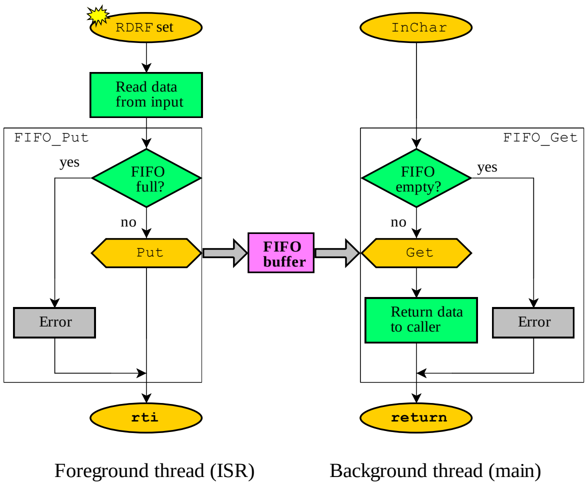
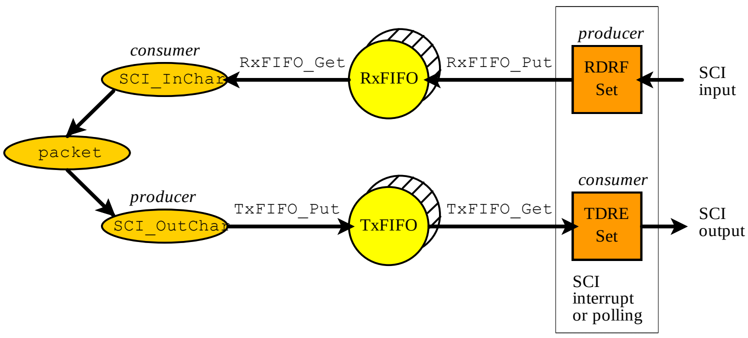
48434
Embedded Software

Topic Notes

2020





PMcL

Preface

These notes comprise part of the learning material for 48434 Embedded Software. They are not a complete set of notes. Extra material and examples may also be presented in the face-to-face activities.

Using the electronic version of these notes

These notes are hyperlinked. All green text is a link to somewhere else within this document. For example, the contents page links to the appropriate page in the text, and the page numbers in the header on each page link back to the contents page. There are also some internal linked words that take you to the relevant text.

Links to external web pages are red in colour. Provided your PDF reader (e.g. Adobe Acrobat Reader) is set up correctly these links should open the appropriate page in your web browser.

Contact

If you discover any errors or feel that some sections need clarifying, please do not hesitate in contacting me:

Peter McLean
School of Electrical, Mechanical and Mechatronic Systems
Faculty of Engineering and Information Technology
University of Technology Sydney

Office: CB11.11.405 - Building 11 (Broadway), Level 11, Room 11.405
Voice : +61-2-9514-2339
Email : peter.mclean@uts.edu.au
Web : http://www.uts.edu.au/staff/peter.mclean

Contents

5 Interrupts

6 Timing Generation and Measurements

7 Concurrent Software

8 Interfacing

9 Fixed-Point Processing

10 Real-Time Operating Systems

1 Embedded Systems

Contents

1.2

Introduction

Computing systems are everywhere. Billions of computing systems are built every year that are embedded within larger electronic devices, repeatedly carrying out a particular function, often going completely unrecognized by the device's user.

Examples of embedded systems

A quick look around our environment turns up embedded systems in a surprising number of places. The picture below shows just a few such systems in common environments.

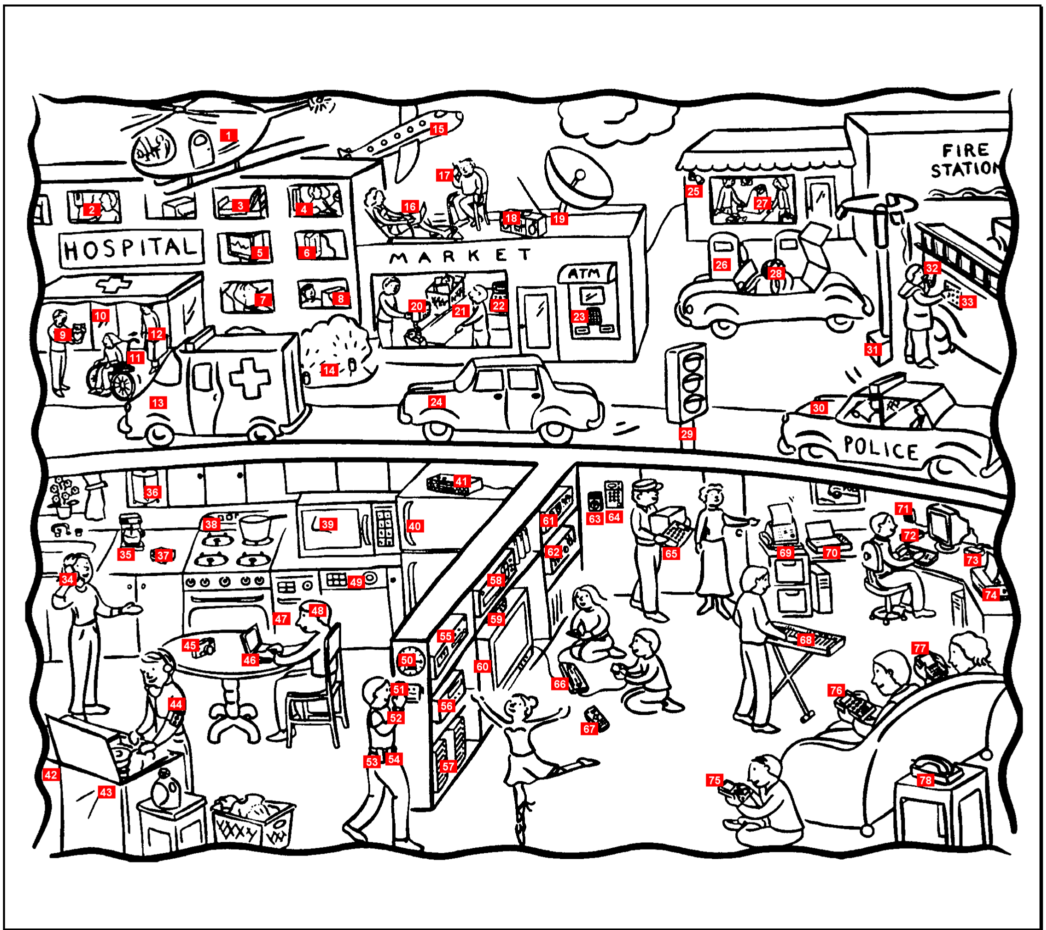


Figure 1.1 – Examples of Embedded Systems

A listing of these systems is given below:

Outdoors

1. Helicopter: control, navigation, communication , etc.
2. Medicine administering systems
3. Smart hospital bed with sensors and communication
4. Patient monitoring system
5. Surgical displays
6. Ventilator
7. Digital thermometer
8. Portable data entry systems
9. Pacemaker
10. Automatic door
11. Electric wheelchair
12. Smart briefcase with fingerprint enabled lock
13. Ambulance: medical and communication equipment
14. Automatic irrigation systems
15. Jet aircraft: control, navigation, communication, autopilot, collision-avoidance, in-flight entertainment, passenger telephones, etc.
16. Laptop computer (contains embedded systems)
17. Mobile telephone
18. Portable stereo
19. Satellite receiver system
20. Credit / debit card reader
21. Barcode scanner
22. Cash register
23. Automatic teller machine
24. Car (engine control, cruise control, temperature control, music system, anti-lock brakes, active suspension, navigation, toll transponder, etc.)
25. Automatic lighting
26. Pump monitoring system
27. Lottery ticket dispenser
28. Pager
29. Traffic light controller

Indoors

34. Cordless phone
35. Coffee maker
36. Rice cooker
37. Portable radio
38. Programmable oven
39. Microwave oven
40. Smart refrigerator
41. In-home computer network router
42. Clothes dryer
43. Clothes washing machine
44. Portable MP3 player
45. Digital camera
46. Electronic book
47. Garbage compactor
48. Hearing aid
49. Dishwasher
50. Electronic clock
51. Video camera
52. Electronic wristwatch
53. Pager
54. Mobile phone
55. CD player
56. DVD player
57. Smart speakers
58. Stereo receiver
59. TV set-top box
60. Television
61. PVR
62. TV-based Web access box
63. House temperature control
64. Home alarm system
65. Point-of-sale system
66. Video-game console
67. TV remote control
68. Electronic keyboard
69. Fax machine
70. Scanner

30. Police car (data lookup, communication, sirens, radar, etc.)
31. Mobile phone base station
32. Hand-held communicator (walkie-talkie)
33. Fire-control onboard computer

71. Wireless networking
72. Telephone modem
73. ADSL modem
74. Printer
75. Portable video game
76. Personal digital assistant
77. Portable digital picture viewer
78. Phone with answering machine

Nearly any device that runs on electricity either already has or soon will have a computing system embedded within it. In 2015, 1.43 billion smart phones, 212 million tablet PCs and 20 million eReaders were shipped.[1]

[1]http://www.idc.com/getdoc.jsp?containerId=prUS40664915
http://www.idc.com/getdoc.jsp?containerId=prUS25867215
(Accessed 2016-02-19)

1.1 Embedded Systems Characteristics

Embedded systems have several common characteristics that distinguish such systems from other computing systems:

1. *Single-functioned*: An embedded system usually executes a specific program or set or prgorams repeatedly.
2. *Tightly constrained*: Embedded systems often must cost just a few dollars, must be sized to fit into compact spaces, must perform fast enough to process data in real time, must consume minimum power, and must be designed rapidly to capture market windows.
3. *Reactive and real-time*: Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without delay.

1.2 Microcontrollers

A microcontroller is an integrated circuit that has a microprocessor connected up to various peripherals such as timers, serial ports, analog-to-digital converters, etc. You can think of a microcontroller as a "system-on-a-chip". An embedded system is usually made from a microcontroller and associated electronic circuitry that deals with interfacing the microcontroller to the "real world". The "art" of embedded systems programming is to write an application that utilises the hardware peripherals and interacts with the outside world in a manner which meets constraints 2 and 3 listed above. This requires a rudimentary understanding of the microcontroller architecture, the nature of the on-board peripherals, as well as understanding how the microcontroller interfaces with the real world. Invariably this means that an embedded software engineer must have a basic ability to interpret an electrical schematic diagram.

1.3 Development Boards

A microcontroller development board is a printed circuit board (PCB) containing a microcontroller and the minimal support logic needed for an engineer or hobbyist to become acquainted with the microprocessor on the board and to learn to program it. It also serves as a method to prototype applications before full product development.

Unlike a general-purpose system such as a personal computer, a development board usually contains little or no hardware dedicated to a user interface. In terms of the human machine interface (HMI), there may be only a few buttons and LEDs. They all have the provision to accept and run a user-supplied program, such as downloading a program through a serial port to Flash memory. Usually there is no enclosure or power supply.

Various microcontroller development boards

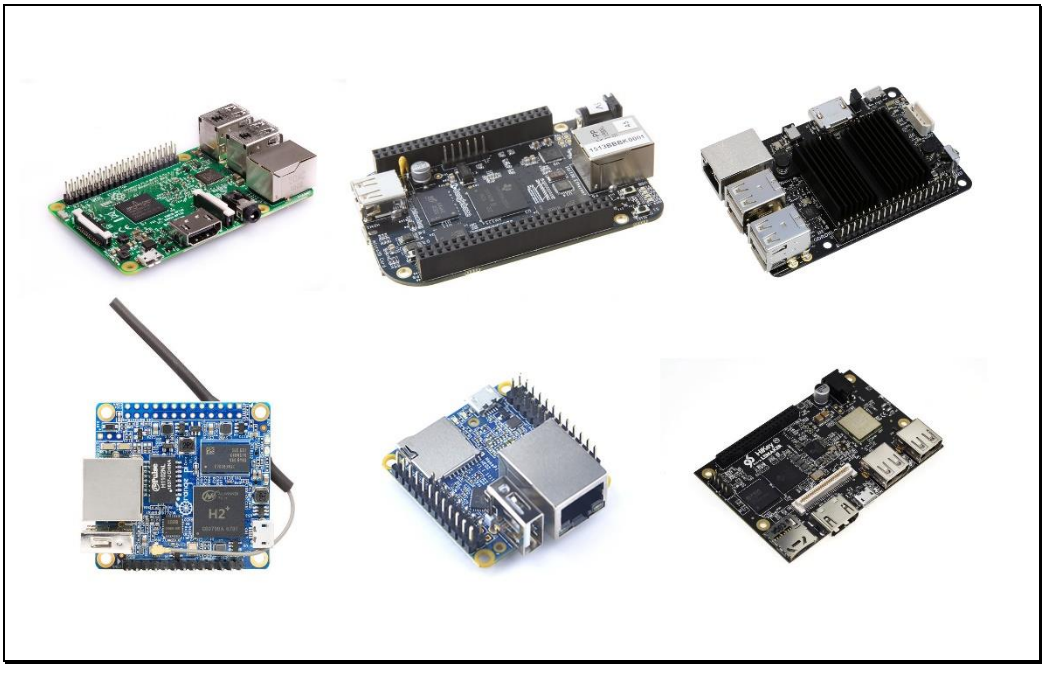


Figure 1.2 – Various Microcontroller Development Boards

Debugging (the process of running a program and validating its functionality) with a development board invariably involves connecting the development board to a personal computer. The personal computer runs an integrated development environment (IDE) which has a cross-compiler and linker and which can generate machine code for the embedded target chip. Development boards generally support advanced debugging capabilities of the target system.

