5 Interrupts

Contents

5.2

Introduction

An interrupt is a request by another module for access to CPU processing time

You are studying at your desk at home. The phone rings (an *interrupt*). You stop studying and answer the phone (you accept the interrupt). It is your friend, who wants to know the URL for a particular Freescale datasheet relating to the K70 so she can look up some information required to complete a laboratory assignment. You give her the URL (you process the interrupt request immediately). You then hang up and go back to studying. Note that the additional time it will take you to complete your study is miniscule, yet the amount of time for your friend to complete her task may be significantly reduced (she didn't have to wait until you were free). This simple example clearly illustrates how interrupts can drastically improve response time in a real-time system.

Interrupts are essential features of real-time systems

Interrupts are an essential feature of a microcontroller. They enable the software to respond, in a timely fashion, to internal and external hardware events. For example, the reception and transmission of bytes via the UART is more efficient (in terms of processor time) using interrupts, rather than using a polling method. Performance is improved because tasks can be given to hardware modules which "report back" when they are finished.

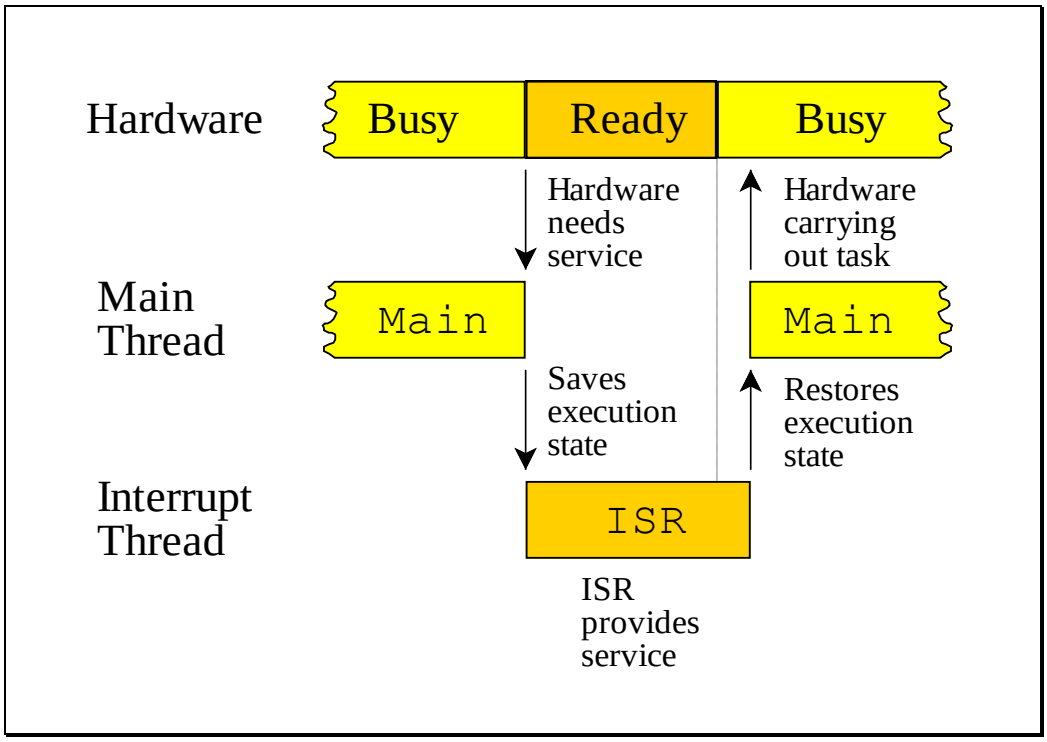
Using interrupts requires that we first understand how a CPU processes an interrupt so that we can configure our software to take advantage of them.

5.1 Exceptions

Exceptions are events that cause changes to program flow. When one happens, the processor suspends the current executing task and executes a part of the program called an *exception handler*. After the execution of the exception handler is completed, the processor then resumes normal program execution. In the ARM® architecture, interrupts are one type of exception.

5.2 Interrupts

An *interrupt* is an event triggered inside the microcontroller, usually by internal or external hardware, and in some cases by software. The exception handler for an interrupt is referred to as an *interrupt service routine* (ISR). On completion of the ISR, software execution returns to the next instruction that would have occurred without the interrupt.



An interrupt causes the main thread to be suspended, and the interrupt thread is run

Figure 5.1

A *thread* is defined as a sequence of instructions that has its own program counter, stack and registers; it shares its address space and system resources with other threads. By contrast, a *process* has its own virtual address space (stack, data, code) and system resources (e.g. open files). Processes are normally used in systems with an operating system, whereas threads are easily implemented in simple embedded systems using interrupt service routines.

In the K70 microcontroller, the hardware automatically pushes the contents of most of the internal registers onto the stack, thus creating the correct environment for a new thread invoked by an ISR.

5.2.1 Using Interrupts

Each potential interrupt source has a separate *arm* bit, e.g. `RIE` (the UART receive interrupt enable bit). The software must set the arm bits for those devices from which it wishes to accept interrupts, and deactivate the arm bits within those devices from which interrupts are not to be allowed. After reset, all the interrupt arm bits are set to deactivate the corresponding interrupt.

Each potential interrupt source has a separate *flag* bit, e.g. `RDRF` (the UART receive data register full flag). The hardware sets the flag when it wishes to request an interrupt. The software must clear the flag in the ISR to signify it has handled the interrupt request, and to allow the device to again trigger an interrupt.

There are a number of special registers in the MCU that contain the processor status and define the operation states and interrupt/exception masking. Special registers are not memory mapped, which means special assembly language instructions are required to access them.

The `PRIMASK` register is used for exception or interrupt masking. It is a 1-bit wide interrupt mask register. When set, it blocks all exceptions (including interrupts) apart from the Non-Maskable Interrupt (NMI) and the HardFault exception.

Software enables all armed interrupts by setting `PRIMASK = 0`, (“`__asm("CPSIE i");`” in C), and disables all interrupts by setting `PRIMASK = 1` (“`__asm("CPSID i");`” in C). `PRIMASK = 1` does not dismiss the interrupt requests, rather it postpones them.

Four conditions must be true simultaneously for an interrupt to occur:

- The hardware peripheral’s interrupt arm bit must be set (by software).
- The hardware peripheral’s interrupt flag must be set (by hardware).
- The interrupt has a higher priority than any executing ISR, and the `PRIMASK` register is 0 (interrupts are enabled).
- The interrupt source is enabled in the Nested Vectored Interrupt Controller (NVIC).

The following figure shows the hardware arrangement for interrupt generation.

