associated service routine will then be executed.

Note: The help command is used to output the command information in the command list.

3.3.2 CAN Command Line

The radar system works as a sub-node on the CAN bus to communicate with other nodes. It can send and receive messages. As a device terminal, the radar system receives and responds to the commands on the CAN bus. The received commands are called CAN command lines in the Alps radar system.

In the Alps radar system, firstly, call the can_cli_init() API, which will do the following steps:

1. Initialize the CAN hardware, set Baud rate, and install CAN ISR.
2. Create the CAN ISR queue called queue_can_isr and CAN task called can_cli_task.
    - If no interrupt is available in queue_can_isr, can_cli_task will be in the sleep mode.
    - If there is an interrupt available in queue_can_isr, can_cli_task will be woken up.
3. Register the callback function can_cli_rx_indication for CAN receiving message. Pass the received message to the array rx_msg_buffer[idx].

Secondly, register CAN CLI commands with can_cli_register (SCAN_FRAME_ID, can_scan_signal). Pass CAN CLI (message ID and callback function) to the global static array can_cli_cmd_list[idx].

When the received data is available on CAN bus, an interrupt will be generated. The interrupt callback function can_cli_rx_indication passes the received message to rx_msg_buffer[idx]. The queue message will then be sent through queue_can_isr, and wake up the can_cli_task of task. The task will look for the received message ID in the array can_cli_cmd_list[idx]. If the message ID is registered, it will call the callback function in can_cli_cmd_list[idx], which is the response event of the CAN CLI command.

Currently, the default registered CAN CLI command on Alps is SCAN_FRAME_ID. It controls the output of the radar data. For more instructions about CAN applications, see *CAN Communication API User Guide* in the release package.

Fig. 3.2 illustrates the framework of CAN CLI.

3.4 Cascade Interface

In cascade applications, two radar chips are connected through a SPI interface and a GPIO called S2M_SYNC signal, as shown in Fig. 3.3. The cascade master and slave can transfer data through the interface in half duplex communication mode.

To ensure the accuracy of the data transferred through the interface, the cascade interface module has a design of a simple communication protocol as shown in Fig. 3.4.

The original data is split into multiple data blocks, also called payloads. The maximum length of these blocks is 4 KB. A data block, the CRC of the data block, and a header form a data frame, with the header being used to indicate the