1.3.1 The NXP FRDM-K64F

The NXP Freedom Boards are small, low-cost, Arduino® UNO Rev3 form-factor, development boards that support a variety of microcontrollers and peripheral boards that can be mixed-and-matched. The NXP FRDM-K64F development board (we will call it "the Freedom board") carries a Kinetis MK64FN1M0VLL12 microcontroller unit (MCU), an on-board JTAG debug circuit with a virtual serial port, 1Mib[2] of Flash memory, 256 kib of static RAM, a six-axis accelerometer and magnetometer, a tri-color LED, two pushbuttons, an Ethernet interface and a Micro-SD card slot:

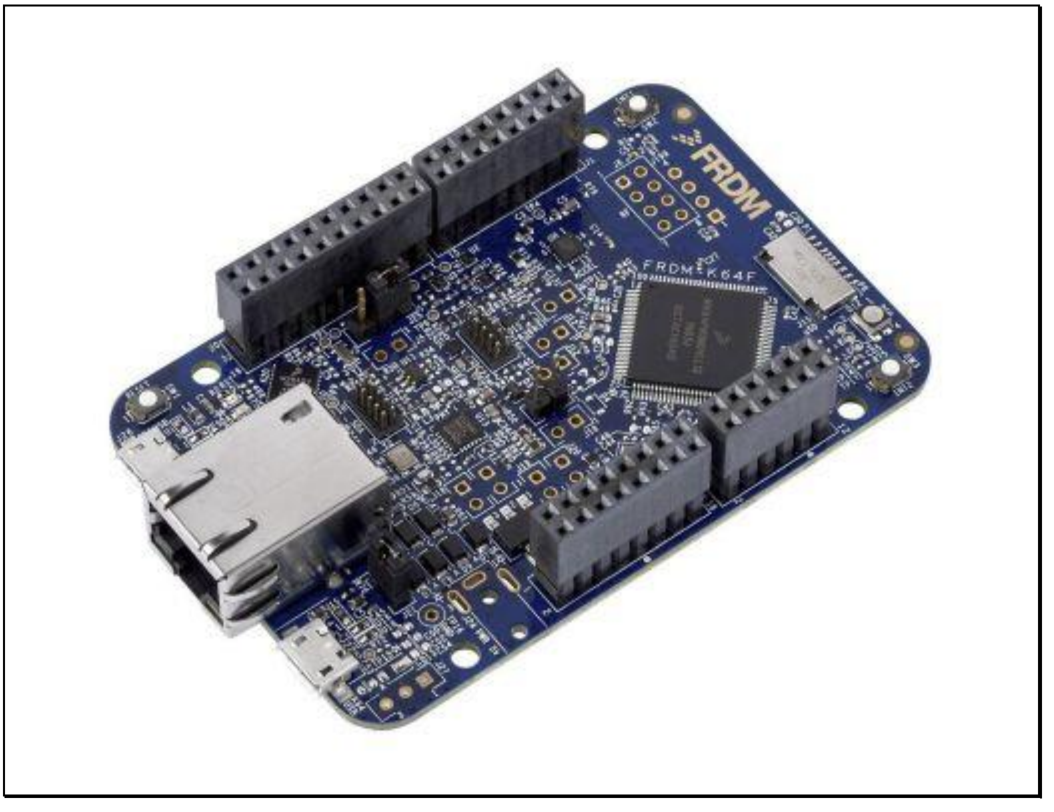


Figure 1.3 – The FRDM-K64F Development Board

Refer to NXP's FRDM-K64F Freedom Modules User's Guide for more information.

[2] The mebibit, abbreviated Mib, is a multiple of the unit "bit" used to quantify digital information. It is a member of the set of units with binary prefixes defined by the International Electrotechnical Commission (IEC). The prefix mebi (symbol Mi) represents 1024^2, or 1 048 576.

1.3.2 The Kinetis MK64FN1M0VLL12 Microcontroller Unit (MCU)

Kinetis is a family of NXP microcontroller units (MCUs) that are based on the extremely popular 32-bit ARM® Cortex®-M series of embedded processors.

The Kinetis MK64FN1M0VLL12 (hereafter abbreviated "K64") is based on the Cortex®-M4F, which implements the ARMv7E-M architecture. This is a "high-end" processor (for a microcontroller) that supports digital signal processing (DSP) instructions, single instruction multiple data (SIMD) instructions, and contains a hardware single precision floating-point unit (FPU). For more information on the ARM® Cortex®-M4, see:

http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php

The following figure shows the block diagram for the K64 family:

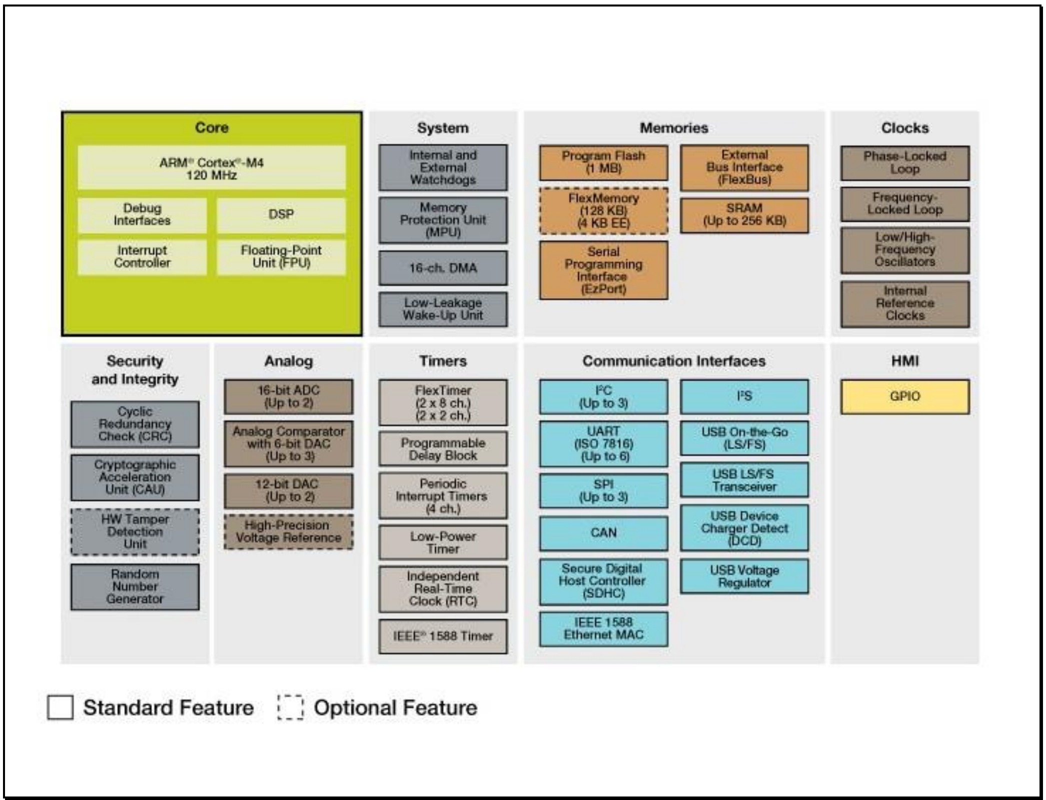


Figure 1.4 – Kinetis K64 Block Diagram

The Kinetis K64 MCU family offers low power and mixed-signal analog integration for applications such as industrial control panels, navigational displays, point-of-sale terminals, and medical monitoring equipment.

The K64 boasts an impressive array of system features and peripherals:

Module	Brief Description
Core	
ARM Cortex-M4	A high-end member of ARM's Cortex®-M series of processors.
Floating point unit (FPU)	A single-precision FPU that is compliant to the IEEE Standard for Floating-Point Arithmetic (IEEE 754).
Nested Vectored Interrupt Controller (NVIC)	The NVIC supports many interrupts, exceptions, and priority levels.
AWIC	The AWIC detects wake-up events.
Debug Interface	Four debug interfaces are supported.
System	
System integration module (SIM)	System integration logic and module settings.
System mode controller (SMC)	Provides control and protection on entry and exit to each power mode, and for resets.
Power management controller (PMC)	Provides the user with multiple power options.
Low-leakage wakeup unit (LLWU)	Allows the device to wake from low leakage power modes.
Miscellaneous control module (MCM)	Includes integration logic and embedded trace buffer details.
Crossbar switch (XBS)	The XBS connects bus masters and bus slaves.
Memory protection unit (MPU)	The MPU provides memory protection and task allocation.
Peripheral bridge	Allows the XBS to interface peripherals.
Direct memory access (DMA) controller	Provides for data movement without the CPU.
DMA multiplexer (DMAMUX)	Selects from many DMA requests down to a smaller number for the DMA controller.
External watchdog monitor (EWM)	Monitors both internal and external system operation for fail conditions.
Software watchdog (WDOG)	Monitors internal system operation and forces a reset in case of failure.

Table continues on the next page…

1.10

Memories and Memory Interfaces	
Flash memory	Program flash memory – non-volatile memory that can execute program code.
Flash memory controller	Manages the interface between the device and the on-chip flash memory.
Static Random Access Memory (SRAM)	Internal system RAM.
System register file	32-byte register file power by VDD.
VBAT register file	32-byte register file power by VBAT.
Serial programming interface (EzPort)	Provides the ability to read, erase, and program Flash memory and to reset and boot the system after flash programming.
FlexBus	External bus interface that support SRAM, PROM, EPROM, EEPROM, Flash and other peripherals.
Clocks	
Multi-clock generator (MCG)	Provides several clock sources for the MCU including: - a phase-locked loop (PLL) - a frequency-locked loop (FLL) - internal reference clocks
48 MHz Internal Reference Clock (IRC48M)	Provides an internally generated clock source which can be used as a reference of MCG or crystal-less FS USB implementation.
System oscillator	The system oscillator, in conjunction with an external crystal or resonator, generates a reference clock for the MCU.
Real-time clock (RTC) oscillator	The RTC oscillator has an independent power supply and supports a 32 kHz crystal oscillator.

Table continues on the next page…

Security and Integrity Modules	
Cryptographic acceleration unit (CAU)	Supports DES, 3DES, AES, MD5, SHA-1, and SHA-256 algorithms via simple C calls.
Random number generator (RNG)	Supports the key generation algorithm defined in the Digital Signature Standard.
Cyclic Redundancy Check (CRC)	The CRC generator can be used for error detection for all single, double, odd, and most multi-bit errors.
Analog Modules	
16-bit analog-to-digital converter (ADC)	16-bit successive-approximation ADC.
Analog comparator (CMP)	Compares two analog input voltages across the full range of the supply voltage.
6-bit digital-to-analog converter (DAC)	Provides a selectable voltage reference.
12-bit digital-to-analog converter (DAC)	Low-power general-purpose DAC, whose output can be placed on an external pin.
Voltage reference (VREF)	Supplies an accurate voltage output that is trimmable in 0.5 mV steps.
Timer Modules	
Programmable delay block (PDB)	16-bit counter that is initiated by a trigger event and provides for flexible delayed output signals.
Flexible timer module (FTM)	16-bit counter flexible counter with input capture, output compare and PWM.
Periodic interrupt timers (PIT)	32-bit general purpose interrupt timer.
Low-power timer (LPTimer)	16-bit time or pulse counter with compare.
Carrier modulator timer (CMT)	Provides the means to generate the protocol timing and carrier signals for a wide variety of encoding schemes.
Real-time clock (RTC)	32-bit and 64-bit real-time clock with alarm.
IEEE 1588 timers	The 10/100 Ethernet module contains timers to provide IEEE 1588 time stamping.

Table continues on the next page…

Communication Interfaces	
Ethernet MAC with IEEE 1588 capability (ENET)	10/100 Mbps Ethernet MAC with hardware support for IEEE 1588.
USB On-The-Go (OTG) (low-/full-speed)	USB 2.0 compliant module with support for host, device, and On-The-Go modes.
USB Device Charger Detect (USBDCD)	Detects a smart charger meeting the USB Battery Charging Specification Rev 1.2.
USB voltage regulator	Powers on-chip USB subsystem.
Controller Area Network (CAN)	Supports the full implementation of the CAN Specification Version 2.0, Part B.
Serial peripheral interface (SPI)	Synchronous serial bus for communication to an external device.
Inter-integrated circuit (I2C)	Allows communication between a number of devices.
Universal asynchronous receiver / transmitter (UART)	Asynchronous serial bus communication interface with support for CEA709.1-B (Lon works) and the ISO 7816 smart card interface.
Secure Digital host controller (SDHC)	Interface between the host system and the SD, SDIO, MMC, or CE-ATA cards.
Inter-IC Sound (I2S)	Provides a synchronous audio interface (SAI) that supports full duplex serial interfaces such as AC97 and codec / DSP interfaces.
Human-Machine Interface	
General purpose input/output (GPIO)	General purpose pins.

Table 1.1 – K64 Modules Grouped by Functional Categories

For more information, see the K64 Sub-Family Reference Manual.

1.4 ARM Microcontrollers

ARM is an acronym which stands for Advanced RISC Machine. RISC is another acronym which stands for Reduced Instruction Set Computing.

RISC is, in its broadest form, a design philosophy for processors. It stems from a belief that a processor with a relatively simple instruction set will be more efficient than one which is more complex. The term originally came into use back in the 1980s with a research project called Berkeley RISC that investigated the possibilities of this approach to design and then created processors based on it.

All ARM processors are considered RISC designs. Processors that have a RISC architecture typically require fewer transistors than those with a complex instruction set computing (CISC) architecture (such as the x86 processors found in most personal computers), which improves cost, power consumption, and heat dissipation. These characteristics are desirable for light, portable, battery-powered devices – including smartphones, laptops and tablet computers, and other embedded systems.

A company called ARM Holdings is responsible for ARM, and it is only a design company. They manage the instruction set and design new versions of the core architecture and then license it to other companies. Those companies can then improve it and pair it with whatever hardware seems appropriate.