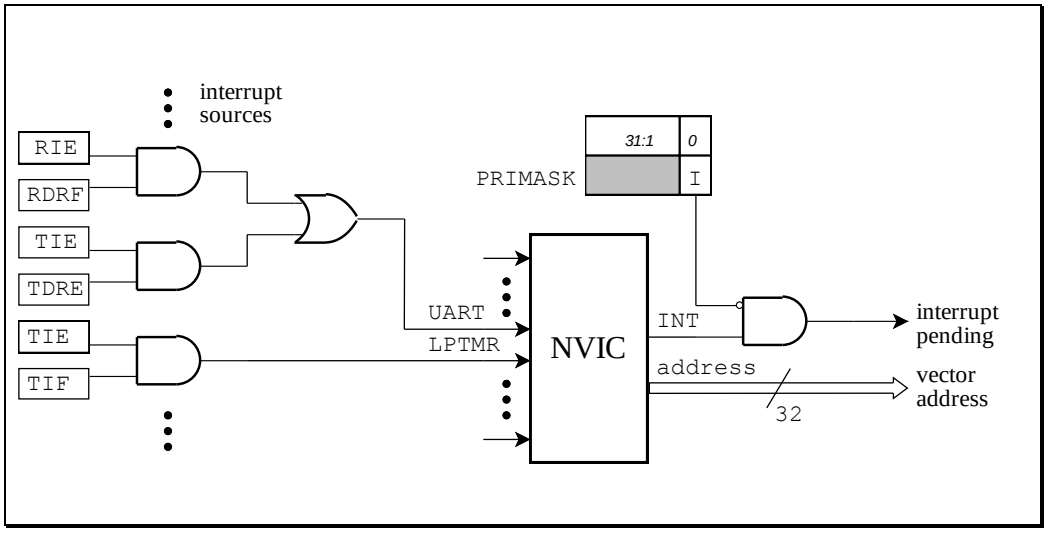


Figure 5.2

5.2.2 Interrupt Processing

When an interrupt occurs, the following sequence is followed.

1. The execution of the main program is suspended by the hardware:
   - the current instruction is finished
   - all the registers are pushed onto the stack
   - the vector address is retrieved from the *vector table* and placed in the program counter, `PC`

2. The interrupt service routine, or foreground thread, is executed. The ISR:
   - clears the flag that requested the interrupt
   - performs the necessary operations
   - communicates with other threads using global variables

3. The main program is resumed when the ISR executes the `EXC_RETURN` instruction:
   - Hardware pulls all the registers from the stack, including the `PC`, so that the program continues from the point where it was interrupted.

5.2.3 Interrupt Polling

Some interrupts share the same interrupt vector. For example, the reception and transmission of a byte via the UART leads to just one interrupt, and there is one vector associated with it. In Figure 5.2, the two interrupt sources are ORed together to create one interrupt request. In such cases, the ISR is responsible for polling the status flags to see which event actually triggered the interrupt. Care must be taken because both flags may be set, and only the hardware events that are enabled must be serviced by the software.

For example, the UART shares an interrupt for transmit and receive operations. Therefore, in the ISR, we would need code to respond to either of those events, but only if the corresponding interrupt enable bit is enabled:

```
...
// Receive a character
if (UART2_C2 & UART_C2_RIE_MASK)
{
  // Clear RDRF flag by reading the status register
  if (UART2_S1 & UART_S1_RDRF_MASK)
    // Do something with the received byte
    ...
}

// Transmit a character
if (UART2_C2 & UART_C2_TIE_MASK)
{
  // Clear TDRE flag by reading the status register
  if (UART2_S1 & UART_S1_TDRE_MASK)
  {
    // Get a new byte and transmit it
   ...
```

Listing 5.1 – Polling the Source of an Interrupt in an ISR

5.3 The Vector Table

When an exception occurs, the processor will need to locate the starting point of the corresponding exception handler. A Cortex®-M processor will automatically locate the starting point of the exception handler from a *vector table* in the memory. The vector table is an array of word data inside the system memory, with each entry in the table giving the starting address of the exception type.

The vector table starts at memory address 0. The first entry is special – it is not an address but the initial value of the stack pointer. It is needed because some exceptions such as the NMI could happen as the processor just comes out of reset and before any other initialization steps are executed.

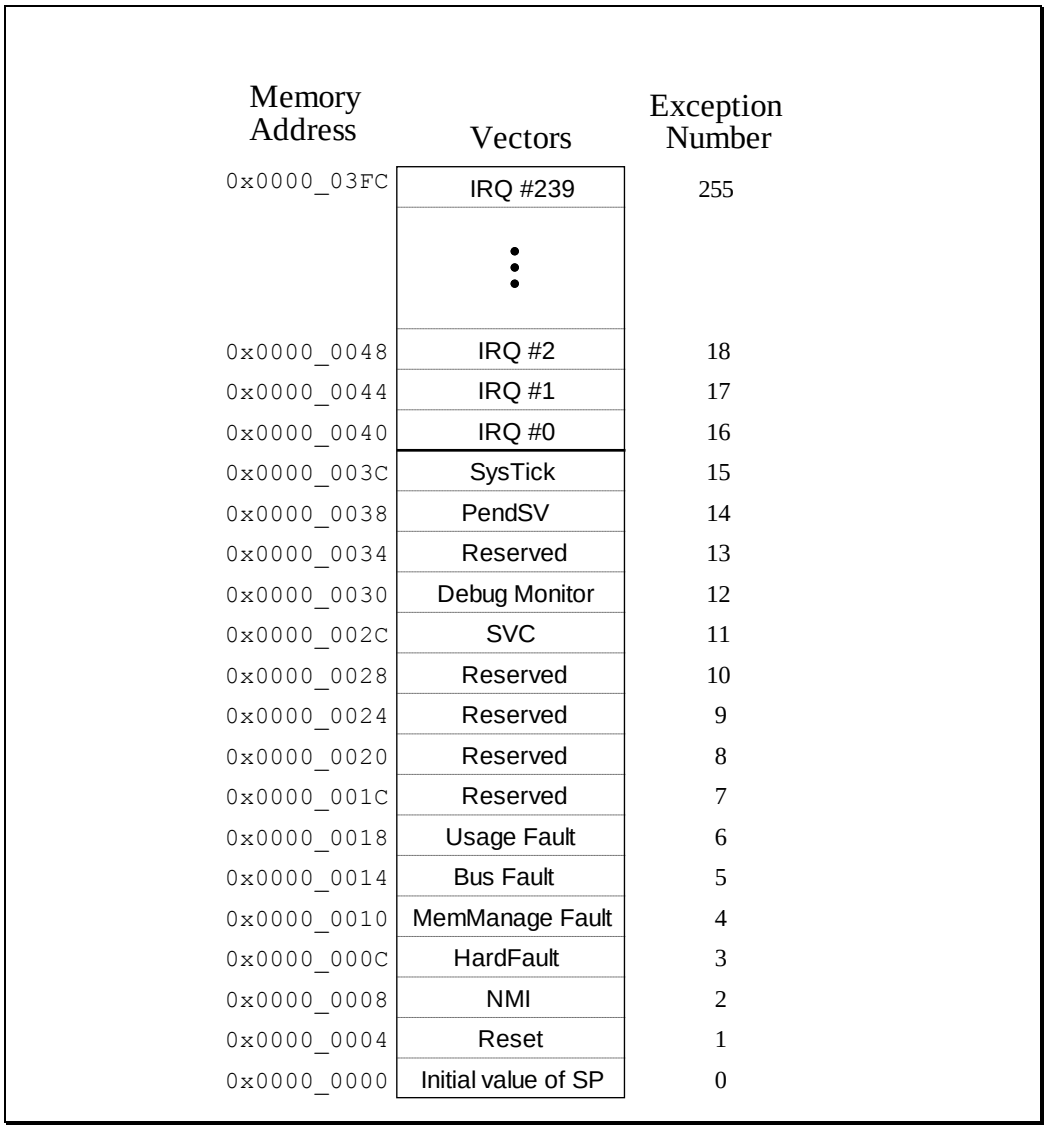


Figure 5.3 – Vector Table

5.4 Interrupt Service Routines (ISRs)

An *interrupt service routine* (ISR) is a section of code specifically designed to respond to the interrupt request. When the CPU begins to service an interrupt, the instruction queue is refilled, a return address calculated, and then the return address and the contents of the CPU registers are automatically stacked as shown below:

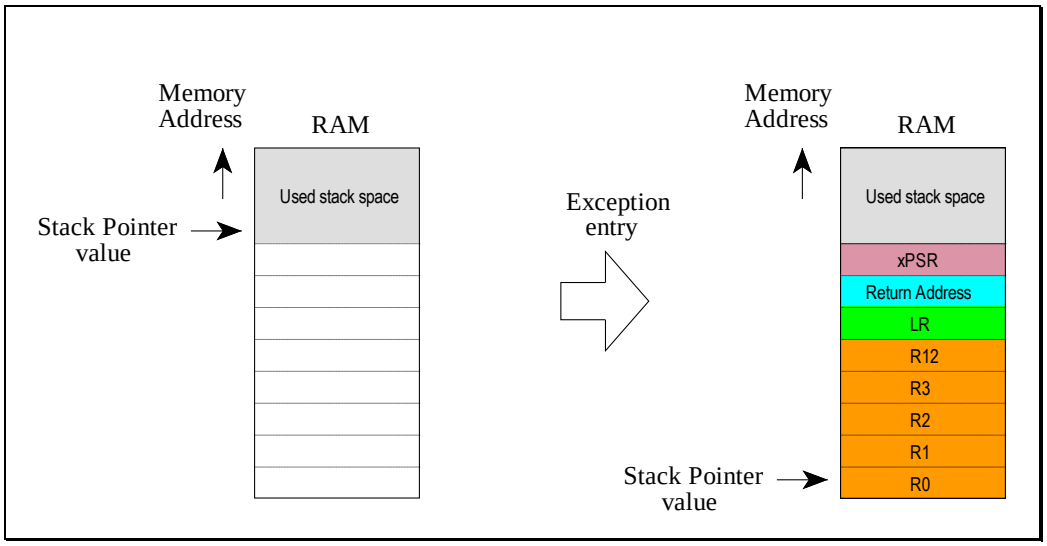


Figure 5.4 – Exception Stack Frame on Entry to Interrupts

A higher priority exception pre-empts a currently executing exception handler – this is called a nested exception

Execution continues at the address pointed to by the vector for the highest-priority interrupt that was pending at the beginning of the interrupt sequence – this is the interrupt service routine. If an interrupt source of higher priority occurs during execution of the ISR, the ISR will itself be interrupted – this is called interrupt *nesting*.

The body of an interrupt service routine varies according to the source of the interrupt. For an interrupt service routine written to handle external events, they typically respond to the interrupt by retrieving or sending external data, e.g. the reception of a byte of data via the UART is normally handled via an ISR which places the received byte into a FIFO for later processing by the `main` function.

At the end of the interrupt service routine, an `EXC_RETURN` instruction restores context from the stacked registers, and normal program execution resumes (which could be recognition of another interrupt of lower priority).

5.4.1 Declaring Interrupt Service Routines in C for ARM® Cortex®-M Processors

In some processor architectures, a special instruction is used for exception return. However, this means that the exception handlers cannot be written and compiled as normal C code. In ARM® Cortex®-M processors, the exception return mechanism is triggered using a special return address called `EXC_RETURN`. This value is generated at exception entrance and is stored in the Link Register (`LR`). The `LR` is a register normally used to hold a return address. When the special value of `LR` is written to the Program Counter (`PC`) it triggers the exception return sequence. When the exception return mechanism is triggered, the processor accesses the previously stacked register values in the stack memory during exception entrance and restores them back to the register bank. This process is called *unstacking*.

The use of the `EXC_RETURN` value for triggering exception returns allows exception handlers (including interrupt service routines) to be written as normal C functions.

5.4.2 Declaring Interrupt Service Routines in C for Generic Processors

In GNU C, you use function attributes to declare certain things about functions called in your program which help the compiler optimize calls and check your code more carefully. You can also use attributes to control memory placement, code generation options or call/return conventions within the function being annotated. Many of these attributes are target-specific. For example, many targets support attributes for defining interrupt handler functions, which typically must follow special register usage and return conventions.

Function attributes are introduced by the `__attribute__` keyword on a declaration, followed by an attribute specification inside double parentheses.

In the GNU Compiler Collection (GCC) for ARM® processors, the function attribute `interrupt` is used to indicate that the specified function is an interrupt service routine. For example, to declare an ISR for a UART, you would use:

```
void __attribute__ ((interrupt)) UART_ISR(void)
{
  /* code goes here */
}
```

The `interrupt` function attribute for the ISR is really only needed for previous generations of ARM® processors, since the Cortex®-M has a special hardware instruction for exception return, as outlined in the previous section.

However, we will still define our ISRs with a function attribute as a matter of *style* – it will make it easier for anyone reading our code to see that the function's intended use is as an ISR.

5.4.3 Specifying an ISR Address in the Vector Table

To place the address of the ISR in the vector table, which is defined in `vectors.c` in the `Generated_Code` folder, we need to find the relevant vector number and insert the address of the ISR in the table. The vector number for a particular interrupt source is given in Table 3-5 of the K70 Sub-Family Reference Manual. For example, if `UART_ISR` is used for UART2's transmit and receive ISR, we would put:

```
...
(tIsrFunc)&Cpu_Interrupt,           /* 0x3F  UART1_RX_TX */
(tIsrFunc)&Cpu_Interrupt,           /* 0x40  UART1_ERR   */
(tIsrFunc)&UART_ISR,                /* 0x41  UART2_RX_TX */
(tIsrFunc)&Cpu_Interrupt,           /* 0x42  UART2_ERR   */
(tIsrFunc)&Cpu_Interrupt,           /* 0x43  UART3_RX_TX */
...
```

Listing 5.2 – Vector Table Extract Showing ISR Address

5.5 Enabling and Disabling Interrupts

Interrupts can be enabled and disabled with the macros defined in the `PE_Types.h` file which you can include by placing `#include "PE_Types.h"` in your main file. The macros are:

```
#define __EI()   __asm("CPSIE f");
#define __DI()   __asm("CPSID f");
```

The assembly language instruction `CPSIE` stands for Change Processor State Interrupt Enable and the `f` parameter refers to the single-bit "fault mask" register `FAULTMASK`. This register is similar to `PRIMASK`, but it also blocks the HardFault exception.

Interrupts are disabled by default on reset, but it is good *style* to disable them before you embark on peripheral module initialization – it acts as a reminder that no interrupts will occur. You should then enable interrupts before the main loop of your program:

```
...
__DI();
TowerInit();
__EI();
while (1)
{
  /* Main loop */
}
```

5.5.1 Interrupt Latency

Interrupts cannot disturb an instruction in progress, and thus are only recognized *between* the execution of two instructions (apart from special instructions on the K70 which are designed to be interrupted). Therefore the maximum latency from interrupt request to completion of the hardware response consists of the execution time of the slowest instruction plus the time required to complete the memory transfers required by the hardware response.

5.6 Interrupt Priority

Each exception (including interrupts) has a priority level where a smaller number is a higher priority and a larger number is a lower priority.