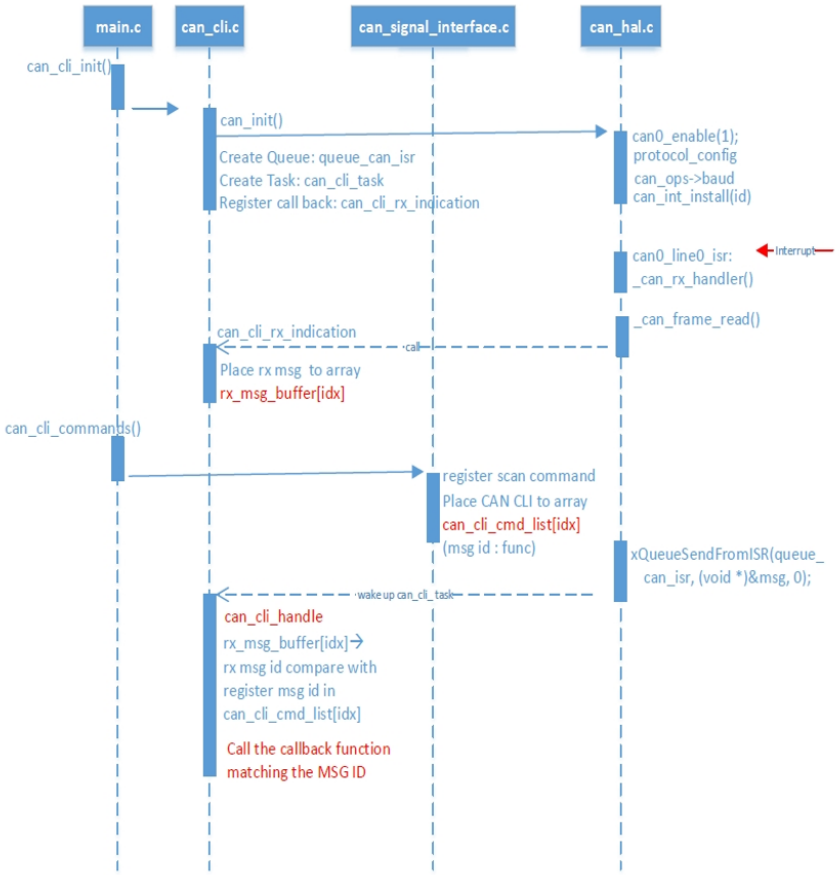

Fig. 3.2: CAN CLI Framework

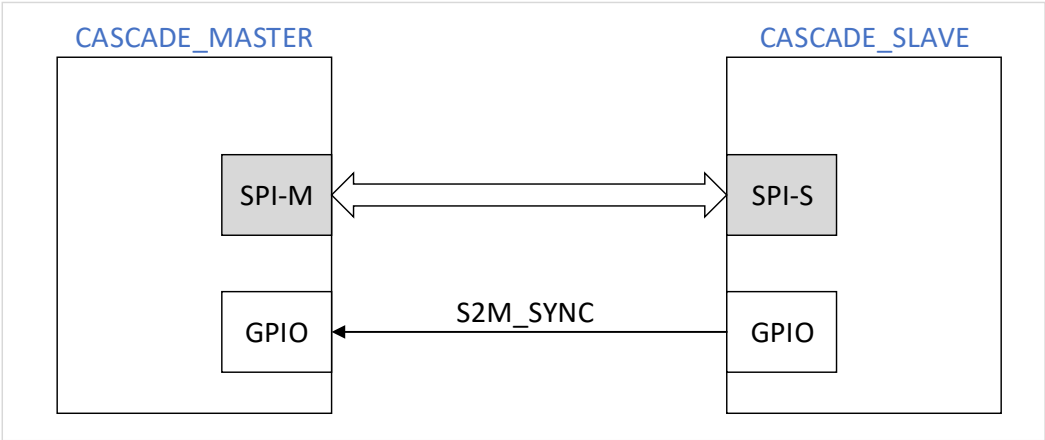

Fig. 3.3: Connection of Cascade Interface

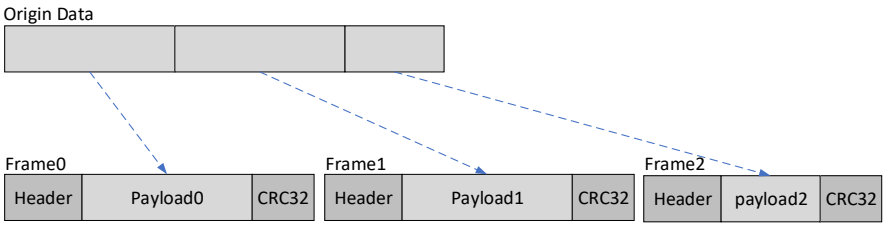

Fig. 3.4: Communication Protocol of Cascade Interface (Example)

length of the payload, which is 4 bytes. For the format of the data frame header, see Fig. 3.5. The header of the first data frame indicates the length of the whole data transfer.