ARM's core architecture is only a processor. It doesn't handle wireless connectivity. It doesn't handle graphics. It doesn't handle USB or other forms of wired connectivity. All the peripherals on a chip are the responsibility of silicon vendors. Companies like Apple, NXP, Texas Instruments, ST etc. employ in-house engineering teams to create ARM-based microcontrollers – they add the desired peripherals around the ARM processor and package it as a microcontroller or as a "system-on-a-chip".

1.4.1 The ARM® Cortex®-M Series of Processors

There are 3 families of ARM processor, each designed for a specific market:

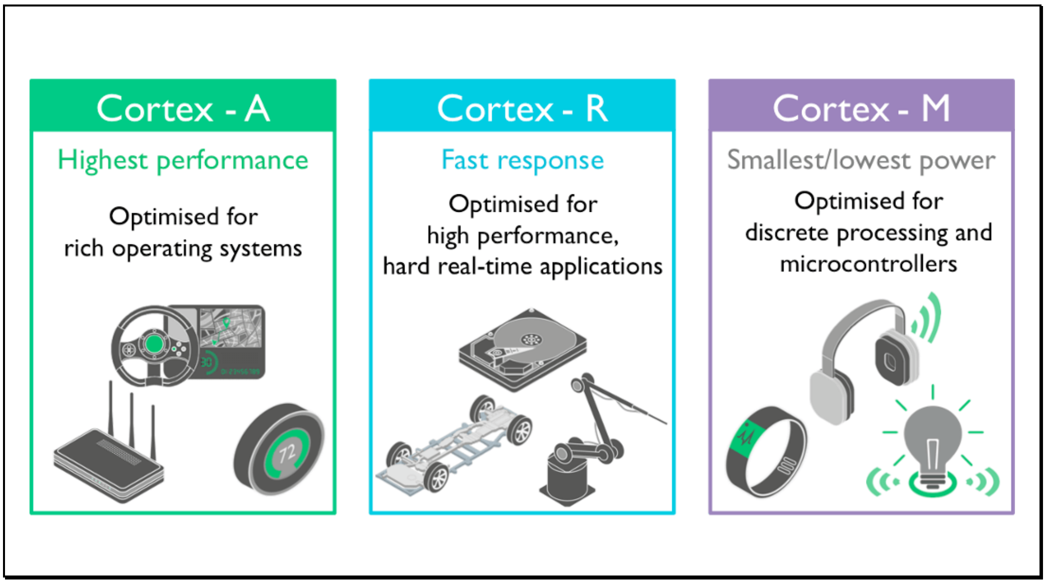


Figure 1.5 – The ARM® Cortex®-M series of Embedded Processors

ARM developed the popular 32-bit ARM® Cortex®-M series of embedded processors primarily for the microcontroller domain where the need for fast, highly deterministic interrupt management is coupled with extremely low gate count and lowest possible power consumption. For more information on the ARM® Cortex®-M Series of processors, see:

http://www.arm.com/products/processors/cortex-m/index.php

ARM states that:

> *The Cortex-M family is optimized for cost and power sensitive MCU and mixed-signal devices for applications such as Internet of Things, connectivity, motor control, smart metering, human interface devices, automotive and industrial control systems, domestic household appliances, consumer products and medical instrumentation.*[3]

[3] http://www.arm.com/products/processors/cortex-m/index.php (Accessed 2015-07-24)

There is a huge "ecosystem" surrounding ARM® processors because of their popularity – most mobile phones have an ARM® processor (based on the Cortex®-A Series), as does the Microsoft Surface, the Apple iPad, and the Raspberry Pi.

In summary, there are many variants of ARM on the market and they all perform differently. However, if you are familiar with the ARM core, you should be able to move easily from chip to chip and vendor to vendor – this is the advantage of learning about and using a chip from the ARM ecosystem.

Between 1991 and 2013, ARM shipped 50 billion units. By 2005, ARM was producing a billion units per year, and by 2009 the company was shipping a billion units every quarter. To date, ARM has 100 billion units in the market; by 2021, the company expects to have shipped 100 billion more.[4]

[4] https://www.tomshardware.com/news/arm-dynamiq-multicore-microachitecture,33947.html (Accessed 2018-04-26)

1.5 Embedded Systems Programming Languages

There are just a handful of computer programming languages suitable for embedded systems. The requirements for a language to be suitable for embedded systems are:

- Efficient compilation (small code size)
- Portability (between devices)
- Sufficient low-level operations (e.g. bit manipulation)
- High-level constructs (e.g. pointers, strong types)
- Availability of software libraries (e.g. math, signal processing)

In all these respects, and for historical reasons, the C language is dominant in embedded systems programming. An IEEE survey from 2019 shows the top 10 programming languages used in industry:

Language Ranking: Jobs

Rank	Language	Type	Score
1	Python		100.0
2	C		96.0
3	C++		85.7
4	C#		77.2
5	Assembly		67.4
6	Arduino		53.9
7	Rust		52.1
8	LabView		38.2
9	VHDL		34.4
10	Verilog		33.9

Figure 1.6 – The Top 10 Embedded Systems Programming Languages

In Figure 1.6, note that Python does not have the smartphone symbol which represents "embedded systems". You can see that the C family of languages (C, C++ and C#) are the most popular languages for embedded systems in industry, although Python variants, such as a MicroPython, are starting to become more popular.

In this subject, we will be using just plain old C, and specifically C11 which is a superseded international standard (ISO/IEC 9899:2011). C18 is now the current standard (ISO/IEC 9899:2018), but compiler support in the "embedded world" for this standard is minimal at this time.

If you are comfortable and proficient in the C language (i.e. you consider yourself an advanced C programmer), you are welcome to program in C++.

1.6 The Apollo Guidance Computer (AGC)

A generation ago, in the 1960’s, humans decided to go to the moon – this was dubbed the “Apollo program” by the United Stated of America’s National Aeronautics and Space Administration (NASA). This was an audacious goal, and to get there demanded huge technological advancements. One of the key technologies developed in that era was the Apollo Guidance Computer (AGC) – the very first silicon integrated circuit based computer. It was developed by a team at the Massachusetts Institute of Technology (MIT) Instrumentation Lab in the 1960’s. Without it, humans would not have made it to the moon. The design and programming of the AGC set the stage for the development of embedded systems and computers – just about every principle and architecture of the hardware and software has been adopted by subsequent technology. The influence of the AGC cannot be overstated!

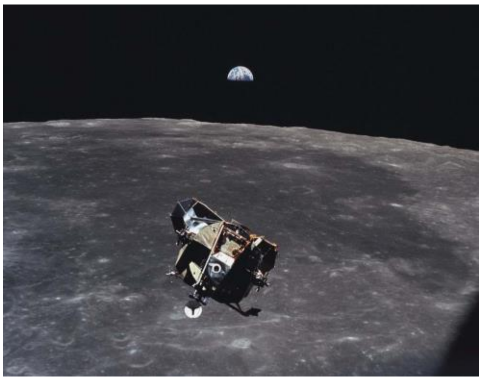
The AGC was the first ever digital computer based on silicon integrated circuits – and the first to run a real-time operating system

Wikimedia Commons

You have to remember that when the Apollo program began, computers were still gigantic machines that took up whole rooms. The task of the MIT team (which ultimately expanded to over 400 engineers and programmers) was to handle the complexity of space navigation with a digital computer that weighed only 32 kg, consumed only 55 W of power, had the computing power of a modern hand-held calculator and the dimensions of 61 x 32 x 17 cm.

Each Apollo mission to the Moon carried two AGCs, one in the Command and Service Module (CSM) and the other in the Lunar Module (LM). In their hardware the two machines were nearly identical; software tailored them to their distinctive functions.

For a taste of what the computers were asked to accomplish, consider the workload of the lunar module's AGC during a critical phase of the flight – the powered descent to the Moon's surface. The first task was navigation: measuring the craft's position, velocity, and orientation, then plotting a trajectory to the target landing site. Data came from the gyroscopes and accelerometers of an inertial guidance system, supplemented in the later stages of the descent by readings from a radar altimeter that bounced signals off the Moon's surface.



Wikimedia Commons

The lunar module returning from the moon to rendezvous with the Apollo 11 command module

After calculating the desired trajectory, the AGC had to swivel the nozzle of the rocket engine to keep the capsule on course. At the same time it had to adjust the magnitude of the thrust to maintain the proper descent velocity. These guidance and control tasks were particularly challenging because the module's mass and centre of gravity changed as fuel was consumed and because a spacecraft sitting atop a plume of rocket exhaust is fundamentally unstable – like a broomstick balanced upright on the palm of your hand.

Along with the primary tasks of navigation, guidance, and control, the AGC also had to update instrument displays in the cockpit, respond to commands from the astronauts, and manage data communications with ground stations. Such multitasking is routine in computer systems today. In the early 1960s, however, the tools and techniques for creating an interactive, "real-time" computing environment were in a primitive state.

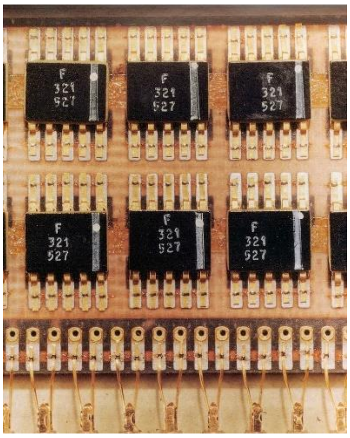
Hardware

The AGC's processor specifications don't seem impressive today:

- 34 instructions
- 2048 words of erasable memory
- 36864 words of read-only memory
- 16-bit word format (15 data + 1 parity)
- 1.024 MHz clock

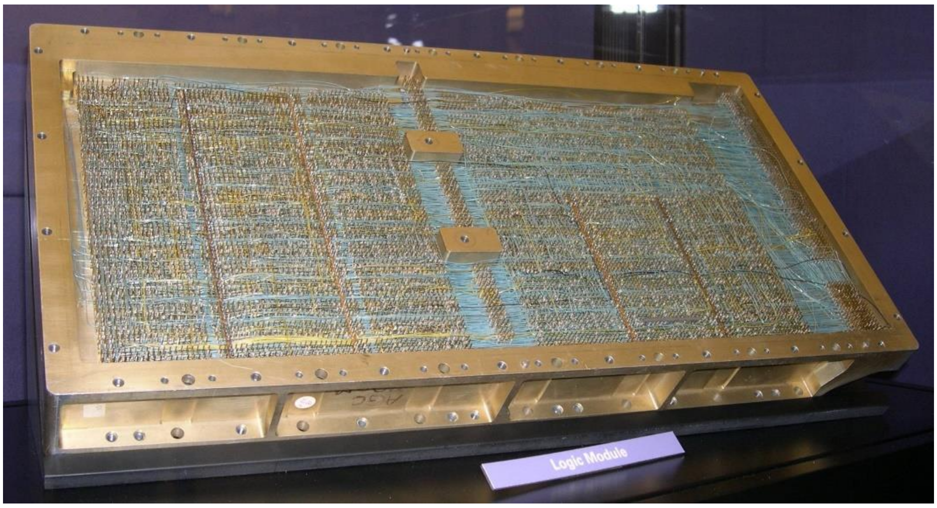
The hardware was built from just one type of logic gate – the 3-input NOR gate. There were only 2800 integrated circuits (ICs) used in the design – each IC was a dual 3-input NOR gate.

The AGC hardware logic was built entirely from 3-input NOR gates



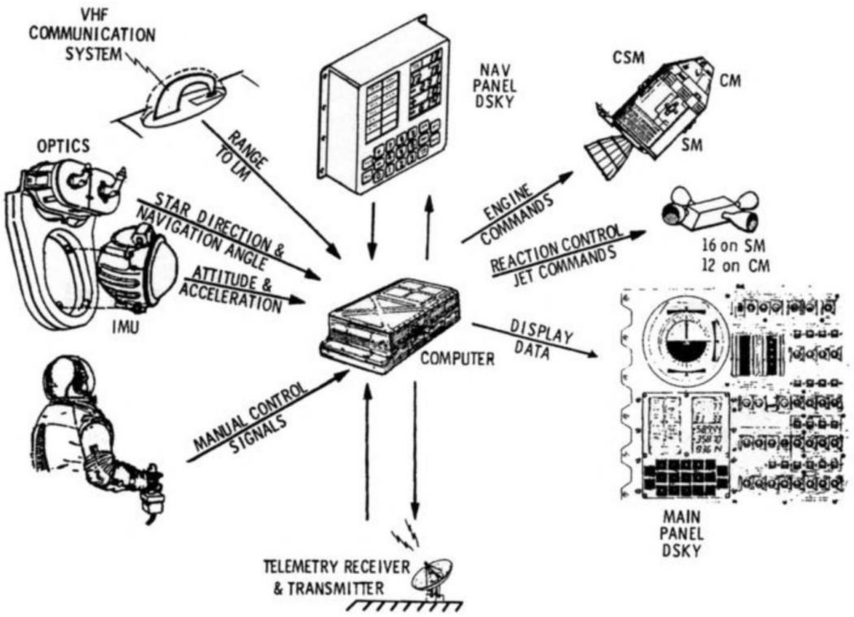
Charles Stark Draper Laboratory Archives, Photograph Number 40589-C

They were connected on the back of a flat panel via welds and wire wrap, and then cast in epoxy resin.



Wikimedia Commons

In the CSM the AGC was connected to an inertial measurement unit (IMU), a hand controller, a telemetry receiver and transmitter, an engine command unit, a reaction control system, a VHF communication system, an optics subsystem with a sextant and a telescope, various analog displays, and a display/keyboard unit.



The AGC interfaced to many types of peripherals in real-time. This picture shows the peripherals it connected to in the Command and Service Module.