In order to allow more flexible interrupt masking, the ARMv7-M architecture provides a special register called `BASEPRI`, which masks exceptions or interrupts based on priority level. The K70 has sixteen programmable exception priority levels (4-bit width). When `BASEPRI` is set to 0, it is disabled. When it is set to a non-zero value, it blocks exceptions (including interrupts) that have the same or lower priority level, while still allowing exceptions with a higher priority level to be accepted by the processor. By default, `BASEPRI` is 0, which means the masking (disabling of exceptions / interrupts) is not active.

In many cases, rather than simply disabling all interrupts to carry out a certain time-sensitive task, you only want to disable interrupts with priority lower than a certain level. In this case, you write the required masking priority level to the `BASEPRI` register.

When you enable an interrupt source in your application, you get to decide on its priority level (0-15). Some of the exceptions (reset, NMI and HardFault) have fixed priority levels. Their priority levels are represented with negative numbers to indicate that they are of higher priority than other exceptions.

5.7 The Nested Vectored Interrupt Controller (NVIC)

All Cortex®-M processors provide a Nested Vectored Interrupt Controller (NVIC) for interrupt handling. The NVIC receives interrupt and exception requests from various sources:

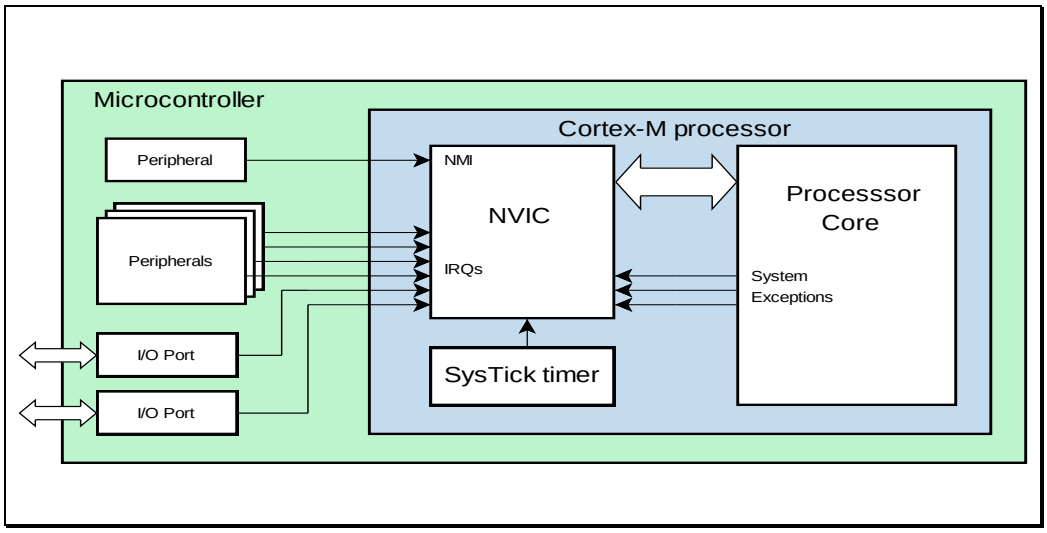


Figure 5.5 – The Nested Vectored Interrupt Controller

The K70 supports up to 240 interrupt requests (IRQs), a Non-Maskable Interrupt (NMI), a System Tick (SysTick) timer interrupt, and a number of system exceptions. Most of the IRQs are generated by peripherals such as timers, I/O ports, and communication interfaces (e.g. UART). The NMI is usually generated from peripherals like a watchdog timer or Brown-Out Detector (BOD). The rest of the exceptions are from the processor core. Interrupts can also be generated using software.

There are various status attributes applicable to each interrupt:

- disabled (default) or enabled
- pending (a request is waiting to be served) or not pending
- active (being served) or inactive

To support this, the NVIC contains programmable registers for interrupt enable control, pending status, and read-only active status bits.

5.7.1 Pending Status

The pending status of the interrupts are stored in programmable registers in the NVIC. When an interrupt input of the NVIC is asserted, it causes the pending status of the interrupt to be asserted. The pending status remains high even if the interrupt request is de-asserted.

The pending status means it is put into a state of waiting for the processor to serve the interrupt. In some cases, the processor serves the request as soon as an interrupt becomes pending. However, if the processor is already serving another interrupt of higher or equal priority, or if the interrupt is masked by one of the interrupt masking registers (e.g. `PRIMASK`), the pended request will remain until the other interrupt service routine is finished, or when the interrupt masking is cleared.

When the processor starts to process an interrupt request, the pending status of the interrupt is cleared automatically.

The pending status of interrupts are stored in interrupt pending status registers, which are accessible from software code. Therefore, you can clear the pending status of an interrupt or set it manually. If an interrupt arrives when the processor is serving another higher-priority interrupt and the pending status is cleared before the processor starts responding to the pending request, the request is cancelled and will not be served.

The pending status of an interrupt can be set even when the interrupt is disabled. In this case, when the interrupt is enabled later, it can be triggered and get served. In some cases this might not be desirable, so in this case you will have to clear the pending status manually before enabling the interrupt in the NVIC.

5.7.2 NVIC Registers for Interrupt Control

There are a number of registers in the NVIC for interrupt control (exception type 16 up to 255). By default, after a system reset, all interrupts:

- are disabled (enable bit = 0)
- have priority level of 0 (highest programmable level)
- have their pending status cleared

Interrupt Enable Registers

The Interrupt Enable register is programmed through two addresses. To set the enable bit, you need to write to the NVIC's Set Enable Register, `NVICSERx`; to clear the enable bit, you need to write to the NVIC's Clear Enable Register `NVICCERx`. In this way, enabling or disabling an interrupt will not affect other interrupt enable states. The `NVICSERx` / `NVICCERx` registers are 32-bits wide; each bit represents one interrupt input. As there are more than 32 external interrupts in the Cortex®-M4 processor, there is more than one `NVICSERx` and `NVICCERx` register.

Interrupt Pending Registers

The interrupt-pending status can be accessed through the Interrupt Set Pending (`NVICISPx`) and Interrupt Clear Pending (`NVICICPx`) registers. Similarly to the enable registers, there is more than one pending ISP and ICP register.

The values of the pending status registers can be changed by software, so you can cancel a current pended exception through the `NVICICPx` register, or generate software interrupts through the `NVICISPx` register.

Interrupt Priority-Level Registers

Each interrupt has an associated interrupt priority-level register (IPR). The 16 priority levels in the K70 are represented in the upper 4 bits of an IPR. The priority-level registers are generally accessed as 32-bit words, which means each `NVICIPRx` register holds 4 IRQ priorities.

See Section 3.2.2.3.1 of the K70 Sub-Family Reference Manual for an example on how to access the correct NVIC registers for a particular interrupt source.

5.16

EXAMPLE 5.1 Real-Time Interrupt using the Low Power Timer

Suppose we wish to make a simple application using the low-power timer interrupt to generate pulses on Port A, bits 0 and 1 (the pulses on these output pins could be used to keep track of the elapsed time by an external counter, or for viewing interrupt processing time on a DSO, for example).

The code below shows a simple scheme that shows the duration of the ISR and the timing operation of the main loop.