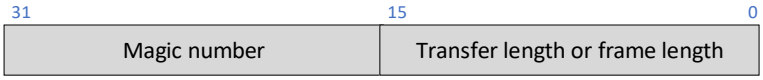

Fig. 3.5: Format of Data Frame Header

3.4.1 Data Transfer from Master to Slave

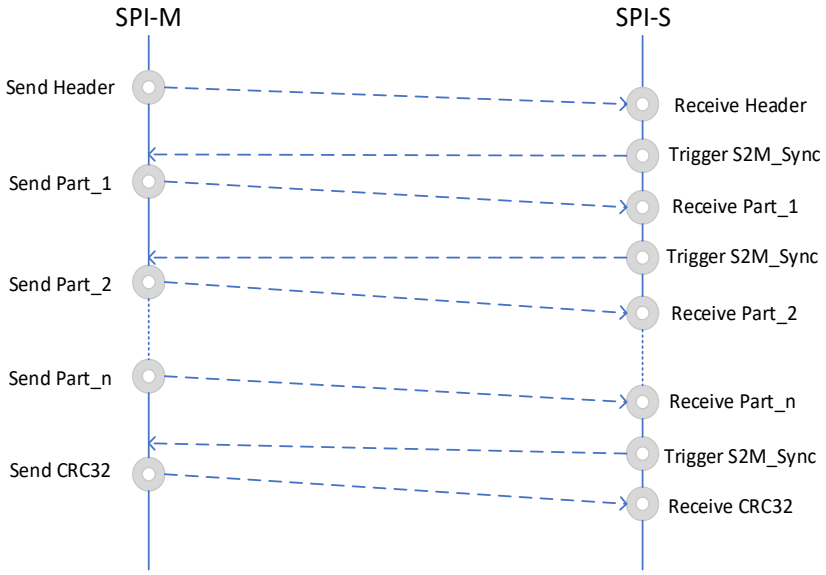

Fig. 3.6: Data Transfer from Master to Slave

Fig. 3.6 illustrates the data transfer flow from the master to the slave.

1. The master sends the header of the data frame to the slave.
2. After receiving the header, the slave checks the magic number in the header, and parses the transfer or payload length from it.
3. If the header is valid, the slave asserts the S2M_SYNC signal to the master.

4. After detecting the S2M_SYNC signal, the master sends the payload of the data frame to the slave .

   As described in *SPI Driver*, the lower layer driver will split the payload into multiple parts, with the maximum length of these parts being 32 words or 128 bytes, which is the depth of the hardware FIFO.

   After a part of the payload has been filled into the hardware FIFO, the master can process other system service or tasks.

5. Each time when a part of the payload is received, the slave asserts the S2M_SYNC signal to the master to start sending next part.
6. After all the data frames have been sent, the master sends the payload CRC to the slave.

3.4.2 Data Transfer from Slave to Master

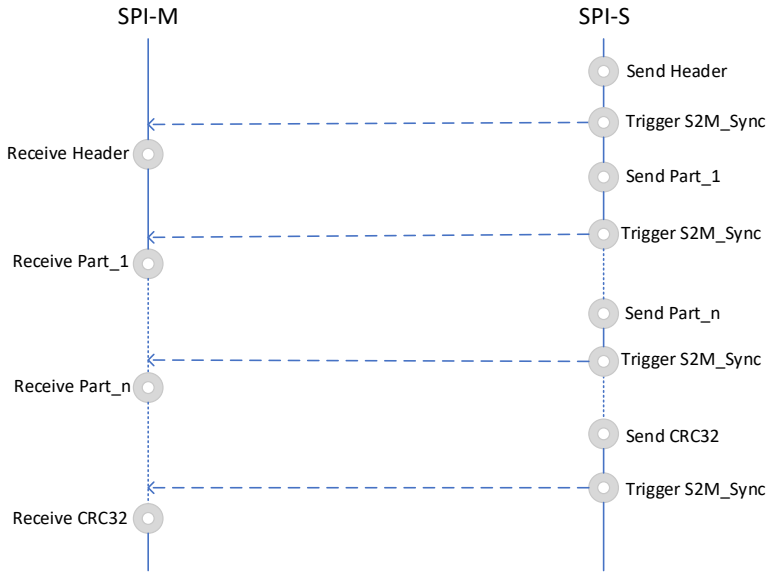

Fig. 3.7: Data Transfer from the Slave to the Master

Fig. 3.7 illustrates the data transfer flow from the slave to the master.

1. The slave fills the header into the hardware FIFO, and informs the master to receive the header by asserting the S2M_SYNC signal.
2. After receiving the header, the master checks the magic number in the header and parses the transfer or payload length from it.
3. If the header is valid, the master initializes the transfer progress and prepares to receive the data frames to come.
4. Once the header in the hardware FIFO of the slave has been sent, the slave begins to fill in a part of the payload, and sends the S2M_SYNC signal again to the master.
5. After detecting the S2M_SYNC signal, the master sends a clock to the slave and receives the part of the payload.
6. After sending all the data frames, the slave fills the CRC of the payload and asserts the S2M_SYNC signal to the master.