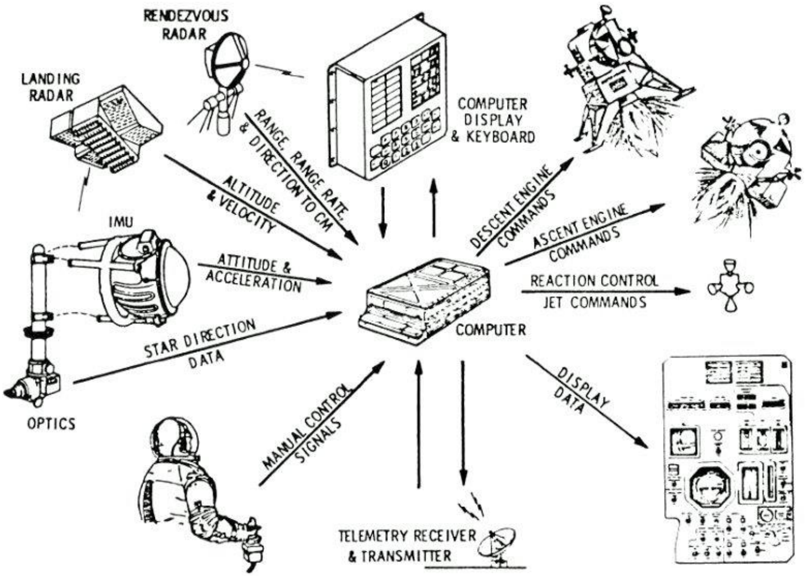
The peripherals were fed directly into memory-mapped I/O channels that were accessible only by special I/O instructions. There were 7 input channels and 14 output channels, all 16-bits wide.

There were 20 “involuntary counters” which were memory locations which functioned as up/down counters or shift registers. Hardware, such as the Inertial Measurement Unit, were fed directly into the counters which then represented accelerometer and gimbal changes. The counters were updated directly by hardware, so the software was *not* interrupted. Only when a counter overflowed would an interrupt be triggered.

The AGC also had a power-saving mode for use in midcourse flight, but this was never used (it was left on full power for all phases of the mission).

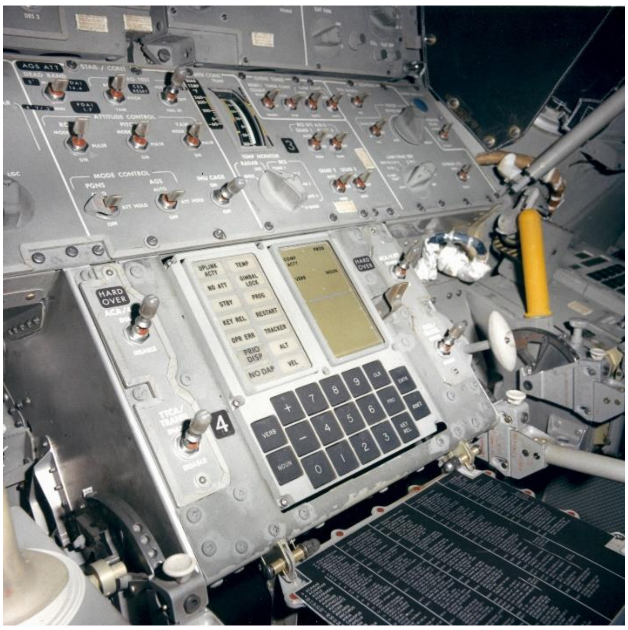
In the Lunar Module the AGC was connected to a similar set of peripherals, but this time also had a landing radar, a rendezvous radar, and 2 engine control units – one for descent and one for ascent.

The AGC and peripherals for the Lunar Module.



The AGC system had an interface for the astronaut to communicate directly with it. Engineer Ramon Alonso came up with a simple display and keyboard, named the "DSKY" (pronounced DIS-kee), where two-digit numbers represent programs, verbs and nouns. The astronaut punched data and commands into the system. When the computer requested the astronaut to take some action, the numbers would flash to attract attention.

The DSKY shown at the bottom of the control panel in the Lunar Module used in Apollo 16. Note the detailed information available on the fold-down pallet at the bottom of the image.

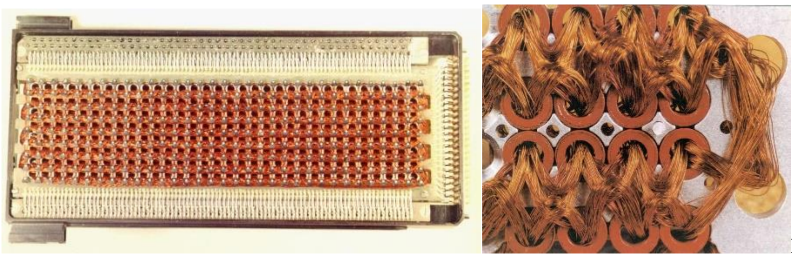


NASA

Software

The fact that the AGC had such limited computing power meant the software had to be very special. When the design requirements for the AGC were defined, software programming techniques did not exist so it had to be designed from scratch. A real-time operating system (RTOS) was designed by Hal Laning, with no prior examples to guide him. The AGC effectively prioritised certain “tasks”, and let the “unimportant” ones languish whilst carrying out the vital control and navigation tasks as the lunar module landed.

The bulk of the software was stored in read-only rope memory and thus couldn't be changed in operation, but some key parts of the software were stored in standard read-write magnetic-core memory and could be overwritten by the astronauts using the DSKY interface.



Raytheon

The AGC’s core rope memory – a technique of physically weaving software into high-density storage

During the Apollo 14 mission, a faulty abort switch would have caused an aborted landing attempt when the lunar descent was begun. The hardware error was detected shortly before descent. In less than 2 hours, the problem was diagnosed and a software patch was developed, tested, and relayed to the astronauts to key in by hand using the DSKY – resulting in a successful lunar landing!

The RTOS in the AGC became foundational to software engineering. It consisted of the *Exec*, a batch job-scheduler using cooperative multi-tasking, and an interrupt-driven pre-emptive scheduler called the *Waitlist* which could schedule multiple timer-driven 'tasks'. All work was assigned a priority. The *Exec* selected the job with the highest priority to run. Every 20 ms, the job queue was checked for the highest priority task, which was then executed. A “night watchman”, which executed every 1.2 s, verified that a job was not looping and new work was being scheduled.

If errors were encountered, a “Program Alarm” was issued, and depending on the severity of the error, could force a computer “restart” – in which all critical data was saved, jobs were terminated, hardware was reinitialized, and programs were re-entered at predefined restart points. The whole process of error recovery took only a few seconds. If you watch a video of the Apollo 11 moon landing, you can hear the call of “1201” and “1202” master alarms, indicating the AGC was overloaded. The design of the AGC software, functioning exactly as it was designed to do (by restarting), is the reason Neil Armstrong and Buzz Aldrin were able to land on the moon on 20 July 1969, despite experiencing the hardware fault that led to the master alarms.

An interrupt system was implemented in the AGC via five “vectored interrupts”. The AGC responded to each interrupt by temporarily suspending the current program, executing a short interrupt service routine, and then resuming the interrupted program.

The AGC also had a sophisticated software interpreter that implemented a virtual machine with more complex and capable pseudo-instructions than the native AGC. These instructions simplified the navigational programs. Interpreted code, which featured double precision trigonometric, scalar and vector arithmetic (16 and 24-bit), and even a (matrix × vector) instruction, could be mixed with native AGC code. While the execution time of the pseudo-instructions was increased (due to the need to interpret these instructions at runtime) the interpreter provided many more instructions (more than 100) than the AGC natively supported (34). The use of pseudo-instructions lowered the memory (at that time memory was very expensive) and eased the burden of programming complex mathematical and logical operations.

Margaret Hamilton

Margaret Hamilton, who ended up leading the on-board flight software team for both the CSM and LM, said this of the software:

> The biggest challenge: The astronauts' lives depended on our systems and software being man-rated. Not only did it have to work, it had to work the first time. Because of the never-ending focus on making everything as perfect as possible, anything to do with the prevention of errors was not only not off the table, but it was top priority both during development and in real-time. Not only did the software itself have to be ultra-reliable, but the software would need to have the flexibility to detect anything unexpected and recover from it in real-time; that is, at any time during the entirety of a real mission. To meet the challenge, the software was developed with an ongoing, overarching focus on finding ways to capitalize on the asynchronous and distributed functionality of the system at large, in order to perfect the more systems-oriented aspects of the flight software.
>
> Our software was designed to be asynchronous to have the flexibility to handle the unpredictable, and so that higher-priority jobs would have the capability to interrupt lower-priority jobs, based on events as they happened, especially in the case of an emergency. This goal was accomplished by our correctly (and wisely) assigning a unique priority to every process in the flight software, to ensure that all the events in the flight software would take place in the correct order and at the right time relative to everything else that was going on. Steps taken earlier within the software to create solutions within an asynchronous software environment became a basis for solutions within a distributed systems-of-systems environment.
>
> The responsibilities of the people within the on-board flight software group (the software engineers) included: the development of software algorithms designed by various senior scientists for the Apollo Command Module and Lunar Lander; the overall design of the structure (the "glue") of the software as an integrated system-of-systems; ensuring that all the modules within the flight software – including all aspects of the modules such as those related to timing, data and priority – were completely integrated; the design and development of the "systems-software"; and the design of the "software engineering" techniques, which included rules, methods, tools, and processes for ensuring that the software being developed would result in an ultra-reliable system (i.e., making sure that the software would have no errors, both during development and in real-time). Because of these requirements, the team developed and evolved "software engineering" techniques for both the development of the software, the testing of the software (included 6 formal levels of testing) within a system-of-systems environment, and the management of evolving and daily releases that contained and documented everyone's most recent changes (and the reason for the changes) for each and every mission. Methods and tools evolved for these kinds of software-management techniques as well.

Margaret Hamilton started on the Apollo program in a position programming the AGC – by the end of the program she was the director of software engineering of all on-board software.

Wikimedia Commons

Legacy

Being new and untested technology, NASA was reluctant to let the AGC be the primary means of control and navigation of the Command and Service Module (CSM) spacecraft. Ultimately, primary navigation of the CSM was performed using Earth-based radar systems during the majority of the moon mission (using Doppler shift radar for position and velocity). However, data generated by the AGC would provide critical navigational data to the crew about spacecraft position, direction, velocity and acceleration when they were completely cut off from ground radar and communications – while orbiting the far side of the moon.

The AGC was, however, absolutely essential for the Lunar Module (LM). It carried programs for the three phases of landing – braking, approach, and final descent. In the final descent (started between 300 to 150 m altitude), the LM would be manually flown by the astronauts for about a minute. There was only one attempt at landing – if anything went wrong, astronauts would have to hit an abort button which would fire the ascent engines and return the LM to the orbiting CSM.

At the time, the AGC was the latest and most advanced fly-by-wire and inertial guidance system, the first digital flight computer, the most advanced miniature computer, the first computer to use silicon chips, and the first on-board computer where the lives of crew depended on it functioning as advertised.

By the end of the last Apollo mission in 1972, the AGC was hopelessly outdated. But it had flown on 15 manned missions, including nine moon flights, six lunar landings, and three Skylab missions. It was also used in experimental fly-by-wire aircraft. It *never* failed.

The design of the AGC has a powerful human resonance, and the history of its development offers a glimpse of the cultural milieu of a high-profile, high-risk, high-stress engineering project. I encourage you to delve deeper into its history.

The AGC’s legacy lives on in every real-time embedded system.

References

Hall, Eldon C: *Journey to the Moon: The History of the Apollo Guidance Computer*, American Institute of Aeronautics and Astronautics, Inc., 1996.

https://en.wikipedia.org/wiki/Apollo_Guidance_Computer
(Accessed 2020-02-12)

https://www.americanscientist.org/article/moonshot-computing
(Accessed 2020-02-12)

https://tcf.pages.tcnj.edu/files/2013/12/Apollo-Guidance-Computer-2009.pdf
(Accessed 2020-02-12)

http://www.righto.com/2019/07/software-woven-into-wire-core-rope-and.html
(Accessed 2020-02-12)

2 Embedded C

Contents

2.4

Introduction

This document gives a basic overview of programming in C for an embedded system – based on a specific 32-bit ARM® processor, NXP's "K64".

2.1 Program Structure

Some basic terms will be introduced so that you get a feel for the language. It is not important yet that you understand the example programs fully. The examples are included to illustrate particular features of the language.

2.1.1 Case Study 1: Microcomputer-Based Lock

To illustrate the software development process, we will implement a simple digital lock. The lock system has 7 toggle switches and a solenoid as shown in the following figure.

A digital lock using a microcontroller

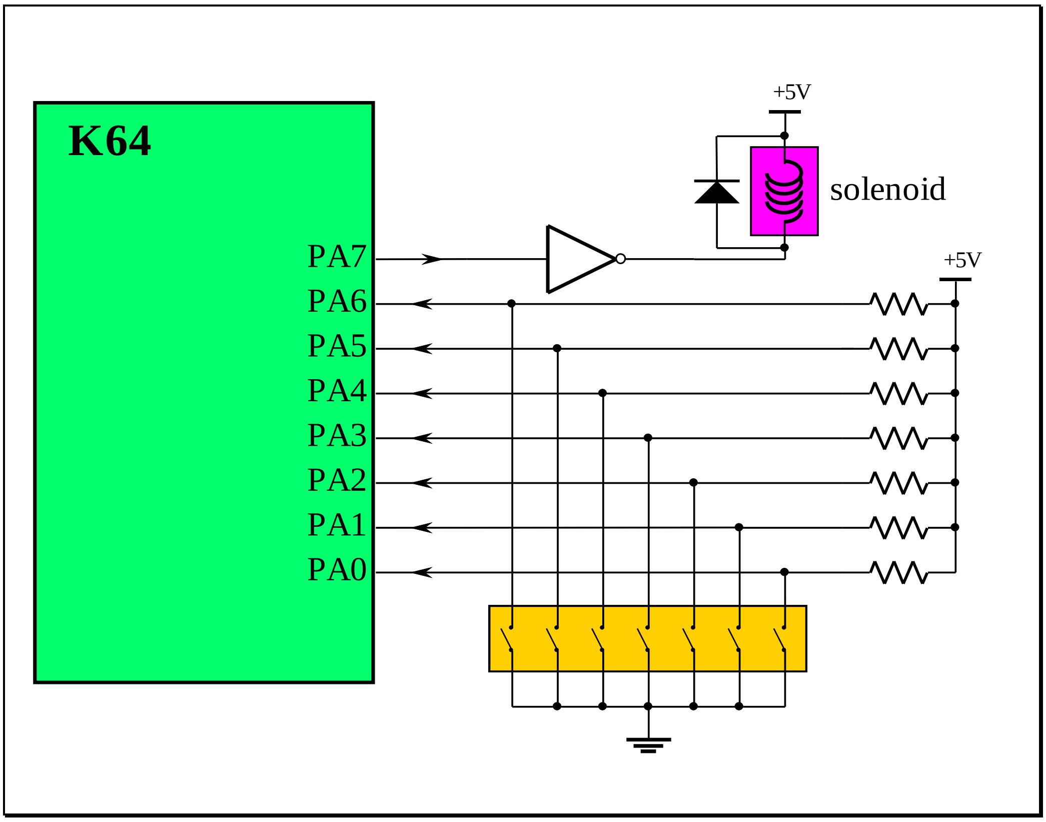


Figure 2.1 – Digital lock hardware

If the 7-bit binary pattern on Port A bits 6-0 becomes `0100011` for at least 10 ms, then the solenoid will activate. The 10 ms delay will compensate for the switch bounce. We see that Port A bits 6-0 are input signals to the computer and Port A bit 7 is an output signal.

