Contents

Chapter 1. PikoRT

This is Piko/RT, a tiny Linux-like real-time operating system kernel, optimized for ARM Cortex-M series microprocessors [1].

Chapter 2. Task and Thread

Process be presented as task struct in kernel, whose data struct is as Figure 2.1.

```
struct task_info {
    pid_t pid;
    unsigned long filemap;
    struct file filetable[FILE_MAX];
    struct list_head list;
    struct list_head thread_head;
    struct list_head signal_head; /* list of installed handlers */
};
```

Figure 2.1

Pid field is process id; Filemap field is file description; Thread_head field record all threads in this process; Signal_head record all signal this process receive.

When user want to launch a program. In addition to create task, kernel will also create main thread for executing the program main function. Figure 2.2 is thread_create() description. Start_routine parameter is thread will execute function; arg parameter is strart_routine function's parameter; priv parameter means thread privilege; stacksize parameter indicates process user mode stack size; task parameter is the task this thread belong to.

```
struct thread_info *thread_create(void *(*start_routine)(void *),
                                  void *arg,
                                  enum thread_privilege priv,
                                  size_t stacksize,
                                  struct task_info *task)
{
```

Figure 2.2

Figure 2.3 is state machine of thread statement. Thread is THREAD_STATE_NEW when creating a thread; system add thread into scheduling via sched_enqueue(). In turn, marking thread statement as THREAD_STATE_READY; Piko-RT use bitmap scheduling to decide which thread be launched. In turn, marking thread statement as THREAD_STATE_RUNNING; When active thread needs to wait or lock in order to get some resource. Kernel will marking now thread state as

THREAD_STATE_BLOCKED, and select a new thread to execute; Blocked thread change statement to THREAD_STATE_READY until getting resource.

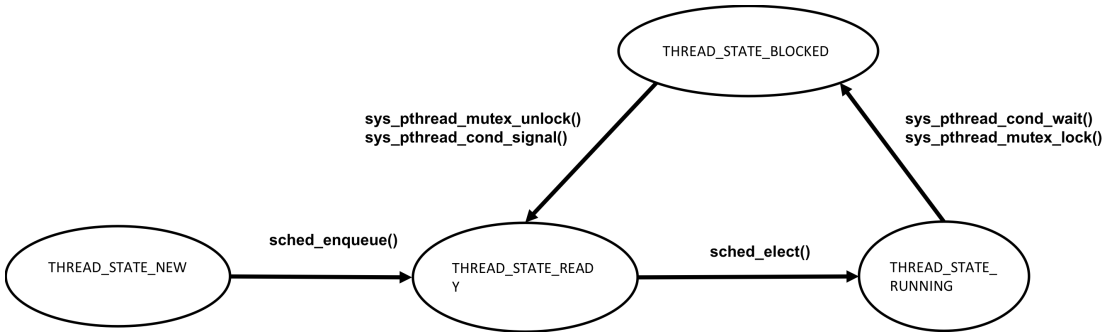

Figure 2.3

When sys_pthread_mutex_unlock() or sys_pthread_cond_signal() be called, kernel check whether another thread wait the mutex or condition. If it is, kernel, in turn, check if the thread's priority is greater. If it is higher, kernel activate the thread.

Chapter 3. Inter-Process Communication

Piko-RT kernel does not support IPC mechanism.

Chapter 4. Scheduling

Kernel introduces bitmap scheduling. Bitmap scheduling maintains two bitmap_struct, active and expire.

Bimamp_struct has two members, map and queue, Figure 4.1. Map has 32 bits, each bit represents a priority, and queue represents 32 scheduling queues. For example, kernel want to add a thread whose priority is 7 into scheduling, kernel will enable 7th bit of map to indicate a thread whose priority is 7 wait CPU resources, and, enqueue the thread into 7th queue according to thread priority.