3.5 Baseband Service

Digital BB (Baseband) is responsible for the processing of IF signals from the radio subsystem. The signals go through the following processing of Alps Radar Baseband in sequence:

1. ADC Sampling
2. 2D-FFT
3. CFAR
4. DoA

3.5.1 Baseband Task

In Alps Radar Firmware, a task is created to handle the baseband behavior. This task is called `baseband_task` and implemented in the file `baseband.c` under the directory of `\calterah\common\`.

During radar signal processing, as soon as DBF (digital beamforming) unit finishes its job, an IRQ service called `baseband_int_handler` will be raised to notify CPU by `queue_bb_isr` message queue for further processing, such as tracking and other application-specific data process.

Fig. 3.8 illustrates the main flow of the `baseband_task` function.

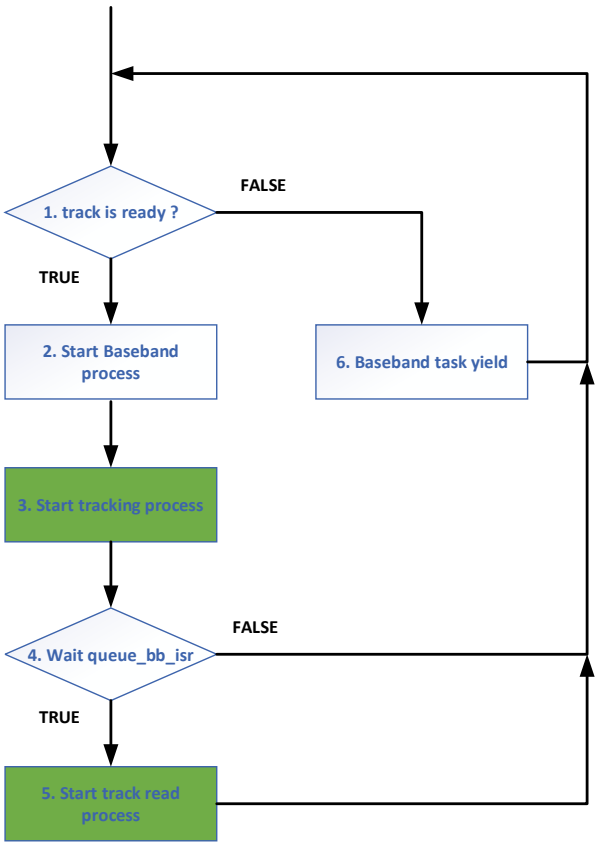

Fig. 3.8: Flow of baseband_task

The radar parameters (range, velocity, angle, power, and status) read from Baseband in Step 5 will be processed in Step 3. This means that after the Baseband process, a new frame is started in Step 2. The tracking process of last frame that is read from Baseband in Step 5 can be executed immediately, because of the separate Baseband and CPU architectures in Alps radar chip.

3.5.2 Tracking System

The Tracking module is sitting in the directory of `calterah/common/track`. We have the following conversion to organize the current code:

- `track.h` exports interfaces to the remaining part of the firmware.
- Those interfaces are implemented in `track.c`.
- Tracking algorithm is implemented in `YYY_track.c`, where `YYY` stands for the name of a particular algorithm.

  For example, currently we have standard *extended Kalman filter* (EKF) implemented with the c-file `ekf_track.c`.
- `YYY_track.h` includes interfaces that we are intent to export to the rest of firmware, in particular, the tracking system.

The remaining part of this section gives an overview of the tracking system. The focus will be interfaces and code structures instead of algorithm itself. For details, please refer to *[Anderson1979]* or other classical textbooks.

3.5.2.1 Interfaces of Tracking

The following is a list of function call interfaces[1]:

- *track_init* : Called at initialization of the system. It is designed to set up the tracking algorithm and timers for tracking.
- *track_pre_start* : Used to call initialization for the underlying tracking algorithm.
- *track_is_ready* : Used to check whether the predefined timing expired so that the tracking system can be ready to kick out another frame.
- *track_lock* : Used to prevent pre-mature tracking runs.
- *track_start* : Used to kick off the tracking algorithm by starting the tracking timer.
- *track_stop* : Used to stop the tracking algorithm. A counter-part of *track_start*.
- *track_read* : Used to read out hardware(HW) raw object outputs.
- *track_run* : Used to kick out a tracking update for the current frame.