Code to generate and respond to real-time interrupts

```
void LPTMR_Init(void)
{
  // Enable clock gate to LPTMR module
  SIM_SCGC5 |= SIM_SCGC5_LPTIMER_MASK;

  // Disable the LPTMR while we set up
  LPTMR0_CSR &= ~LPTMR_CSR_TEN_MASK;
  // Enable LPTMR interrupts
  LPTMR0_CSR |= LPTMR_CSR_TIE_MASK;
  // Reset the LPTMR free running counter whenever
  //  the 'counter' equals 'compare'
  LPTMR0_CSR &= ~LPTMR_CSR_TFC_MASK;
  // Set the LPTMR as a timer rather than a counter
  LPTMR0_CSR &= ~LPTMR_CSR_TMS_MASK;

  // Bypass the prescaler
  LPTMR0_PSR |= LPTMR_PSR_PBYP_MASK;
  // Select the clock source
  LPTMR0_PSR |= LPTMR_PSR_PCS(1);

  // Set compare value – 1000 ticks of the 1 kHz clock = 1s
  LPTMR0_CMR = LPTMR_CMR_COMPARE(1000);

  // Initialize NVIC
  // see p. 91 of K70P256M150SF3RM.pdf
  // Vector 0x65=101, IRQ=85
  // NVIC non-IPR=2 IPR=21
  // Clear any pending interrupts on LPTMR
  NVICICPR2 = (1 << 21);
  // Enable interrupts from LPTMR module
  NVICISER2 = (1 << 21);

  //Turn on LPTMR and start counting
  LPTMR0_CSR |= LPTMR_CSR_TEN_MASK;
```

```
  // Clear bits 0 and 1 of Port A
  // Assumes Port A already set up for output
  GPIOA_PCOR = 0x00000003;
  // Interrupt counter
  Count = 0;
  // Foreground is ready
  Ack = 1;
}

void __attribute__ ((interrupt)) LPTMR_ISR(void)
{
  // Acknowledge interrupt, clear interrupt flag
  LPTMR0_CSR |= LPTMR_CSR_TCF_MASK;
  // Set bit 0
  GPIOA_PSOR = 0x00000001;
  // Software handshake - means LPTMR interrupt happened
  if (Ack == 1)
    Ack = 0;
  // Number of interrupts
  Count++;
  // Clear bit 0
  GPIOA_PCOR = 0x00000001;
}

void main(void)
{
  // Globally disable interrupts while we set up
  __DI();
  PortA_Init();
  LPTMR_Init();
  // Globally enable interrupts
  __EI();
  for (;;)
  {
    if (Ack == 0)
    {
      Ack = 1;
      // Toggle bit 1
      GPIOA_PTOR = 0x00000002;
    }
  }
```

In the vector table in `vectors.c`, we put:

```
  ...
  (tIsrFunc)&Cpu_Interrupt,          /* 0x63  TSI0        */
  (tIsrFunc)&Cpu_Interrupt,          /* 0x64  MCG         */
  (tIsrFunc)&LPTMR_ISR,              /* 0x65  LPTimer     */
  (tIsrFunc)&Cpu_Interrupt,          /* 0x66  Reserved102 */
  (tIsrFunc)&Cpu_Interrupt,          /* 0x67  PORTA       */
  ...
```

5.18

The figure below gives a flow chart of what is happening.

Flowchart for simple real-time interrupt program

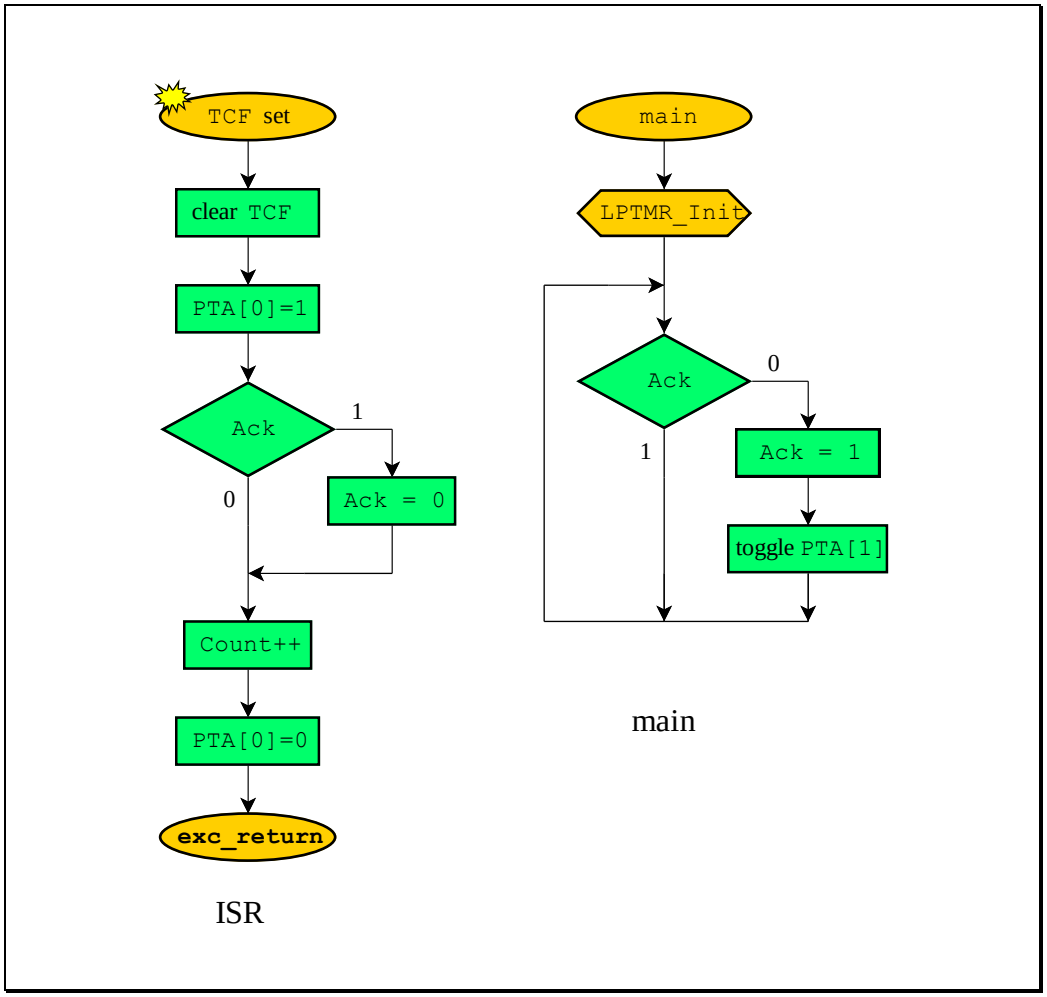
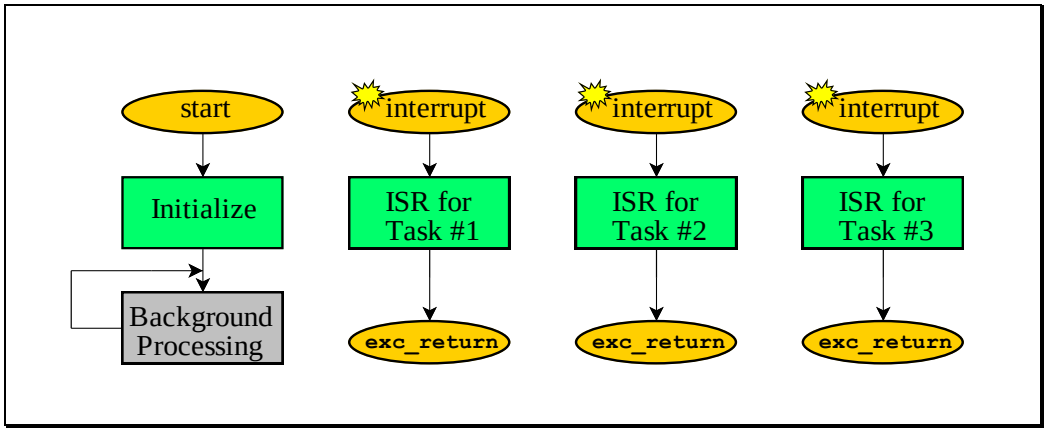


Figure 5.6 – Flowchart for Real-Time Interrupt Program

5.8 Foreground / Background Threads

In many systems where response time is critical, it is common to organize the program as a foreground / background system, as shown below.