If time slice is exhaust, kernel will add the thread into expire bitmap and select a new thread from active bitmap to execute. When all threads are expiring, kernel will swap contents of active bitmap and expire one, and do scheduling again.

```
 7 struct bitmap_struct {
 8     unsigned long map;
 9     struct list_head queue[32];
10 };
```

Figure 4.1

```
11 static struct bitmap_struct _active, _expire;
12
13 static struct {
14     struct bitmap_struct *active;
15     struct bitmap_struct *expire;
16 } sched_struct = {
17     .active = &_active,
18     .expire = &_expire,
19 };
```

Figure 4.2

Chapter 5. Softirq

When kernel initialize, kernel will register a softirq, PRIO_TASKLET, and create a softirq daemon kernel thread. When CPU receives a systick exception, systick interrupt handler (top half) crates a systick_bh tasklet and adds into tasklet list, kernel will also adds softirq daemon kernel thread into scheduling. So far, piko-RT only support systick softirq.

When softeirq daemon kernel thread gets CPU resources, softirq daemon kernel thread will execute all pending softeirq.

Chapter 6. Micro Control Unit – stm32p103

6.1 Introduction

Stm32p103 MCU is a Cortex-M3 processor. Figure 6.1 is STM32 Cortex-M3 implementation.

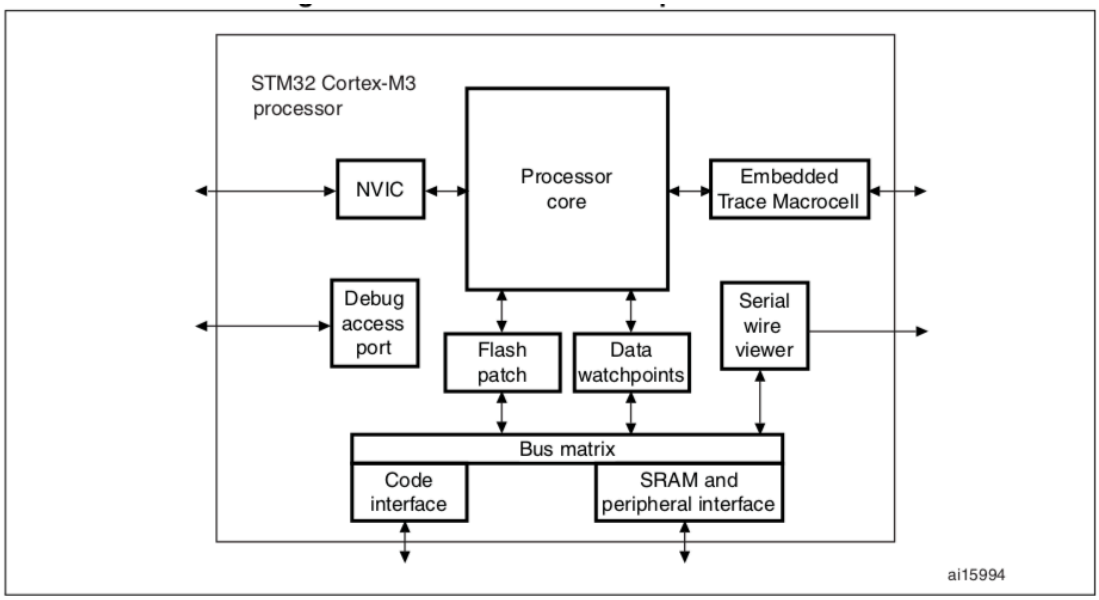

Figure 6.1

6.2 Registers

The Cortex-M3 processor has registers R0 to R15. R13 (the stack pointer) is banked, with only one copy of the R13 visible at a time.

R0 to R12: General-Purpose Registers

R0 to R12 are 32-bit general-purpose registers for data operations. Some 16-bit Thumb instructions can only access a subset of these registers (low registers, R0 to R7).