Data interfaces are wrapped inside the struct `track_t`. In general, the fields can be divided into two categories :

- Interfaces to the outside of tracking system
- Interfaces to the underlying algorithm

Interfaces to the outside world can change more frequently from version to version. Currently, it includes:

[1] Interfaces can be different from version to version. However, the basic principles are kept as same.

Table 3.1: Interfaces to Outside of Tracking System

Name	Description
f_last_time	time stamp of last frame
f_now	time stamp of current frame
noise_factor	noise factor coming baseband
vel_wrap_max	dynamically allocate velocity index to positive
cdi_pkg	raw data read from HW with conversion

Interfaces to the underlying algorithm are more stable. In principle, once these interfaces have been implemented and hooked up to the tracking system properly, the tracking algorithm is able to running smoothly with the rest of the firmware:

Table 3.2: Interfaces to Underlying Algorithm

Name	Description
output_obj	The algorithm updates it for object data output.
output_hdr	The algorithm updates it for header data output.
trk_data	Tracking algorithm specific data
trk_init	Algorithm init handler
trk_pre	Algorithm pre-run handler
trk_run	Algorithm run handler
trk_post	Algorithm post-run handler
trk_stop	Algorithm stop handler
trk_has_run	Indicates whether it is the first time to run algorithm.
trk_set_frame_int	Sets the frame interval for algorithm to run.
trk_update_obj_info	The handler to update “output_obj”
trk_update_header	The handler to update “output_hdr”

3.5.2.2 How to Hook up Your Own Tracking

1. The first step of developing a new tracking algorithm is to implement interfaces listed in Table 3.2.

   If delving into `track.c`, one can see a lot of abstract implementation as following:

```
void track_run(track_t *track)
{
     void* sys_params = (void *)&(CONTAINER_OF(track, baseband_t, track)->sys_
↪params);
     /* pre run */
     if (track->trk_pre)
             track->trk_pre(track->trk_data, sys_params);
     track_update_time(track);
     /* run */
     if (track->trk_run)
             track->trk_run(track->trk_data, sys_params);
     /* output */
     track_run_print(track);
     /* post run */
     if (track->trk_post)
             track->trk_post(track->trk_data, sys_params);
}
```

   As we see, it does nothing but call the handlers that are supposed to fulfill your algorithm.

2. The second step is to hook up your algorithm with the tracking system.

   To achieve this, one should create a function, say `install_YYY_track`, which will pass your handlers and data to the `track_t` struct. For example,

```
void install_ekf_track(track_t *track)
{
        track_trk_pkg_t* trk_pkg = &trk_internal;
        trk_pkg->raw_input = &track->cdi_pkg;
        trk_pkg->output_obj = &track->output_obj;
        trk_pkg->output_hdr = &track->output_hdr;
        track->trk_data = trk_pkg;
        track->trk_init = func_track_init;
        track->trk_pre = NULL;
        track->trk_run = func_track_run;
        track->trk_post = func_track_post;
        track->trk_stop = func_track_stop;
        track->trk_set_frame_int = func_set_frame_int;
        track->trk_update_obj_info = func_track_obj_info_update;
        track->trk_update_header = func_track_header_update;
        track->trk_has_run = func_has_run;
}
```

3.5.2.3 Summary

Interfaces inside `track.h` plays a major role in bridging the underlying algorithm with the rest of firmware, which is summarized in Fig. 3.9