Organization of a foreground / background system

Figure 5.7

Most of the actual work is performed in the "foreground", implemented as one or more interrupt service routines with each ISR processing a particular hardware event. This allows the system to respond to external events with a predictable amount of latency. To the extent that the external events are independent, there may be little or no communication between the various ISRs.

The main program performs the necessary initialization and then enters the "background" portion of the program, which is often nothing more than a simple loop that processes non-critical tasks and waits for interrupts to occur. Examples of background processing include: processing data from an input device, creating data for an output device, making calculations based on analog-to-digital conversion results, determining the next digital-to-analog output, and updating a display seen by human eyes.

5.9 Serial Communication Interface using Interrupts

Consider the common case of an application that uses the UART. The UART hardware receives characters at an asynchronous rate. In order to avoid loss of data in periods of high activity, the characters need to be stored in a FIFO buffer. The background task (main program) can process the characters at a rate which is independent of the rate at which the characters arrive. It must process the characters at an *average* rate which is faster than the *average* rate at which they can arrive, otherwise the FIFO buffer will become full and data will be lost. In other words, the buffer allows the input data to arrive in bursts, and the main program can access them when it is ready.

The following figure shows the situation for character reception.

An interrupt-driven input routine

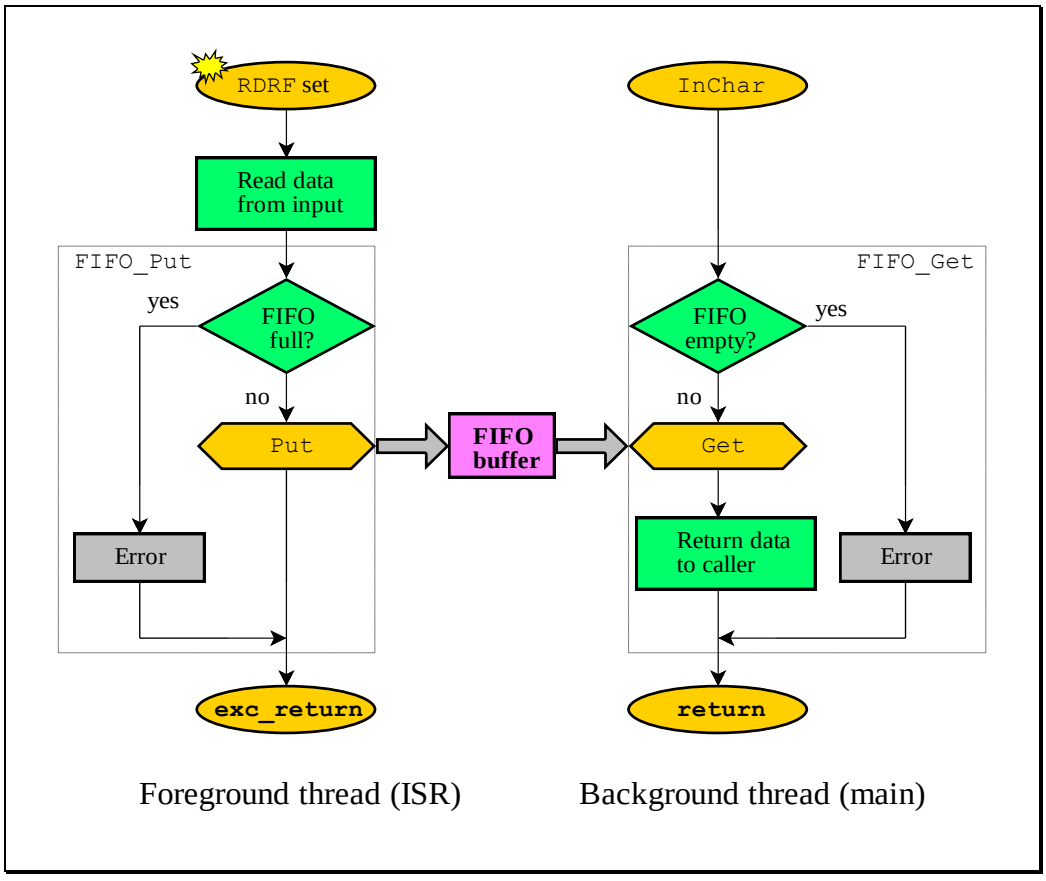


Figure 5.8

The structure for interrupt-driven character transmission is similar, except for one minor detail which must be resolved. Output device interrupt requests come in two varieties – those that request an interrupt on the *transition* to the ready state, and those that request an interrupt when they are *in* the ready state.

5.9.1 Output Device Interrupt Request on Transition to Ready

In this case, an output device requests an interrupt when it *finishes* processing the current output to indicate that it is now ready for the next output. In other words, the output ISR is invoked only when the output device *transitions* from a "busy" condition to a "ready" condition. In the context of serial port transmission, this creates two problems:

- When the background thread puts the first byte into the FIFO buffer, the output device is idle and already in the "ready" state, so no interrupt request from the output device is about to occur. The output ISR will not be invoked and the data will not be removed from the buffer.

- If somehow started, the "interrupt – **`FIFO_Get`** – output" cycle will repeat as long as there is data in the buffer. However, if the output device ever becomes ready when the buffer is empty, no subsequent interrupt will occur to remove the next byte placed in the buffer.

In these situations, the hardware normally provides a mechanism to determine whether or not the output device is busy processing data, such as a flag in a device status register. In these cases, the main work of the ISR should be placed in a separate function (e.g. **`SendData`**) that actually outputs the data.

Output devices need to be kick started

The background thread checks the output busy flag every time it writes data into the buffer. If the device is busy, then a device ready interrupt is expected and nothing needs to be done; otherwise, the background thread *arms* the output and calls **`SendData`** to "kick start" the output process.

The **`SendData`** routine is responsible for retrieving the data from the buffer and outputting it. If there is no more data in the buffer, then it must *disarm* the output to prevent further interrupts.

5.22

The flowchart given below illustrates the process.