R13: Stack Pointers

The Cortex-M3 contains two stack pointers, R13. They are banked so that only one is visible at a time:

• Main Stack Pointer (MSP): The default stack pointer; used by the OS kernel and exception handlers

• Process Stack Pointer (PSP): Used by user application code

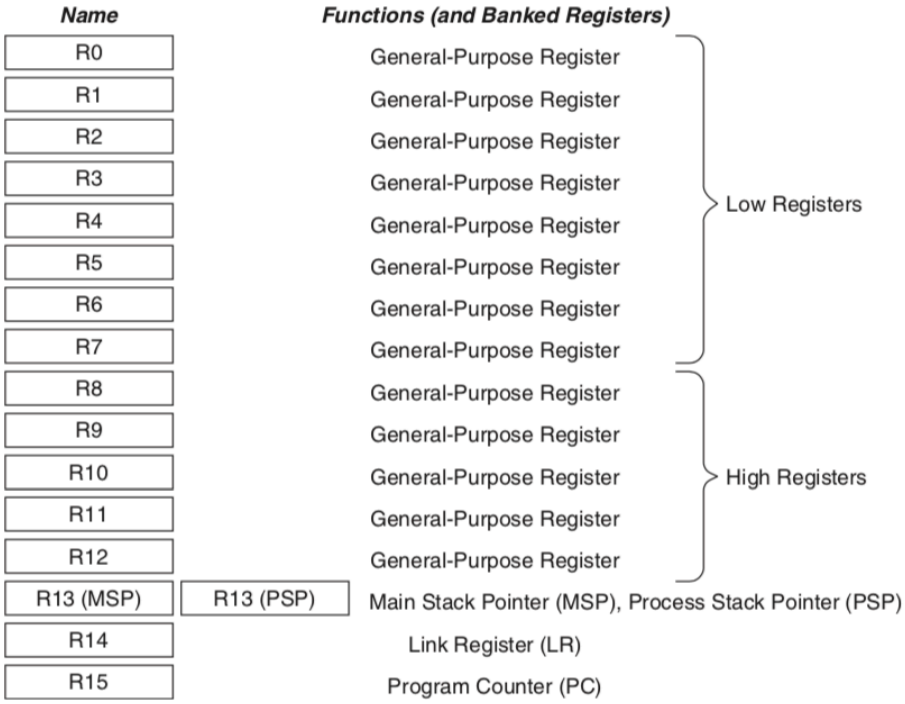

Figure 6.2

6.3 Exceptions

The number of external interrupt inputs is defined by chip manufacturers. A maximum of 240 external interrupt inputs can be supported, Figure 6.3.

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (Prefetch Abort or Data Abort)
6	Usage fault	Programmable	Exceptions due to program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	System service call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
...	...	...	...
255	IRQ #239	Programmable	External interrupt #239

Figure 6.3

Interrupt/Exception Sequences

When an exception takes place, a number of things happen:

- Stacking (pushing eight registers' contents to stack)
- Vector fetch (reading the exception handler starting address from the vector table)
- Update of the stack pointer, link register, and program counter

Figure 6.4 and Figure 6.5 are operation mode transitions when exception enter or exit.

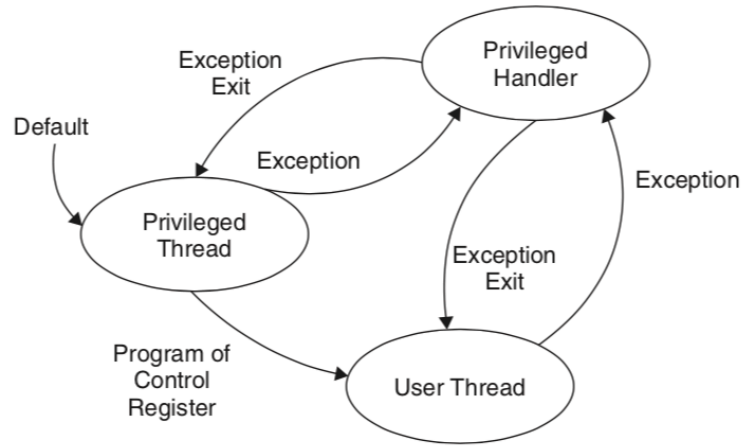

Figure 6.4

	Privileged	*User*
When running an exception	Handle Mode	
When running main program	Thread Mode	Thread Mode

Figure 6.5

Stacking

When an exception takes place, the registers PC, PSR, R0–R3, R12, and LR are pushed to the stack. If the code that is running uses the PSP, the process stack will be used; if the code that is running uses the MSP, the main stack will be used. Afterward, the main stack will always be used during the handler, so all nested interrupts will use the main stack.