Before we write C code, we need to develop a software plan. Software development is an iterative process. The steps below are listed in a 1, 2, 3, … order, whereas in reality we iterate these steps over and over.

Iterative steps used to write software

1. We begin with a list of the inputs and outputs. We specify the range of values and their significance. In this example we will use `GPIOA`. Bits 6-0 will be inputs. The 7 input signals represent an unsigned integer from 0 to 127. Port A bit 7 will be an output. If `GPIOA` bit 7 is `1` then the solenoid will activate and the door will be unlocked. In C we use `#define` macros to assign symbolic names, `GPIOA_PSOR`, `GPIOA_PCOR` and `GPIOA_PDDR`, to the corresponding addresses of these registers, `0x400FF004`, `0x400FF008` and `0x400FF014`.

Accessing microcontroller ports in C

```
#define GPIOA_PSOR *(uint32_t volatile *)(0x400FF004)
#define GPIOA_PCOR *(uint32_t volatile *)(0x400FF008)
#define GPIOA_PDDR *(uint32_t volatile *)(0x400FF014)
```

2. Next, we make a list of the required data structures. Data structures are used to save information. If the data needs to be permanent, then it is allocated in global space. If the software will change its value then it will be allocated in RAM. In this example we need a 16-bit unsigned counter.

```
uint16_t Count;
```

If a data structure can be defined at compile time and will remain fixed, then it can be allocated in Flash memory. In this example we will define an 8-bit fixed constant to hold the key code, which the operator needs to set to unlock the door. The compiler will place these lines with the program so that they will be defined in Flash memory.

```
const uint8_t KEY = 0x23; /* key code */
```

It is not clear at this point exactly where in Flash this constant will be, but luckily for us, the compiler will calculate the exact address automatically. After the program is compiled, we can look in the listing file or in the map file to see where in memory each structure is allocated.

3. Next we develop the software algorithm, which is a sequence of operations we wish to execute. There are many approaches to describing the algorithm. Experienced programmers can develop the algorithm directly in the C language. On the other hand, most of us need an abstract method to document the desired sequence of actions. Flowcharts and pseudo-code are two common descriptive formats. There are no formal rules regarding pseudo-code, rather it is a shorthand for describing what to do and when to do it. We can place our pseudo-code as documentation into the comment fields of our program. The following figure shows a flowchart on the left and pseudo-code and C code on the right for our digital lock example.

A flowchart and corresponding pseudo-code and C code

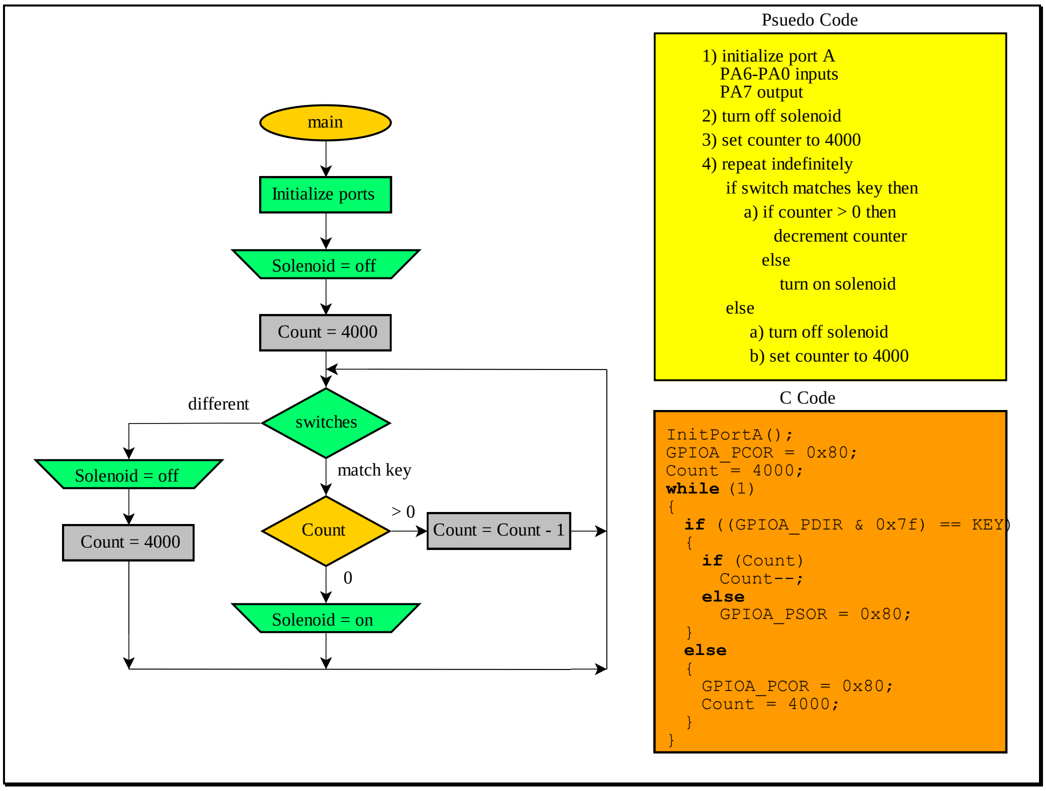


Figure 2.2 – Digital lock software

Normally we place the programs in Flash memory. Typically, the compiler will initialize the stack pointer to the last location of RAM. On the K64, the stack is initialized to `0x20000000` (the starting address of the internal SRAM). Next we write C code to implement the algorithm as illustrated in the above flowchart and pseudo-code.

4. The last stage is debugging.

2.1.2 Case Study 2: A Serial Port K64 Program

Let's begin with a small program. This simple program is typical of the operations we perform in an embedded system. This program will read 8-bit data from parallel port C and transmit the information in serial fashion using the Universal Asynchronous Receiver/Transmitter number 2 (UART2). The numbers in the first column are not part of the software, but have been added to simplify our discussion.

Sample serial port program for the K64

```c
/* Translates parallel input data to serial outputs */

#define GPIOC_PDIR *(uint32_t volatile *)(0x400FF090)
#define GPIOC_PDDR *(uint32_t volatile *)(0x400ff094)
#define UART2_BD   *(uint16_t volatile *)(0x4006C000)
#define UART2_C2   *(uint8_t  volatile *)(0x4006C003)
#define UART2_S1   *(uint8_t  volatile *)(0x4006C004)
#define UART2_D    *(uint8_t  volatile *)(0x4006C007)
#define TDRE 0x80

void UART_Init(void)
{
  /* 9600 baud, 16 MHz Xtal assumed */
  UART2_BD = 0x34;
  /* enable UART, no interrupts */
  UART2_C2 = 0x0C;
}

/* Data is 8 bit value to send out serial port */
void UART_Out(const uint8_t data)
{
  /* Wait for TDRE to be set */
  while ((UART2_S1 & TDRE) == 0);
  /* then output */
  UART2_D = data;
}

void main(void)
{
  unsigned char info;
  /* initialise serial port */
  UART_Init();
  /* specify Port C as input */
  GPIOC_PDDR = 0;
  while (1)
  {
    /* input 8 bits from parallel port C */
    info = (unsigned char)GPIOC_PDIR;
    /* output 8 bits to serial port */
    UART_Out(info);
  }
 }
```

Listing 2.1 – Sample K64 Program

<u>**Note**</u>: This program is *vastly* simplified and will *not* run on a K64.

The first line of the program is a *comment* giving a brief description of its function. Lines 3 through 8 define *macros* that provide programming access to ports and registers of the K64. These macros specify the format (unsigned 8, 16 or 32 bit) and address (the K64 employs memory mapped I/O). For example, in line 3 the `#define` invokes the preprocessor to replace each instance of `GPIOC_PDIR` with `*(uint32_t volatile *)(0x400FF090)`.

Lines 11-17 define a *function* or procedure that when executed will initialize the UART port. The assignment statement is of the form `value at address = data`. In particular line 14 (`UART2_BD = 0x34;`) will output a hexadecimal `0x34` to I/O configuration register at location `0x4006C000`. Similarly line 16 will output a hexadecimal `0x0C` to I/O configuration register at location `0x4006C003`. Notice that comments can be added virtually anywhere in order to clarify the software function. `UART_Init` is an example of a function that is executed only once at the beginning of the program.

Line 9 is another `#define` that specifies the transmit data ready empty (`TDRE`) bit as bit 7. This `#define` illustrates the usage of macros that make the software more readable. Line 19 is a comment. Lines 20-26 define another function, `UART_Out`, having an 8-bit input parameter that when executed will output the data to the UART2 port. In particular line 23 will read the UART2 status register at `0x4006C004` over and over again until bit 7 (`TDRE`) is set. Once `TDRE` is set, it is safe to start another serial output transmission. This is an example of I/O polling. Line 25 copies the input parameter, `data`, to the serial port, starting a serial transmission. Line 25 is an example of an I/O output operation.

Lines 28 through 42 define the main program. After some brief initialization this is where the software will start after a reset or after being powered up. The sequence `unsigned char info` in line 30 will define a local variable. Notice that the size (`char` means 8-bit), type (`unsigned`) and name (`info`) are specified. Line 32 calls the initialization function **`UART_Init`**. Line 34 writes a `0` to the I/O configuration register at `0x400ff094`, specifying all 32 bits of `PORTC` will be inputs (writing ones to a direction register specifies the bits as outputs). The sequence `while (1) {...}` defines a control structure that executes forever and never finishes. In particular lines 37 to 40 are repeated over and over without end. Most software on embedded systems will run forever (or until the power is removed). Line 38 will read Port C and copy the voltage levels into the variable `info`. This is an example of an I/O input operation. Each of the lower 8 lines of the 32-bit PORTC corresponds to one of the 8 bits of the variable `info`. A digital logic high (a voltage above 2 V), is translated into a `1`. A digital logic low (a voltage less than 0.7 V) is translated into a `0`. Line 40 will execute the function **`UART_Out`** that will transmit the 8-bit data via the UART2 serial port.

With the MCUXpresso (MX) IDE, the system installs a reset vector address and will create code to initialize then jump to the main program automatically.

2.1.3 Free field language

In some programming languages the column position and line number affect the meaning. On the contrary, C is a free field language. Except for preprocessor lines (that begin with **#**), spaces, tabs and line breaks have the same meaning. The other situation where spaces, tabs and line breaks matter is string constants. We cannot type tabs or line breaks within a string constant. This means we can place more than one statement on a single line, or place a single statement across multiple lines. For example the function **`UART_Init`** could have been written without any line breaks

```
void UART_Init(void){UART2_BD=0x34; UART2_C2=0x0C;}
```

Since we never make hardcopy printouts of our software, it is *not* necessary to minimize the number of line breaks.

(2.1) Use a programming style that is easy to read

Similarly we could have added extra line breaks:

```
void UART_Init(void)
{
UART2_BD=
   0x34;
UART2_C2=
   0x0C;
}
```

Just because C allows such syntax, it does not mean it is desirable. After much experience you will develop a programming style that is easy to understand. Although spaces, tabs, and line breaks are syntactically equivalent, their proper usage will have a profound impact on the readability of your software.

A *token* in C can be a user defined name (e.g., the variable `info` and function `UART_Init`) or a predefined operation (e.g., `*`, `unsigned`, `while`). Each token must be contained on a single line. We see in the above example that tokens can be separated by white spaces (space, tab, line break) or by the special characters, which we can subdivide into punctuation marks and operations. Punctuation marks (semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses) are very important in C. It is one of the most frequent sources of errors for both beginning and experienced programmers.

Punctuation marks separate tokens

Punctuation	Name	Meaning
;	semicolon	End of statement
:	colon	Defines a label
,	comma	Separates elements of a list
( )	parentheses	Start and end of a parameter list
{ }	braces	Start and stop of a compound statement
[ ]	brackets	Start and stop of an array index
" "	quotation marks	Start and stop of a string
' '	apostrophes	Start and stop of a character constant

Table 2.1 – Special characters can be punctuation marks

The next table shows the single character operators.

Special characters can be operators

Operation	Name	Meaning
=	equals	Assignment statement
@	at	Address of
?	question mark	Selection
<	less than	Less than
>	greater than	Greater than
!	exclamation mark	Logical not (true to false, false to true)
~	tilde	1's complement
+	plus	Addition
-	minus	Subtraction
*	asterisk	Multiplication or pointer dereference
/	back slash	division
%	percent	Modulo, division remainder
|	pipe	Bitwise OR
&	ampersand	Bitwise AND, or address of
^	hat	Bitwise XOR
.	period	Used to access parts of a structure

Table 2.2 – Special characters can be operators

The next table shows the operators formed with multiple characters.