Kick starting an interrupt-driven output routine for a device that requests interrupts on transitioning from busy to ready

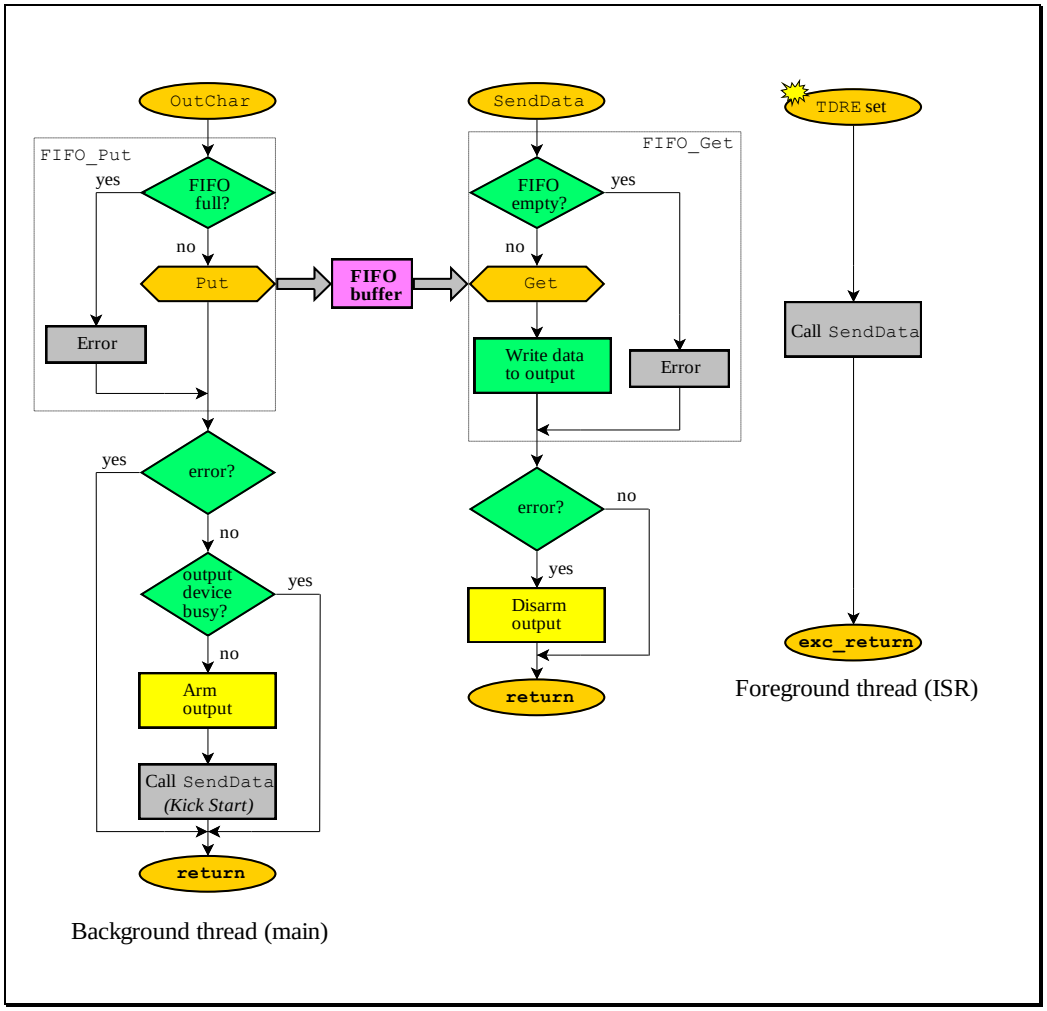


Figure 5.9

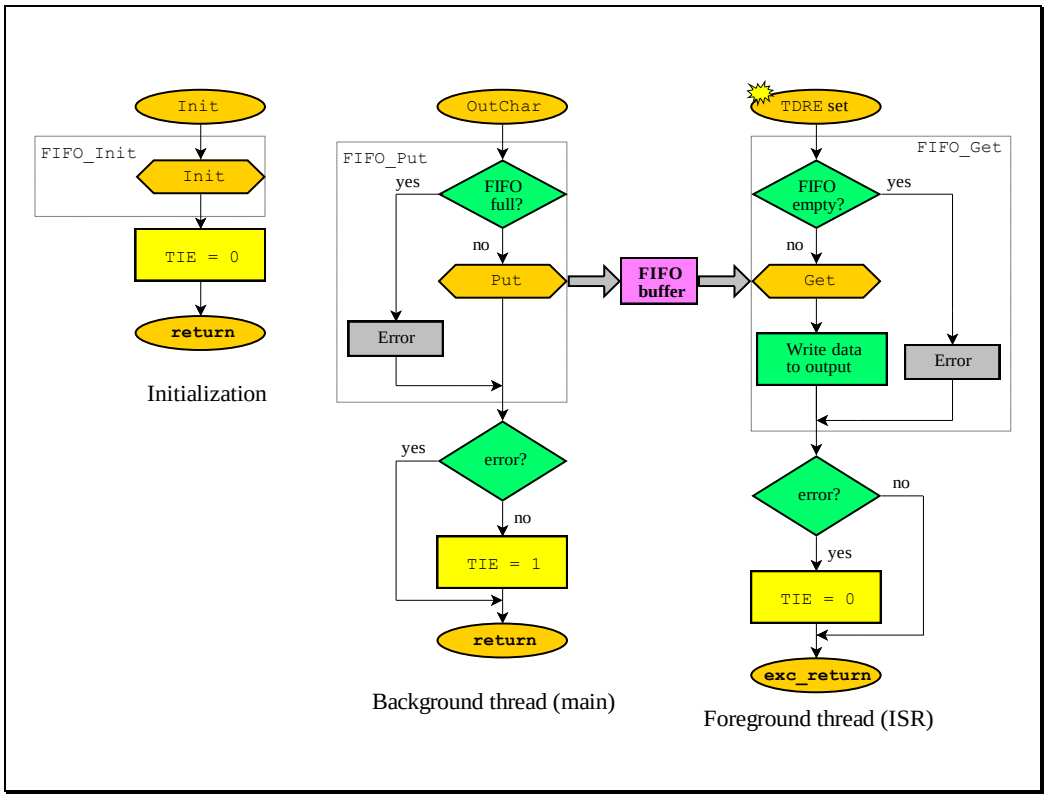
5.9.2 Output Device Interrupt Request on Ready

In this case, an output device sets its interrupt request flag when it is idle and ready for output (this will be the case after a reset condition, too). This means that upon initially arming the interrupt for such a device, an ISR will be invoked immediately. In the context of serial port transmission this creates two problems:

- How and when do we generate the first interrupt?
- What do we do if the device is ready but there is no data to output?

The technique to handle this type of interrupt is to modify both the **`OutChar`** routine and the ISR. The UART transmit interrupt is armed after every **`FIFO_Put`** (if the UART transmit interrupt were already armed, then rearming would have no effect). If the transmit FIFO is empty, then the ISR should disarm the transmit interrupt.

The flowchart given below illustrates the process.



An interrupt-driven output routine for devices that request an interrupt when they are ready

Figure 5.10

5.10 Communicating Between Threads

Communication between threads is accomplished using global variables

Communication between threads, without the support of an operating system, is accomplished with global variables. This leads to a problem – two (or more) threads may be trying to access and operate on the same variable at the same time. For example, in a FIFO buffer implementation for sending a byte out the serial port, the background thread (main) calls **`TxFIFO_Put`** to place a character into the buffer. This is a safe operation, because the byte is added to the end of the buffer. However, if the **`TxFIFO`** is keeping track of the number of bytes in the buffer with a global variable called `NbBytes`, then it must read, increment and write to this variable. A problem arises if a foreground thread (ISR) interrupts the background thread in the middle of the read-modify-write access to the global variable – erroneous values of **`NbBytes`** can result.