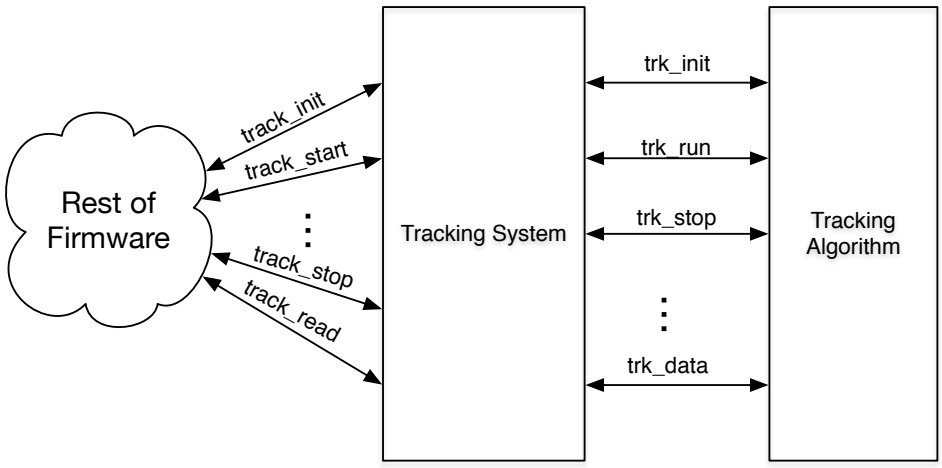

Fig. 3.9: Summary of Tracking System Interfaces

APPENDIX

A

CHIP APIS

Function Name:	clock_select_source	
Syntax:	int32_t clock_select_source(clock_source_t clk_src, clock_source_t sel)	
Parameters (in):	clk_src	The clock node that needs to select its clock source.
	sel	The clock source that is going to be selected.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Select the clock source for the indicated clock node.	

Function Name:	clock_divider	
Syntax:	int32_t clock_divider(clock_source_t clk_src, uint32_t div)	
Parameters (in):	clk_src	Clock divider whose output frequency is going to be divided.
	div	Division factor.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Divide the output frequency of the indicated clock divider.	

Function Name:	**clock_frequency**	
Syntax:	int32_t clock_frequency(clock_source_t clk_src)	
Parameters (in):	clk_src	Clock node whose output frequency is going to return.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code (<= 0) or the frequency value (> 0).	
Description:	Get the output frequency of the indicated clock node.	

BIBLIOGRAPHY

[Anderson1979] 2. Anderson and J. Moore, *Optimal Filtering* 1979.

APPENDIX

B

BOARD API

Function Name:	**can_get_baud**	
Syntax:	can_baud_t *can_get_baud(uint32_t ref_clock, uint32_t baud_rate)	
Parameters (in):	ref_clock	The reference clock of the CAN.
	baud_rate	The baud rate of the CAN bus.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	A pointer to the baud configuration of the indicated baud rate.	
Description:	Get the baud configuration of the indicated baud rate.	

APPENDIX

C

PERIPHERAL APIS

This section gives descriptions of the following categories of APIs in sequence.

- GPIO Driver APIs
- DMA Driver APIs
- SPI Driver APIs
- NOR Flash APIs

Function Name:	**gpio_init**
Syntax:	int32_t gpio_init(void)
Parameters (in):	
Parameters (out):	None
Parameters (in/out):	None
Return value:	Error code.
Description:	Initialize the GPIO controller and the GPIO driver.

Function Name:	**gpio_set_direct**	
Syntax:	int32_t gpio_set_direct(uint32_t gpio_no, uint32_t dir)	
Parameters (in):	gpio_no	GPIO logic number.
	dir	The GPIO direction.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Set the direction of the indicated GPIO.	

Function Name:	gpio_write	
Syntax:	int32_t gpio_write(uint32_t gpio_no, uint32_t level)	
Parameters (in):	gpio_no	GPIO logic number.
	level	The electrical level of the indicated GPIO.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Output the indicated electrical level of indicated GPIO.	

Function Name:	gpio_read	
Syntax:	int32_t gpio_read(uint32_t gpio_no)	
Parameters (in):	gpio_no	GPIO logic number.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code or the electrical level of the indicated GPIO.	
Description:	Get the indicated electrical level of indicated GPIO.	

Function Name:	gpio_int_register	
Syntax:	int32_t gpio_int_register(uint32_t gpio_no, callback func, gpio_int_active_t type)	
Parameters (in):	gpio_no	GPIO logic number.
	func	The Interrupt callback.
	type	The interrupt type.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Enable the interrupt feature of the indicated GPIO, and register the interrupt callback.	

Function Name:	gpio_int_unregister	
Syntax:	int32_t gpio_int_unregister(uint32_t gpio_no)	
Parameters (in):	gpio_no	GPIO logic number.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error cod.	
Description:	Disable the interrupt feature of the indicated GPIO, and uninstall its interrupt callback.	