Multiple special characters can also be operators

Operation	Name	Meaning
==	is equal to	Equal to comparison
<=	less than or equal to	Less than or equal to
>=	greater than or equal to	Greater than or equal to
!=	not equal to	Not equal to
<<	shift left	Shift left
>>	shift right	Shift right
++	plus plus	Increment
--	minus minus	Decrement
&&	logical and	Boolean AND
||	logical or	Boolean OR
+=	plus equals	Add value to
-=	minus equals	Subtract value to
*=	asterisk equals	Multiply value to
/=	slash equals	Divide value to
|=	pipe equals	Bitwise OR value to
&=	ampersand equals	Bitwise AND value to
^=	hat equals	Bitwise XOR value to
<<=	shift left equals	Shift value left
>>=	shift right equals	Shift value right
%=	percent equals	Modulo divide value to
->	Arrow	Pointer to a part of a structure

Table 2.3 – Multiple special characters also can be operators

The following section illustrates some of the common operators. We begin with the assignment operator.

```
/* Three variables */
short x, y, z;
void Example(void)
{
  /* set the value of x to 1 */
  x = 1;
  /* set the value of y to 2 */
  y = 2;
  /* set the value of z to the value of x (both are 1) */
  z = x;
  /* all three to zero */
  x = y = z = 0;
}
```

Listing 2.2 – Simple program illustrating C arithmetic operators

Notice that in the line `x = 1;`, `x` is on the left hand side of the `=`. This specifies the address of `x` is the destination of assignment. On the other hand, in the line `z = x;`, `x` is on the right hand side of the `=`. This specifies the value of `x` will be assigned into the variable `z`. Also remember that the line `z = x;` creates two copies of the data. The original value remains in `x`, while `z` also contains this value.

Next we will introduce the arithmetic operations addition, subtraction, multiplication and division. The standard arithmetic precedencies apply.

Arithmetic operators

```
/* Three variables */
short x, y, z;
void Example(void)
{
  /* set the values of x and y */
  x = 1; y = 2;
  /* arithmetic operation */
  z = x + 4 * y;
  /* same as x = x + 1; */
  x++;
  /* same as y = y - 1; */
  y--;
  /* left shift same as x = 4 * y; */
  x = y << 2;
  /* right shift same as x = y / 4; */
  z = y >> 2;
  /* same as y = y + 2; */
  y += 2;
}
```

Listing 2.3 – Simple program illustrating C arithmetic operators

2.14

Next we will introduce a simple conditional control structure.

The C **if-else** control structure

```
#define GPIOB_PSOR *(uint32_t volatile *)(0x 400FF044)
#define GPIOE_PDIR *(uint32_t volatile *)(0x 400FF110)

void Example(void)
{
  /* test bit 2 of PORTE */
  if ((GPIOE_PDIR & 0x00000004) == 0)
  {
    /* if PORTE bit 2 is 0, then make PORTB = 0 */
    GPIOB_PSOR = 0;
  }
  else
  {
    /* if PORTE bit 2 is not 0, then make PORTB = 100 */
    GPIOB_PSOR = 100;
  }
}
```

Listing 2.4 – The C `if-else` control structure

`GPIOB_PSOR` is an output port, and `GPIOE_PDIR` is an input port on the K64. The expression `(GPIOE_PDIR & 0x00000004)` will return `0` if `PORTE` bit 2 is `0` and will return a `4` if `PORTE` bit 2 is `1`. The expression `(GPIOE_PDIR & 0x00000004) == 0` will return `TRUE` if `PORTE` bit 2 is `0` and will return a `FALSE` if `PORTE` bit 2 is `1`. The statement immediately following the **`if`** will be executed if the condition is `TRUE`. The **`else`** statement is optional.

Like the `if` statement, the `while` statement has a conditional test (i.e., returns a `TRUE`/`FALSE`).

```
#define GPIOA_PTOR *(uint32_t volatile *)(0x400FF00C)

void Example(void)
{
  unsigned char counter;
   /* loop until counter equals 200 */
  counter = 0;
  while (counter != 200)
  {
    /* toggle PORTA bit 3 output */
    GPIOA_PTOR = 0x00000008;
    /* increment counter */
    counter++;
  }
}
```

The C `while` control structure

Listing 2.5 – The C `while` control structure

`GPIOA_PTOR` is a register used to toggle the `PORTA` pins on the K64. The statement immediately following the `while` will be executed over and over until the conditional test becomes `FALSE`.

The `for` control structure has three control expressions and a body.

```
#define GPIOA_PTOR *(uint32_t volatile *)(0x400FF00C)

void Example(void)
{
  unsigned char counter;
  /* loop until counter equals 200 */
  for (counter = 0; counter < 200; counter++)
  {
    /* toggle PORTA bit 3 output */
    GPIOA_PTOR = 0x00000008;
  }
}
```

The C `for` loop control structure

Listing 2.6 – The C `for` loop control structure

The *initializer expression*, `counter = 0`, is executed once at the beginning.

The *loop test expression*, `counter < 200`, is evaluated at the beginning of each iteration through the loop, and if it is `FALSE` then the loop terminates.

Then the body, `GPIOA_PTOR = 0x00000008;`, is executed.

Finally, the *counting expression*, `counter++`, is evaluated at the end of each loop iteration and is usually responsible for altering the loop variable.

2.1.4 Precedence

As with all programming languages the order of the tokens is important. There are two issues to consider when evaluating complex statements. The *precedence* of the operator determines which operations are performed first.

Precedence of operators in C

```
short Example(short x, short y)
{
  short z;
  z = y + 2 * x;
  return z;
}
```

In the preceding example, the `2 * x` is performed first because `*` has higher precedence than `+` and `=`. The addition is performed second because `+` has higher precedence than `=`. The assignment `=` is performed last. Sometimes we use parentheses to clarify the meaning of the expression, even when they are not needed. Therefore, the line `z = y + 2 * x;` could also have been written `z = 2 * x + y;` or `z = y + (2 * x);` or `z = (2 * x) + y;`.

2.1.5 Associativity

Associativity determines the left to right or right to left order of evaluation when multiple operations of the precedence are combined. For example + and − have the same precedence, so how do we evaluate the following?

```
z = y - 2 + x;
```

We know that + and − associate from left to right. This function is the same as `z = (y - 2) + x;`, meaning the subtraction is performed first because it is more to the left than the addition. Most operations associate left to right, but the following table illustrates that some operators associate right to left.

Precedence and associativity determine the order of operation

Precedence	Operators	Associativity
highest	`() [] . ->` `++(postfix) --(postfix)`	left to right

	`++(prefix) --(prefix) ! ~` **`sizeof`**`(type)`
	`+(unary) -(unary) &(address)`
	`*(dereference)`
		right to left

	`* / %`	left to right
	`+ -`	left to right
	`<< >>`	left to right
	`< <= > >=`	left to right
	`== !=`	left to right
	`&`	left to right
	`^`	left to right
	`|`	left to right
	`&&`	left to right
	`||`	left to right
	`?:`	right to left
	`= += -= *= /= %= <<= >>=` `|= &= ^=`	right to left
lowest	`,`	left to right

Table 2.4 – Precedence and associativity determine the order of operation

When confused about precedence (and aren't we all?) add parentheses to clarify the expression. (2.2)

2.1.6 Comments

There are two types of comments. The first type explains how to use the software. These comments are usually placed at the top of the file, within the header file, or at the start of a function. The reader of these comments will be writing software that uses or calls these routines. Lines 1 and 19 in Listing 2.1 are examples of this type of comment. The second type of comments assist a future programmer (ourselves included) in changing, debugging or extending these routines. We usually place these comments within the body of the functions. The comments above each line in Listing 2.1 are examples of the second type. We place comments on separate lines so that the implementation is separate from the explanation.

Comments in the C language

Comments begin with the `/*` sequence and end with the `*/` sequence. They may extend over multiple lines as well as exist in the middle of statements. The following is the same as `UART2_BD = 0x34;`

```
UART2_BD /*specifies baud rate*/=0x34/*9600 bits/sec*/;
```

Some compilers do allow for the use of C++ style comments. The start comment sequence is `//` and the comment ends at the next line break or end of file. Thus, the following two lines are equivalent:

Comments in the C++ language

```
UART_Init();  /* turn on UART serial port */
UART_Init();  // turn on UART serial port
```

We will assume (for the sake of clarity) that C++ comments are allowed in this document from now on!

C does allow the comment start and stop sequences within character constants and string constants. For example the following string contains all seven characters, not just the `ac`:

```
const char str[10]="a/*b*/c";
```

Some compilers unfortunately do not support comment nesting. This makes it difficult to comment out sections of logic that are themselves commented.

For example, the following attempt to comment-out the call to `UART_Init` will result in a compiler error.

```
void main(void)
{
  unsigned char info;
/*
  /* turn on UART serial port */
  UART_Init();
*/
  /* specify Port C as input */
  GPIOC_PDDR = 0;
  while (1)
  {
    // input 8 bits from parallel port C
    info = (unsigned char)GPIOC_PDIR;
    // output 8 bits to serial port
    UART_Out(info);
  }
}
```

The *conditional compilation* feature of a compiler can be used to temporarily remove and restore blocks of code.

2.1.7 Preprocessor Directives

Preprocessor directives begin with # in the first column. As the name implies preprocessor commands are processed first, i.e., the compiler passes through the program handling the preprocessor directives. We have already seen the macro definition (`#define`) used to define I/O ports and bit fields. A second important directive is the `#include`, which allows you to include another entire file at that position within the program. The following directive will define all the K64 I/O port names.

Preprocessor directives are processed first by the compiler

```
#include <MK64F12.h>
```

2.1.8 Global Declarations

An object may be a data structure or a function. Objects that are not defined within functions are global. Objects that may be declared in MX include:

Global declaration objects – data or functions

- integer variables (16-, 32- or 64-bit signed or unsigned)
- character variables (8-bit signed or unsigned)
- arrays of integers or characters
- pointers to integers or characters
- arrays of pointers
- structure (grouping of other objects)
- unions (redefinitions of storage)
- functions

MX supports 32-bit **long** integers, 64-bit **long long** integers, and single- and double-precision floating point types. We will focus on 8-, 16- and 32-bit objects. The object code generated with the compiler is often more efficient using 32-bit parameters rather than 8- or 16-bit ones.

2.1.9 Declarations and Definitions

Declarations specify an object – definitions define what they are (for data) or what they do (for functions)

It is important for the C programmer to distinguish the two terms *declaration* and *definition*. A function declaration specifies its name, its input parameters and its output parameter. Another name for a function declaration is *prototype*. A data structure declaration specifies its type and format. On the other hand, a function definition specifies the exact sequence of operations to execute when it is called. A function definition will generate object code (machine instructions to be loaded into memory that perform the intended operations). A data structure definition will reserve space in memory for it. The confusing part is that the definition will repeat the declaration specifications. We can declare something without defining it, but we cannot define it without declaring it. For example the declaration for the function **UART_Out** could be written as:

```
void UART_Out(const unsigned char);
```

We can see that the declaration shows us how to use the function, not how the function works. Because the C compilation is a one-pass process, an object must be declared or defined before it can be used in a statement. (Actually the preprocess performs a pass through the program that handles the preprocessor directives.) Notice that the function **UART_Out** was defined before it was used in Listing 2.1. The following alternative approach first declares the functions, uses them, and lastly defines the functions:

```
// Translates parallel input data to serial outputs
#define GPIOC_PDIR *(uint32_t volatile *)(0x400FF090)
#define GPIOC_PDDR *(uint32_t volatile *)(0x400ff094)
#define UART2_BD   *(uint16_t volatile *)(0x4006C000)
#define UART2_C2   *(uint8_t  volatile *)(0x4006C003)
#define UART2_S1   *(uint8_t  volatile *)(0x4006C004)
#define UART2_D    *(uint8_t  volatile *)(0x4006C007)
#define TDRE 0x80

void UART_Init(void);
void UART_Out(const unsigned char);

void main(void)
{
  unsigned char info;
  // turn on UART serial port
  UART_Init();
  // specify Port C as input
  GPIOC_PDDR = 0;
  while (1)
  {
    // input 8 bits from parallel port C
    info = (unsigned char)GPIOC_PDIR;
    // output 8 bits to serial port
    UART_Out(info);
  }
}

void UART_Init(void)
{
  // 9600 baud
  UART2_BD = 0x34;
  // enable UART, no interrupts
  UART2_C2 = 0x0C;
}

// Data is 8 bit value to send out serial port
void UART_Out(const uint8_t data)
{
  // Wait for TDRE to be set
  while ((UART2_S1 & TDRE) == 0);
  // then output
  UART2_D = data;
}
```

Function declarations

Function definitions

Listing 2.7 – Alternate C program

2.22

An object may be said to exist in the file in which it is defined, since compiling the file yields a module containing the object. On the other hand, an object may be declared within a file in which it does not exist. Declarations of data structures that are defined elsewhere are preceded by the keyword `extern`. Thus

```
short RunFlag;
```

defines a 16-bit signed integer called `RunFlag`, whereas

```
extern short RunFlag;
```

only declares `RunFlag` to exist in another, separately compiled, module.

Likewise, we can use external function declarations to access a function in another module. Thus the line

```
extern void PITHandler(void);
```

Use `extern` to specify that an object is defined elsewhere

declares the function name and type just like a regular function declaration. The `extern` tells the compiler that the actual function exists in another module and the linker will combine the modules so that the proper action occurs at run time. The compiler knows everything about `extern` objects except where they are. The linker is responsible for resolving that discrepancy. The compiler simply tells the assembler that the objects are in fact external. And the assembler, in turn, makes this known to the linker.