The order of stacking is shown in Figure 6.6(assuming that the SP value is N before the exception).

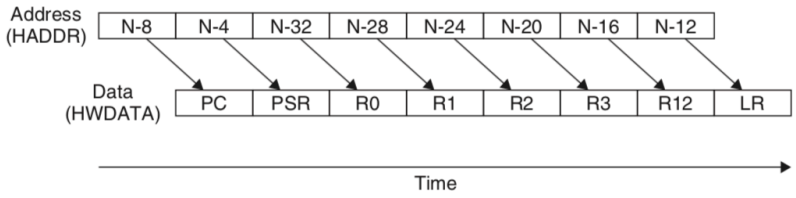

Figure 6.6

The values of PC and PSR are stacked first so that instruction fetch can be started early (which requires modification of PC) and the IPSR can be updated early. After stacking, SP will be updated to N-32 (0 x 20), and the stacked data arrangement

in the stack memory will look like Figure 6.7.

Address	Data	Push Order
Old SP (N) ->	(Previously pushed data)	-
(N-4)	PSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
New SP (N-32) ->	R0	3

Figure 6.7 Stack Memory Content After Stacking and Stacking Order

The reason the registers R0–R3, R12, LR, PC, and PSR are stacked is that these are caller saved registers.

The general registers (R0–R3, R12) are located at the end of the stack frame so that they can be easily accessed using SP-related addressing. As a result, it's easy to pass parameters to software interrupts using stacked registers.

Vector Fetches

While the data bus is busy stacking the registers, the instruction bus carries out another important task of the interrupt sequence: It fetches the exception vector (the starting address of the exception handler) from the vector table. Since the stacking and vector fetch are performed on separate bus interfaces, they can be carried out at the same time.

Register Updates

After the stacking and vector fetch are completed, the exception vector will start to execute. On entry of the exception handler, a number of registers will be updated:

• SP: The Stack Pointer (either the MSP or the PSP) will be updated to the new location during stacking. During execution of the interrupt service routine, the MSP will be used if the stack is accessed.

• PSR: The IPSR (the lowest part of the PSR) will be updated to the new exception number.

• PC: This will change to the vector handler as the vector fetch completes and

starts fetching instructions from the exception vector.

• LR: The LR will be updated to a special value called EXC_RETURN.1 This special value drives the interrupt return operation. The last 4 bits of the LR have a special meaning, which is covered later in this chapter.

Exception Exits

When entering an exception handler, the LR is updated to a special value called EXC_RETURN, with the upper 28 bits all set to 1. This value, when loaded into the PC at the end of the exception handler execution, will cause the processor to perform an exception return sequence.

The instructions that can be used to generate exception returns are:

• POP/LDM

• LDR with PC as a destination

• BX with any register

The EXC_RETURN value has bit [31:4] all set to 1, and bit[3:0] provides information required by the exception return operation (see Figure 6.8). When the exception handler is entered, the LR value is updated automatically, so there is no need to generate these values manually.

Bits	31:4	3	2	1	0
Descriptions	0xFFFFFFF	Return mode (Thread/handler)	Return stack	Reserved; must be 0	Process state (Thumb/ARM)

Figure 6.8

Bit 0 indicates the process state being used after the exception return. Since the Cortex-M3 supports only the Thumb state, bit 0 must be 1.

The valid values (for the Cortex-M3) are shown in Figure 6.9.