Function Name:	dma_init
Syntax:	int32_t dma_init(void)
Parameters (in):	None
Parameters (out):	None
Parameters (in/out):	None
Return value:	Error code.
Description:	Initialize the DMA controller and the DMA driver.

Function Name:	dma_apply_channel
Syntax:	int32_t dma_apply_channel(void)
Parameters (in):	None
Parameters (out):	None
Parameters (in/out):	None
Return value:	Error code or the channel ID.
Description:	Apply a DMA logic channel.

Function Name:	**dma_config_channel**	
Syntax:	int32_t dma_config_channel(uint32_t chn_id, dma_trans_desc_t *desc, uint32_t int_flag)	
Parameters (in):	chn_id	DMA channel logic ID.
	*desc	The transfer parameters.
	int_flag	Indicate whether to enable the transfer completion interrupt.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code	
Description:	Configure the indicated DMA channel.	

Function Name:	dma_start_channel	
Syntax:	int32_t dma_start_channel(uint32_t chn_id, dma_chn_callback func)	
Parameters (in):	chn_id	DMA channel logic ID.
	func	The interrupt callback of the transfer completion interrupt.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code	
Description:	Start a DMA transfer.	

Function Name:	**dma_release_channel**	
Syntax:	int32_t dma_release_channel(uint32_t chn_id)	
Parameters (in):	chn_id	DMA channel logic ID.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code	
Description:	Release the indicated DMA channel.	

Function Name:	dma_memcopy	
Syntax:	int32_t dma_memcopy(uint32_t *dst, uint32_t *src, uint32_t len, dma_chn_callback func)	
Parameters (in):	*dst	A pointer to the source data.
	*src	A pointer to the target memory.
	Len	Data length.
	func	The interrupt callback. If the caller call it, the DMA driver can confirm with the caller after the completion of the DMA transfer.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code	
Description:	Copy data from the source to the target memory.	

Function Name:	spi_open	
Syntax:	int32_t spi_open(uint32_t id, uint32_t baud)	
Parameters (in):	id	SPI logic ID.
	baud	The baud rate of the SPI interface.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Initialize the indicated SPI controller.	

Function Name:	spi_device_mode	
Syntax:	int32_t spi_device_mode(uint32_t id)	
Parameters (in):	id	SPI logic ID.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Get the mode of the indicated SPI (master or slave).	

Function Name:	spi_transfer_config	
Syntax:	int32_t spi_transfer_config(uint32_t id, spi_xfer_desc_t *desc)	
Parameters (in):	id	SPI logic ID.
	*desc	The transfer parameters.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Configure the SPI transfer, such as data frame size and frame format.	

Function Name:	spi_xfer	
Syntax:	int32_t spi_xfer(uint32_t id, uint32_t *txdata, uint32_t *rxdata, uint32_t len)	
Parameters (in):	id	SPI logic ID.
	*txdata	A pointer to a buffer where the to-be-sent data are.
	*rxdata	A pointer to a buffer where the received data will be stored.
	len	Transfer data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Start a full-complex transfer.	

Function Name:	spi_write	
Syntax:	int32_t spi_write(uint32_t id, uint32_t *data, uint32_t len)	
Parameters (in):	id	SPI logic ID.
	*data	A pointer to a buffer where the to-be-sent data are.
	len	Transfer data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Start a semi-complex transfer (send data).	

Function Name:	spi_read	
Syntax:	int32_t spi_read(uint32_t id, uint32_t *data, uint32_t len)	
Parameters (in):	id	SPI logic id.
	*data	A pointer to a buffer where the received data will be stored.
	len	Transfer data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Start a semi-complex transfer (read data).	

Function Name:	spi_databuffer_install	
Syntax:	int32_t spi_databuffer_install(uint32_t id, uint32_t rx_or_tx, DEV_BUFFER *devbuf)	
Parameters (in):	id	SPI logic ID.
	Rx_or_tx	Indicate that the device buffer is used for transmitting or receiving. 0, receiving. 1, transmitting.
	*devbuf	Device buffer including a pointer to the memory and buffer size.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Install a device buffer to SPI driver for receiving or transmitting.	