2.1.10 Functions

A *function* is a sequence of operations that can be invoked from other places within the software. We can pass 0 or more parameters into a function. The code generated by the MX pass the first few input parameters in Registers R0-R3 and the remaining parameters are passed on the stack. A function can have 0 or 1 output parameter. The code generated by MX pass the return parameter in Register R0 (8- and 16-bit return parameters are promoted to 32-bits.) The add function below has two 16-bit signed input parameters, and one 16-bit output parameter. Again the numbers in the first column are not part of the software, but added to simplify our discussion.

Functions use parameters to receive input values, and sometimes return a single value

```
short add(short x, short y)
{
  short z;
  z = x + y;
  if ((x > 0) && (y > 0) && (z < 0))
    z=32767;
  if ((x < 0) && (y < 0) && (z > 0))
    z=-32768;
  return z;
}

void main(void)
{
  short a, b;
  a = add(2000, 2000);
  b = 0;
  while (1)
  {
    b = add(b, 1);
  }
}
```

Listing 2.8 – Example of a function call

2.24

The interesting part is that after the operations within the function are performed control returns to the place right after where the function was called. In C, execution begins with the `main` program. The execution sequence is shown below:

```
12 void main(void)
13 {
14   short a, b;
15   a = add(2000, 2000);           // call to add
1  short add(short x, short y)
2  {
3    short z;
4    z = x + y;                     // z = 4000
5    if ((x > 0) && (y > 0) && (z < 0))
6      z=32767;
7    if ((x < 0) && (y < 0) && (z > 0))
8      z=-32768;
9    return z;
10 }  // return 4000 from call
16   b = 0;
17   while (1)
18   {
19     b = add(b, 1);               // call to add
1  short add(short x, short y)
2  {
3    short z;
4    z = x + y;                     // z = 1
5    if ((x > 0) && (y > 0) && (z < 0))
6      z=32767;
7    if ((x < 0) && (y < 0) && (z > 0))
8      z=-32768;
9    return z;
10 }  // return 1 from call
20   }
17   while (1)
18   {
19     b = add(b, 1);               // call to add
1  short add(short x, short y)
2  {
3    short z;
4    z = x + y;                     // z = 2
5    if ((x > 0) && (y > 0) && (z < 0))
6      z=32767;
7    if ((x < 0) && (y < 0) && (z > 0))
8      z=-32768;
9    return z;
10 }  // return 2 from call
20   }
```

Notice that the return from the first call goes to line 16, while all the other returns go to line 20. The execution sequence repeats lines 17, 18, 19, 1-10, 20 indefinitely.

C does not allow for the nesting of procedural declarations. In other words you cannot define a function within another function. In particular all function declarations must occur at the global level.

A function definition consists of two parts: a *declaration specifier* and a *body*. The declaration specifier states the return type, the name of the function and the names of arguments passed to it. The names of the argument are only used inside the function. In the **add** function above, the declaration specifier is `short add(short x, short y)` meaning it has one 16-bit output parameter, and two 16-bit input parameters.

A function definition has two parts – a declarator and a body

The parentheses are required even when there are no arguments. The following four statements are equivalent:

```
void UART_Init(void){UART2_BD=0x34; UART2_C2=0x0C;}
UART_Init(void){UART2_BD=0x34; UART2_C2=0x0C;}
void UART_Init(){UART2_BD=0x34; UART2_C2=0x0C;}
UART_Init(){UART2_BD=0x34; UART2_C2=0x0C;}
```

The `void` should be included as the return parameter if there is none, because it is a positive statement that the function does not return a parameter. When there are no arguments, a `void` should be specified to make a positive statement that the function does not require parameters.

The body of a function consists of a statement that performs the work. Normally the body is a compound statement between a `{}` pair. If the function has a return parameter, then all exit points must specify what to return.

The program created by the MX compiler actually begins execution at a place called **`thumb_startup()`**. After a power on or hardware reset, the embedded system will initialize the stack and clear all global variables. After this brief initialization sequence the function named **`main()`** is called. Consequently, there must be a **`main()`** function somewhere in the program. If you are curious about what really happens, look in the file `startup.c`. For programs not in an embedded environment (e.g., running on your PC) a return from **`main()`** transfers control back to the operating system. As we saw earlier, software for an embedded system usually does not quit.

The C compiler always calls a `thumb_startup()` function before calling `main()`

2.1.11 Compound Statements

A compound statement is enclosed by { }

A *compound statement* (or *block*) is a sequence of statements, enclosed by braces, that stands in place of a single statement. Simple and compound statements are completely interchangeable as far as the syntax of the C language is concerned. Therefore, the statements that comprise a compound statement may themselves be compound; that is, blocks can be nested.

Compound statements may be nested

Thus, it is legal to write

```
// 3 wide 16 bit signed median filter
short median(const short n1, const short n2, const short n3)
{
  if (n1 > n2)
  {
    if (n2 > n3)
      return n2;      // n1>n2,n2>n3    n1>n2>n3
    else
    {
      if (n1 > n3)
        return n3;    // n1>n2,n3>n2,n1>n3 n1>n3>n2
      else
        return n1;    // n1>n2,n3>n2,n3>n1 n3>n1>n2
    }
  }
  else
  {
    if (n3 > n2)
      return n2;      // n2>n1,n3>n2     n3>n2>n1
    else
    {
      if (n1 > n3)
        return n1;    // n2>n1,n2>n3,n1>n3 n2>n1>n3
      else
        return n3;    // n2>n1,n2>n3,n3>n1 n2>n3>n1
    }
  }
}
```

Listing 2.9 – Example of nested compound statements

Although C is a free-field language, notice how the indenting has been added to the above example. The purpose of this indenting is to make the program easier to read. On the other hand since C is a free-field language, the following two statements are quite different

```
if (n1 > 100) n2 = 100; n3 = 0;
if (n1 > 100) {n2 = 100; n3 = 0;}
```

In both cases `n2 = 100;` is executed if `n1 > 100`. In the first case the statement `n3 = 0;` is always executed, while in the second case `n3 = 0;` is executed only if `n1 > 100`.

2.1.12 Global Variables

Variables declared outside of a function, like `Count` in the following example, are properly called *external* variables because they are defined outside of any function. While this is the standard term for these variables, it is confusing because there is another class of external variable, one that exists in a separately compiled source file. We will refer to variables in the current source file as *globals*, and we will refer to variables defined in another file as externals.

There are two reasons to employ global variables. The first reason is data permanence. The other reason is information sharing. Normally we pass information from one module to another explicitly using input and output parameters, but there are applications like interrupt programming where this method is unavailable. For these situations, one module can store data into a global while another module can view it.

Global variables are used for data sharing between modules

In the following example, we wish to maintain a counter of the number of times `UART_Out` is called. This data must exist for the entire life of the program. This example also illustrates that with an embedded system it is important to initialize RAM-based globals at run time. Most C compilers (including MX) will automatically initialize globals to zero at startup.

Global variables are initialized by the `Init()` function

```
// number of characters transmitted
unsigned short Count;
#define TDRE 0x80

void UART_Init(void)
{
  // initialize global counter
  Count = 0;
  // 9600 baud
  UART2_BD=0x34;
  // enable UART, no interrupts
  UART2_C2=0x0C;
}
```

```c
void UART_Out(const uint8_t data)
{
    // Incremented each time
  Count = Count + 1;
  // Wait for TDRE to be set
  while ((UART2_S1 & TDRE) == 0);
  // then output
  UART2_D = data;
}
```

Listing 2.10 – A global variable contains permanent information

Although the following two examples are equivalent, the second case is preferable because its operation is more self-evident. In both cases the global is allocated in RAM, and initialized at the start of the program to 1.

```c
short Flag = 1;
void main(void)
{
  // main body goes here
}
```

Listing 2.11 – A global variable initialized at run-time by the compiler

```c
short Flag;
void main(void)
{
  Flag = 1;
  // main body goes here
}
```

Listing 2.12 – A global variable initialized at run-time by the compiler

From a programmer's point of view, we usually treat the I/O ports in the same category as global variables because they exist permanently and support shared access.

2.1.13 Local Variables

Local variables are very important in C programming. They contain temporary information that is accessible only within a narrow scope. We can define local variables at the start of a compound statement. We call these *local variables* since they are known only to the block in which they appear, and to subordinate blocks. The following statement adjusts `x` and `y` such that `x` contains the smaller number and `y` contains the larger one. If a swap is required then the local variable `z` is used.

Local variables are allocated on the stack and contain temporary information

```
if (x > y)
{
  // create a temporary variable
  short z;
  // swap x and y
  z = x; x = y; y = z;
  // then destroy z
}
```

Notice that the local variable `z` is declared within the compound statement. Unlike globals, which are said to be *static*, locals are created dynamically when their block is entered, and they cease to exist when control leaves the block. Furthermore, local names supersede the names of globals and other locals declared at higher levels of nesting. Therefore, locals may be used freely without regard to the names of other variables. Although two global variables cannot use the same name, a local variable of one block can use the same name as a local variable in another block. Programming errors and confusion can be avoided by understanding these conventions.

Local variables are local to their block and supersede the names of variables at higher levels

2.1.14 Source Files

Software usually consists of many source files

Our programs may consist of source code located in more than one file. The simplest method of combining the parts together is to use the **#include** preprocessor directive. Another method is to compile the source files separately, then combine the separate object files as the program is being linked with library modules. The linker/library method should normally be used, as only small pieces of software are changed at a time. The MX supports the automatic linking of multiple source files once they are added to a project. Remember that a function or variable must be defined or declared before it can be used. The following example is one method of dividing our simple example into multiple files.

```
// **** file MK64F12.h ************
#define GPIOC_PDIR *(uint32_t volatile *)(0x400FF090)
#define GPIOC_PDDR *(uint32_t volatile *)(0x400ff094)
#define UART2_BD   *(uint16_t volatile *)(0x4006C000)
#define UART2_C2   *(uint8_t  volatile *)(0x4006C003)
#define UART2_S1   *(uint8_t  volatile *)(0x4006C004)
#define UART2_D    *(uint8_t  volatile *)(0x4006C007)
```

Listing 2.13 – Header file for K64 I/O ports

```
// **** file UART.h ************
void UART_Init(void);
void UART_Out(const uint8_t data);
```

Listing 2.14 – Header file for the UART interface

```
// **** file UART.c ************
void UART_Init(void)
{
  // 9600 baud
  UART2_BD=0x34;
  // enable UART, no interrupts
  UART2_C2=0x0C;
}

// Data is an 8-bit value to send out the serial port
#define TDRE 0x80
void UART_Out(const uint8_t data)
{
  // Wait for TDRE to be set
  while ((UART2_S1 & TDRE) == 0);
  // then output
  UART2_D = data;
}
```

Listing 2.15 – Implementation file for the UART interface

```c
// **** file my.c ************
// Translates parallel input data to serial outputs
#include "MK64F12.h"
#include "UART.h"

void main(void)
{
  unsigned char info;

  // turn on UART serial port
  UART_Init();

  // specify Port C as input
  GPIOC_PDDR = 0;
  while (1)
  {
    // input 8 bits from parallel port C
    info = (unsigned char)GPIOC_PDIR;
    // output 8 bits to serial port
    UART_Out(info);
  }
}
```

Listing 2.16 – Main program file for this system

This division of functions across multiple source files is clearly a matter of style.

If the software is easy to understand, debug and change, then it is written with good style. (2.3)

While the main focus of this section is on C syntax, it would be improper to neglect all style issues. This system was divided using the following principles:

Software style principles

- Define the I/O ports in a `MK64F12.h` header file
- For each module place the user-callable prototypes in a `*.h` header file
- For each module place the implementations in a `*.c` program file
- In the main program file, include the header files first

Breaking a software system into files has a lot of advantages. The first reason is code reuse. Consider the code in this example. If a UART output function is needed in another application, then it would be a simple matter to reuse the `UART.h` and `UART.c` files. The next advantage is clarity. Compare the main program in Listing 2.16 with the entire software system in Listing 2.1. Since the details have been removed, the overall approach is easier to understand.

2.32

The next reason to break software into files is parallel development. As the software system grows it will be easier to divide up a software project into subtasks, and to recombine the modules into a complete system if the subtasks have separate files. The last reason is upgrades. Consider an upgrade in our simple example where the 9600 bits/sec serial port is replaced with a high-speed Universal Serial Bus (USB). For this kind of upgrade we implement the USB functions then replace the `UART.c` file with the new version. If we plan appropriately, we should be able to make this upgrade without changes to the files `UART.h` and `my.c`.

2.2 Tokens

This section defines the basic building blocks of a C program. Understanding the concepts in this section will help eliminate the syntax bugs that confuse even the veteran C programmer. A simple syntax error can generate 100's of obscure compiler errors.

To understand the syntax of a C program, we divide it into *tokens* separated by *white spaces* and *punctuation*. Remember that white space includes the space, tab, carriage return and line feed. A token may be a single character or a sequence of characters that form a single item. The first step of a compiler is to process the program into a list of tokens and punctuation marks. The following example includes punctuation marks of `( ) { } ;`. The compiler then checks for proper syntax. Finally, it creates object code that performs the intended operations. Consider the following example:

The C language is written using tokens separated by whitespace

```
void main(void)
{
  short z;
  z = 0;
  while (1)
  {
    z = z + 1;
  }
}
```

Listing 2.17 – Example program with just a few tokens

The following sequence shows the tokens and punctuation marks from the above listing:

```
void main ( ) { short z ; z = 0 ; while ( 1 ) { z = z + 1 ; } }
```

Since tokens are the building blocks of programs, we begin our revision of the C language by defining its tokens.

2.2.1 ASCII Character Set

Like most programming languages C uses the standard ASCII character set. The following table shows the 128 standard ASCII codes. One or more *white space* can be used to separate tokens and or punctuation marks. The white space characters in C include horizontal tab (9=0x09), the carriage return (13=0x0D), the line feed (10=0x0A), and space (32=0x20).