A *critical section* of code is a sequence of program instructions that *must not* be interrupted if erroneous operation is to be avoided. A critical section must prevent access to a global variable by more than one thread.

The solution to this simple problem is to implement access to `NbBytes` as an atomic operation. An *atomic operation* is one that is guaranteed to finish once it is started. In the K70, most *assembly language* instructions are atomic (apart from some special ones that are designed to be abandoned and restarted if an interrupt occurs). However, single lines of C code **cannot** be assumed to be atomic.

Cortex®-M4 processors have 16 registers inside the processor core to perform data processing and control. These registers are grouped in a unit called the register bank. Each data processing instruction specifies the operation required, the source register(s), and the destination register(s) if applicable. In the ARM® architecture, if data in memory is to be processed, it has to be loaded from the memory to registers in the register bank, processed inside the processor, and then written back to the memory, if needed. This is commonly called a “load-store architecture”.

Therefore, if the C code reads:

```
NbBytes++;
```

and `NbBytes` is byte-sized, then the compiler generates the following code:

A nonatomic sequence

```
ldr     r3, [pc, #72]   ; load address of NbBytes into R3
ldrb    r3, [r3, #0]    ; load value of NbBytes into R3
uxtb    r3, r3          ; zero extend upper bytes to 32 bits
adds    r3, #1          ; increment by one
uxtb    r3, r3          ; zero extend again
ldr     r2, [pc, #64]   ; load address of NbBytes into R2
adds    r1, r3, #0      ; copy new NbBytes value to r1
strb    r1, [r2, #0]    ; store new NbBytes in memory
```

As you can see, the C code `NbBytes++;` is nonatomic at an assembly language level because it consists of many instructions which can be interrupted. Thus, a single line of C code can be a "critical section".

We must know the architecture of the processor and have a rudimentary knowledge of the assembly language output produced by the compiler to determine whether a section of code is critical or not.

Since we are not using a real-time operating system (which would inherently support a multithreaded program by providing interthread communication mechanisms), one way of protecting the integrity of shared global variables is to disable interrupts during the critical section. This is a simple and acceptable method of protecting a critical section for a small embedded system.

It is important not to disable interrupts too long so as not to affect the dynamic performance of other threads. There is a problem however – consider what would happen if you simply add an "interrupt disable" at the beginning and an "interrupt enable" at the end of a critical section:

A problem with disabling and enabling interrupts to make a critical section

```
__DI();             // disable interrupts
NbBytes++;          // critical section
__EI();             // enable interrupts
```

What if interrupts were in a disabled state on entry into the critical section? Unfortunately, we have enabled them on exiting the critical section! What we need to do is save the state of the interrupts (enabled or disabled) before we enter the critical section, and restore that state on exiting.

5.10.1 Critical Sections in C for the ARMv7-M

In C, a way of implementing critical sections that preserves the interrupt state and allows nesting of critical sections (e.g. through function calls) is by declaring the following macros (defined in `PE_Types.h`):

C macros for entering and exiting a critical section

```
/* Save status register and disable interrupts */
#define EnterCritical() \
 do {\
  uint8_t SR_reg_local;\
  /*lint -save  -e586 -e950 Disable MISRA rule (2.1,1.1) checking. */\
   __asm ( \
     "MRS R0, FAULTMASK\n\t" \
     "CPSID f\n\t"            \
     "STRB R0, %[output]"  \
     : [output] "=m" (SR_reg_local)\
     :: "r0");\
  /*lint -restore Enable MISRA rule (2.1,1.1) checking. */\
   if (++SR_lock == 1u) {\
     SR_reg = SR_reg_local;\
   }\
 } while(0)


/* Restore status register  */
#define ExitCritical() \
 do {\
   if (--SR_lock == 0u) { \
  /*lint -save  -e586 -e950 Disable MISRA rule (2.1,1.1) checking. */\
     __asm (                \
       "LDRB R0, %[input]\n\t"\
       "MSR FAULTMASK, R0;\n\t" \
       ::[input] "m" (SR_reg)  \
       : "r0");               \
  /*lint -restore Enable MISRA rule (2.1,1.1) checking. */\
   }\
 } while(0)
```

Some explanations are in order.

Firstly, the \ character that appears at the end of each line is C's way of extending a single expression across more than one line.

Secondly, the `do {...} while(0)` construct is the only construct in C that you can use to `#define` a multistatement operation, put a semicolon after, and still use within an `if` statement. It also lets you declare local variables inside the block created with the braces. The multiple statements that appear between the braces `{...}` are only executed once due to the `while(0)`.

Thirdly, there are two global variables used by the macros, which are defined in CPU.c:

```
volatile uint8_t SR_reg;  /* Current value of the FAULTMASK register */
volatile uint8_t SR_lock = 0x00U;      /* Lock */
```

The basic idea of the code is:

1. EnterCritical() copies the value of FAULTMASK into register R0. It then disables all interrupts, including the HardFault exception, even if they were already disabled. All instructions from now on are guaranteed to execute without interruption. Importantly, this means that only this thread will have access to the global variable SR_reg. The saved value of FAULTMASK, which is stored in register R0, is then stored in the local variable SR_reg_local. Then, SR_lock is incremented (it is initially 0), and if equal to 1, the local variable SR_reg_local is copied to the global variable SR_reg. So, on "first entry", the state of the FAULTMASK register is stored in the global variable SR_reg. In subsequent "calls" to EnterCritical() the global variable does not get updated. The SR_lock variable is seen to be a count of the "nesting" of EnterCritical() "calls", and only the first state of FAULTMASK is saved.

2. ExitCritical() decrements the global SR_lock variable – an operation which is safe to perform since interrupts are disabled. If SR_lock is zero after decrementing, then we are leaving the last Enter/Exit pair. In this case the assembly language instructions will load SR_reg, the saved state from the first EnterCritical() "call", and place it in FAULTMASK, thus restoring the interrupt state to its initial value. If the SR_lock variable after decrementing is not zero, then we are still "nested" and there is nothing to do.

You can use the macros, with nesting, as shown in the example below:

```
void function(void)
{
  EnterCritical();
  ...
    EnterCritical();
    ...
    ExitCritical();
  ...
  ExitCritical();
}
```

Listing 5.3 – Nesting Critical Sections

To reiterate – whenever two (or more) threads share a global variable, you must protect access to that variable by operating in a critical section. The macros are **not** robust and you must guarantee that `EnterCritical()` and `ExitCritical()` occur in nested pairs. Be careful in your code that you do not enter a critical section inside a function and then exit that function without a corresponding `ExitCritical()`. Such a situation may arise when there are multiple exit points from a function:

```
void function(void)
{
  EnterCritical();
 ...
  if (error)
    return;  /* Error! We have not “called” ExitCritical() */
  ...
  ExitCritical();
}
```

Listing 5.4 – Incorrect Coding for a Critical Section

5.11 References

Yiu, J.: *The Definitive Guide to ARM® Cortex®-M3 and ARM Cortex®-M4 Processors*, Newnes, 2014. ISBN-13: 978-0-12-408082-9