Function Name:	spi_databuffer_uninstall	
Syntax:	int32_t spi_databuffer_uninstall(uint32_t id, uint32_t rx_or_tx)	
Parameters (in):	id	SPI logic ID.
	Rx_or_tx	Indicate that the device buffer is used for transmitting or receiving. 0, receiving. 1, transmitting.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Uninstall the receiving or transmitting device buffer.	

Function Name:	spi_interrupt_install	
Syntax:	int32_t spi_interrupt_install(uint32_t id, uint32_t rx_or_tx, void (*func)(void *))	
Parameters (in):	id	SPI logic ID.
	Rx_or_tx	Indicate that the device buffer is used for transmitting or receiving. 0, receiving. 1, transmitting.
	func	The interrupt callback.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Register the interrupt callback to the SPI driver for receiving or transmitting, and enable the interrupt feature.	

Function Name:	spi_interrupt_uninstall	
Syntax:	int32_t spi_interrupt_uninstall(uint32_t id, uint32_t rx_or_tx)	
Parameters (in):	id	SPI logic ID.
	Rx_or_tx	Indicate that the device buffer is used for transmitting or receiving. 0, receiving. 1, transmitting.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Disable the interrupt feature, and uninstall the interrupt callback for receiving or transmitting.	

Function Name:	**spi_interrupt_enable**	
Syntax:	int32_t spi_interrupt_enable(uint32_t id, uint32_t rx_or_tx, uint32_t enable)	
Parameters (in):	id	SPI logic id.
	Rx_or_tx	Indicate the device buffer is used for transmitting or receiving. 0, receiving. 1, transmitting.
	enable	Indicate whether to enable interrupt or not.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Enable or disable interrupt for receiving or transmitting.	

Function Name:	**flash_init**
Syntax:	int32_t flash_init(void)
Parameters (in):	None
Parameters (out):	None
Parameters (in/out):	None
Return value:	Error code.
Description:	Initialize the NOR flash driver.

Function Name:	**flash_reset**
Syntax:	int32_t flash_reset(void);
Parameters (in):	None
Parameters (out):	None
Parameters (in/out):	None
Return value:	Error code.
Description:	Reset the external NOR flash device.

Function Name:	flash_memory_readb	
Syntax:	int32_t flash_memory_readb(uint32_t addr, uint8_t *data, uint32_t len)	
Parameters (in):	Addr	The NOR flash memory address.
	*data	A pointer to a buffer where the received data will be stored.
	len	Data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Read data from the indicated NOR flash memory space.	

Function Name:	flash_memory_read	
Syntax:	int32_t flash_memory_read(uint32_t addr, uint32_t *data, uint32_t len)	
Parameters (in):	Addr	The NOR flash memory address.
	*data	A pointer to a buffer where the received data will be stored.
	len	Data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Read data from the indicated NOR flash memory space.	

Function Name:	flash_memory_writeb	
Syntax:	int32_t flash_memory_writeb(uint32_t addr, uint8_t *data, uint32_t len)	
Parameters (in):	Addr	The NOR flash memory address.
	*data	A pointer to a buffer where the to-be-sent data are.
	len	Data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Program the data to the indicated NOR flash memory space.	

Function Name:	flash_memory_write	
Syntax:	int32_t flash_memory_write(uint32_t addr, uint8_t *data, uint32_t len)	
Parameters (in):	Addr	The NOR flash memory address.
	*data	A pointer to a buffer where the to-be-sent data are.
	len	Data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Program the data to the indicated NOR flash memory space.	

Function Name:	Flash_program_resume
Syntax:	int32_t flash_program_resume(void)
Parameters (in):	None
Parameters (out):	None
Parameters (in/out):	None
Return value:	Error code.
Description:	Resume the suspended program progress.

Function Name:	flash_memory_erase	
Syntax:	int32_t flash_memory_erase(uint32_t addr, uint32_t len)	
Parameters (in):	Addr	The NOR flash memory address.
	len	Data length.
Parameters (out):	None	
Parameters (in/out):	None	
Return value:	Error code.	
Description:	Erase the indicated NOR flash memory space.	

APPENDIX

D

REVISION HISTORY

Table D.1: Revision History

Revision	Description	Author
1.0/ April 2019	Initial Release.	pwang
1.1/ February 2020	Major updates throughout the document.	pwang

INDEX