Value	Condition
0xFFFFFFF1	Return to handler mode
0xFFFFFFF9	Return to Thread mode and on return use the main stack
0xFFFFFFFD	Return to Thread mode and on return use the process stack

Figure 6.9

If the thread is using the MSP (main stack), the value of LR will be set to 0xFFFFFFF9 when it enters an exception, and 0xFFFFFFF1 when a nested exception is entered. If the thread is using PSP (process stack), the value of LR would be 0xFFFFFFFD when entering the first exception and 0xFFFFFFF1 for entering a nested exception.

6.4 Thumb State

Versions 4T and 4TxM of the ARM architecture define a 16-bit instruction set called the Thumb instruction set. The functionality of the Thumb instruction set is a subset of the functionality of the 32-bit ARM instruction set.

The Thumb instruction set:

- imposes some limitations on register access
- does not allow conditional execution except for branch instructions
- does not allow access to the barrel shifter except as a separate instruction.

A processor that is executing Thumb instructions is said to be operating in Thumb state. A Thumb-capable processor that is executing ARM instructions is said to be operating in ARM state.

ARM processors always start in ARM state. You must explicitly change to Thumb state using a BX (Branch and exchange instruction set) instruction.

6.5 Memory Map

The Cortex-M3 has a predefined memory map. This allows the built-in peripherals, such as the interrupt controller and debug components, to be accessed by simple memory access instructions. Thus most system features are accessible in C program code. The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC)

designs.

Overall, the 4 GB memory space can be divided into the ranges shown in Figure 6.10.

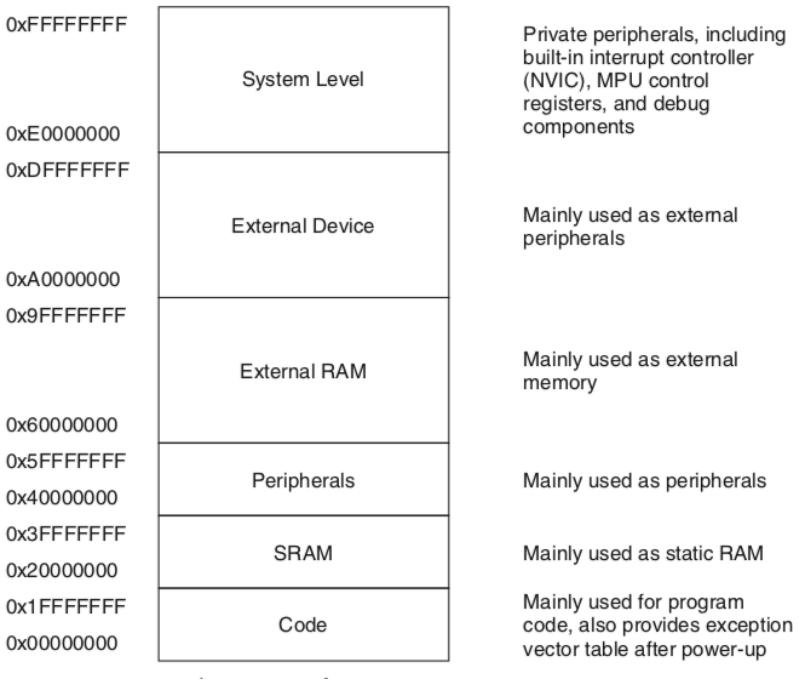

Figure 6.10

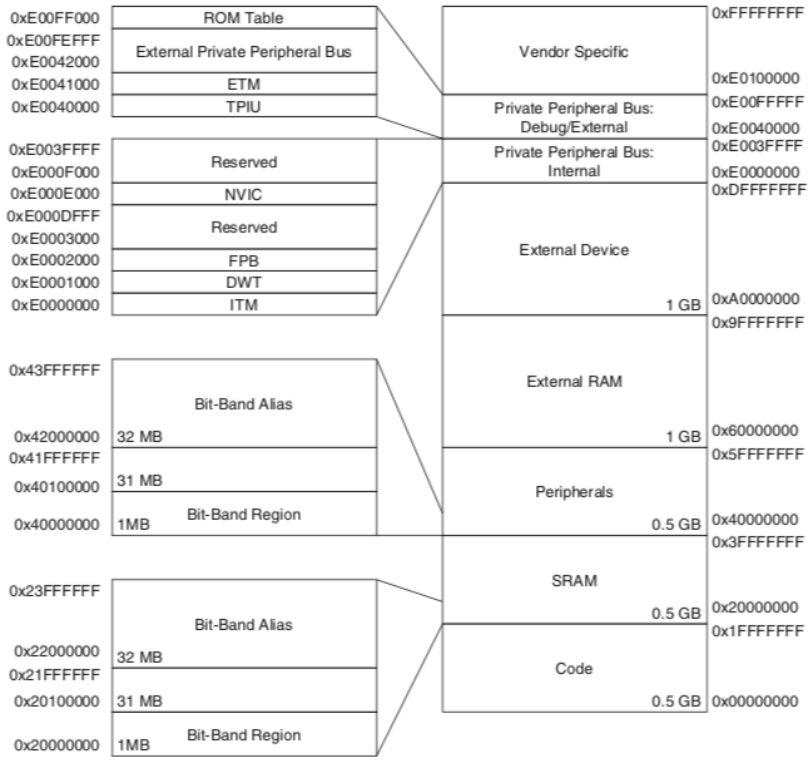

Figure 6.11

6.6 Nested vectored interrupt controller (NVIC)

The Cortex-M3 processor includes an interrupt controller called the Nested Vectored Interrupt Controller (NVIC). It is closely coupled to the processor core and provides a number of features:

- Nested interrupt support
- Vectored interrupt support
- Dynamic priority changes support
- Reduction of interrupt latency
- Interrupt masking

Nested Interrupt Support

The NVIC provides nested interrupt support. All the external interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.

Vectored Interrupt Support

The Cortex-M3 processor has vectored interrupt support. When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory. There is no need to use software to determine and branch to the starting address of the ISR. Thus it takes less time to process the interrupt request.

Dynamic Priority Changes Support

Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the interrupt service routine is completed, so their priority can be changed without risk of accidental reentry.

Reduction of Interrupt Latency

The Cortex-M3 processor also includes a number of advanced features to lower the interrupt latency. These include automatic saving and restoring some register contents, reducing delay in switching from one ISR to another, and handling late arrival interrupts.

Interrupt Masking

Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK, and FAULTMASK. They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E11C	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	*Interrupt Set-enable Registers* on page 4-4
0XE000E180-0xE000E19C	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	*Interrupt Clear-enable Registers* on page 4-5
0XE000E200-0xE000E21C	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	*Interrupt Set-pending Registers* on page 4-5
0XE000E280-0xE000E29C	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	*Interrupt Clear-pending Registers* on page 4-6
0xE000E300-0xE000E31C	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	*Interrupt Active Bit Registers* on page 4-7
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	*Interrupt Priority Registers* on page 4-7
0xE000EF00	STIR	WO	Configurable[a]	0x00000000	*Software Trigger Interrupt Register* on page 4-8

Figure 6.12

6.7 The Cortex Microcontroller Software Interface Standard

For a Cortex-M3 microcontroller system, the Cortex Microcontroller Software Interface Standard (CMSIS) defines:

- a common way to:
  - — access peripheral registers
  - — define exception vectors
- the names of:
  - — the registers of the core peripherals
  - — the core exception vectors

- a device-independent interface for RTOS kernels, including a debug channel.

The CMSIS includes address definitions and data structures for the core peripherals in the Cortex-M3 processor. CMSIS simplifies software development by enabling the reuse of template code and the combination of CMSIS-compliant software components from various middleware vendors. Software vendors can expand the CMSIS to include their peripheral definitions and access functions for those peripherals. This document includes the register names defined by the CMSIS, and gives short descriptions of the CMSIS functions that address the processor core and the core peripherals.

Chapter 7. Reference

[1] PioRT. https://github.com/PikoRT/pikoRT

[2] STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual.pdf

[3] ARM infocenter. http://infocenter.arm.com/help/index.jsp

[4] Definitive_Guide_To_The_ARM_Cortex_M3.pdf