ASCII character codes

Bits 0 to 3	Bits 4 to 6							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table 2.5 – ASCII Character Codes

The first 32 (values 0 to 31 or 0x00 to 0x1F) and the last one (127=0x7F) are classified as *control characters*. Codes 32 to 126 (or 0x20 to 0x7E) include the "normal" characters. Normal characters are divided into

The ASCII codes are divided into control characters and normal characters

- the space character (32=0x20),
- the numeric digits 0 to 9 (48 to 57 or 0x30 to 0x39),
- the uppercase alphabet A to Z (65 to 90 or 0x41 to 0x5A),
- the lowercase alphabet a to z (97 to122 or 0x61 to 0x7A), and
- the special characters (all the rest).

2.2.2 Literals

Numeric literals consist of an uninterrupted sequence of digits delimited by white spaces or special characters (operators or punctuation). Although MX does support floating-point numbers, this document will not cover them. The use of floating-point numbers requires either the use of a hardware floating-point unit (FPU), or a software library that consumes a substantial amount of program memory and execution time. Many applications are implemented using integer maths only to save cost and power. Consequently the period (".") will not appear in numbers as described in this document, since we will be focusing on integer algorithms.

Literals are constants of a particular data type

Character literals are written by enclosing an ASCII character in apostrophes (single quotes). We would write `'a'` for a character with the ASCII value of the lowercase `a` (97). The control characters can also be defined as constants. For example, `'\t'` is the tab character.

String literals are written as a sequence of ASCII characters bounded by quotation marks (double quotes). Thus, `"ABC"` describes a string of characters containing the first three letters of the alphabet in uppercase.

2.2.3 Keywords

Keywords are predefined tokens

There are some predefined tokens, called *keywords*, that have specific meaning in C programs. The reserved words we will cover in this document are:

Keyword	Meaning
`__asm`	Insert assembly code.
`auto`	Specifies a variable as automatic (created on the stack).
`break`	Causes the program control structure to finish.
`case`	One possibility within a `switch` statement.
`char`	8-bit integer.
`const`	Defines a global parameter as a constant in Flash, and defines a local parameter as a fixed value.
`continue`	Causes the program to go to beginning of loop.
`default`	Used in `switch` statement for all other cases.
`do`	Used for creating program loops.
`double`	Specifies a variable as double precision floating point.
`else`	Alternative part of a conditional.
`extern`	Defined in another module.
`float`	Specifies a variable as single precision floating point.
`for`	Used for creating program loops.
`goto`	Causes program to jump to specified location.
`if`	Conditional control structure.
`int`	32-bit integer (same as `long` on the K64). It should be avoided in most cases because the implementation will vary from compiler to compiler.
`long`	32-bit integer.
`register`	Specifies how to implement a local.
`return`	Leave function.
`short`	16-bit integer.
`signed`	Specifies variable as signed (default).
`sizeof`	Built-in function returns the size of an object.
`static`	Stored permanently in memory, accessed locally.
`struct`	Used for creating data structures.
`switch`	Complex conditional control structure.
`typedef`	Used to create new data types.
`unsigned`	Always greater than or equal to zero.
`void`	Used in parameter list to mean no parameter.
`volatile`	Can change implicitly outside the direct action of the software. It disables compiler optimization, forcing the compiler to fetch a new value each time.
`while`	Used for creating program loops.

Table 2.6 – Keywords have predefined meanings

Notice that all of the keywords in C are lowercase. Notice also that as a matter of style, a mixture of upper and lowercase are used for variable names, and all uppercase for the I/O ports. It is a good programming practice not to use these keywords for your variable or function names.

C is case sensitive and all keywords are lowercase

2.2.4 Names

We use *names* to identify our variables, functions, and macros. MX names may be up to 63 characters long. Names must begin with a letter or underscore and the remaining characters must be either letters or digits. We can use a mixture of upper and lowercase or the underscore character to create self-explaining symbols, e.g.,

Names define variables, functions and macros

```
time_of_day
go_left_then_stop
TimeOfDay
GoLeftThenStop;
```

The careful selection of names goes a long way to making our programs more readable. Names may be written with both upper and lowercase letters. The names are case sensitive. Therefore the following names are different:

Names are case sensitive

```
thetemperature
THETEMPERATURE
TheTemperature
```

The practice of naming macros in uppercase calls attention to the fact that they are not variable names but defined symbols. The I/O port names are implemented as macros in the header file `MK64F12.h`.

Every global name defined with the MX is left as-is by the compiler. However, it defines certain names for its own use, such as startup code and library files, and precedes them with an underscore. The purpose of the underscore is to avoid clashes with the user's own global names. So, as a matter of practice, we should not ordinarily use names with leading underscores. For examples of this naming convention, observe the linker map file generated by the compiler (in the `*.map` file in the Debug folder in the project window).

Developing a naming convention will avoid confusion. Possible ideas to consider include:

The Systems Hungarian variable naming convention

1. Start every variable name with its type, like Systems Hungarian notation used by the Microsoft Windows API (abandoned with .NET). For example,

- b means Boolean true/false
- n means 8-bit signed integer
- u means 8-bit unsigned integer
- m means 16-bit signed integer
- v means 16-bit unsigned integer
- l means 32-bit integer
- p means 32-bit pointer (address)
- c means 8-bit ASCII character
- sz means null terminated ASCII string

2. Start every local variable with "the" or "my".

A naming convention similar to C++ objects

3. Start every global variable and function with the associated file or module name. In the following example the names all begin with `Bit_`. Notice how similar this naming convention recreates the look and feel of the modularity achieved by classes in C++.

```
/* ********** file = Bit.c *************
   Pointer implementation of a Bit_FIFO
   These routines can be used to save (Bit_Put) and recall
   (Bit_Get) binary data 1 bit at a time (a bit stream)
   Information is saved / recalled in a first in,
   first out manner
   Bit_FIFOSize is the number of 16-bit words in
   the Bit_FIFO
   The Bit_FIFO is full when it has 16*Bit_FIFOSize bits */

#define Bit_FIFOSize 4
// 16 * 4 = 64 bits of storage
// storage for Bit Stream
unsigned short Bit_FIFO[Bit_FIFOSize];
```

```
struct Bit_Pointer
{
  // 0x8000, 0x4000,...,2,1
  unsigned short mask;
  // Pointer to word containing bit
  unsigned short *pWord;
};

typedef struct Bit_Pointer Bit_PointerType;

Bit_PointerType Bit_PutPt; // Pointer of where to put next
Bit_PointerType Bit_GetPt; // Pointer of where to get next

// Bit_FIFO is empty if Bit_PutPt == Bit_GetPt
// Bit_FIFO is full if Bit_PutPt + 1 == Bit_GetPt
short Bit_Same(Bit_PointerType p1, Bit_PointerType p2)
{
  if ((p1.pWord == p2.pWord) && (p1.mask == p2.mask))
    return(1);  // yes
  return(0);    // no
}

void Bit_Init(void)
{
  Bit_PutPt.mask = Bit_GetPt.mask = 0x8000;
  Bit_PutPt.pWord = Bit_GetPt.pWord = &Bit_FIFO[0]; // Empty
}

// returns TRUE=1 if successful,
// FALSE=0 if full and data not saved
// input is Boolean FALSE if data == 0
short Bit_Put(short data)
{
  Bit_PointerType myPutPt;
  myPutPt = Bit_PutPt;
  myPutPt.mask = myPutPt.mask >> 1;
  if (myPutPt.mask == 0)
  {
    myPutPt.mask = 0x8000;
    if ((++myPutPt.pWord) == &Bit_FIFO[Bit_FIFOSize])
      // wrap
      myPutPt.pWord = &Bit_FIFO[0];
  }
  if (Bit_Same(myPutPt, Bit_GetPt))
    // Failed, Bit_FIFO was full
    return(0);
  else
  {
    if (data)
      // set bit
      (*Bit_PutPt.pWord) |= Bit_PutPt.mask;
    else
      // clear bit
      (*Bit_PutPt.pWord) &= ~Bit_PutPt.Mask;
    Bit_PutPt = myPutPt;
    return(1);
  }
}
```

```c
// returns TRUE=1 if successful,
// FALSE=0 if empty and data not removed
// output is Boolean, 0 means FALSE, nonzero is true
short Bit_Get(unsigned short *datapt)
{
  if (Bit_Same(Bit_PutPt, Bit_GetPt))
    // Failed, Bit_FIFO was empty
    return(0);
  else
  {
    *datapt = (*Bit_GetPt.pWord) & Bit_GetPt.Mask;
    Bit_GetPt.Mask = Bit_GetPt.Mask >> 1;
    if (Bit_GetPt.Mask == 0)
    {
      Bit_GetPt.Mask = 0x8000;
      if ((++Bit_GetPt.pWord) == &Bit_FIFO[Bit_FIFOSize])
        // wrap
        Bit_GetPt.pWord = &Bit_FIFO[0];
    }
    return(1);
  }
}
```

Listing 2.18 – A naming convention that creates modularity

2.2.5 Punctuation

Punctuation marks (semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses) are very important in C. It is one of the most frequent sources of errors for both the beginning and experienced programmers.

Semicolons

The semicolon is used as a statement terminator

Semicolons are used as statement terminators. Strange and confusing syntax errors may be generated when you forget a semicolon, so this is one of the first things to check when trying to remove syntax errors. Notice that one semicolon is placed at the end of every simple statement in the following example,

```c
#define GPIOB_PSOR *(uint32_t volatile *)(0x 400FF044)

void Step(void)
{
  GPIOB_PSOR = 10;
  GPIOB_PSOR = 9;
  GPIOB_PSOR = 5;
  GPIOB_PSOR = 6;
}
```

Listing 2.19 – Semicolons are used to separate statements

Preprocessor directives do not end with a semicolon since they are not actually part of the C language proper. Preprocessor directives begin in the first column with the **#** and conclude at the end of the line. The following example will fill the array `DataBuffer` with data read from the input port (`GPIOC_PDIR`). We assume in this example that Port C has been initialized as an input. Semicolons are also used in the **for** loop statement, as illustrated by:

```
void Fill(void)
{
  short j;
  for (j = 0; j < 100; j++)
  {
    DataBuffer[j] = GPIOC_PDIR;
  }
}
```

The semicolon is also used in the **for** loop

Listing 2.20 – Semicolons are used to separate fields of the `for` statement

Colons

We can define a label using the colon. Although C has a **goto** statement, its use is strongly discouraged. Software is easier to understand using the block-structured control statements (**if**, **if else**, **for**, **while**, **do while**, and **switch case**). The following example will return after the Port C input reads the same value 100 times in a row. Again we assume Port C has been initialized as an input. Notice that every time the current value on Port C is different from the previous value the counter is reinitialized.

```
char Debounce(void)
{
  short count;
  unsigned char lastData;

Start:
  count = 0; // number of times Port C is the same
  lastData = GPIOC_PDIR;
Loop:
  if (++count == 100) goto Done; // same thing 100 times
  if (lastData != GPIOC_PDIR) goto Start; // changed
  goto Loop;
Done:
  return lastData;
}
```

Listing 2.21 – Colons are used to define labels (places we can jump to)

Colons also terminate `case` and `default` prefixes that appear in switch statements. In the following example, the next output is found (the proper sequence is 10, 9, 5, 6). The default case is used to restart the pattern.

The colon is used to terminate `case` prefixes

```
unsigned char NextStep(unsigned char step)
{
  unsigned char theNext;
  switch (step)
  {
    case 10: theNext =  9; break;
    case  9: theNext =  5; break;
    case  5: theNext =  6; break;
    case  6: theNext = 10; break;
    default: theNext = 10;
  }
  return theNext;
}
```

Listing 2.22 – Colons are also used with the `switch` statement

For both applications of the colon (`goto` and `switch`), we see that a label is created that is a potential target for a transfer of control.

Commas

Commas separate items that appear in lists. We can create multiple variables of the same type. For example,

Commas are used to separate items in lists

```
unsigned short beginTime, endTime, elapsedTime;
```

Lists are also used with functions having multiple parameters (both when the function is defined and called):

```
short add(short x, short y)
{
  short z;
  z = x + y;
  if ((x > 0) && (y > 0) && (z < 0))
    z = 32767;
  if ((x < 0) && (y < 0) && (z > 0))
    z = -32768;
  return z;
}
```

```c
void main(void)
{
  short a, b;
  a = add(2000, 2000);
  b = 0;
  while (1)
  {
    b = add(b, 1);
  }
}
```

Listing 2.23 – Commas separate the parameters of a function

Lists can also be used in general expressions. Sometimes it adds clarity to a program if related variables are modified at the same place. The value of a list of expressions is always the value of the last expression in the list. In the following example, first `TheTime` is incremented, `TheDate` is decremented, then `x` is set to `k + 2`.

```c
x = (TheTime++, --TheDate, k + 2);
```

Apostrophes

Apostrophes are used to specify character literals. Assuming the function **`OutChar`** will print a single ASCII character, the following example will print the lower case alphabet:

```c
void Alphabet(void)
{
  unsigned char mych;
  for (mych = 'a'; mych <= 'z'; mych++)
  {
    OutChar(mych); // Print next letter
  }
}
```

Apostrophes are used to specify character literals

Listing 2.24 – Apostrophes are used to specify characters

Quotation marks

Quotation marks are used to specify string literals. For example

```c
// Place for 11 characters and termination
unsigned const char Msg[12] = "Hello World";
void PrintHelloWorld(void)
{
  UART_OutString("Hello World");
  UART_OutString(Msg);
}
```

Quotation marks are used to specify string literals

Listing 2.25 – Quotation marks are used to specify strings

The command `Letter = 'A';` places the ASCII code (65) into the variable `Letter`. The command `pt = "A";` creates an ASCII string and places a pointer to it into the variable `pt`.

Braces

Braces are used to create compound statements

Braces `{}` are used throughout C programs. The most common application is for creating a compound statement. Each open brace `{` must be matched with a closing brace `}`. One approach that helps to match up braces is to use indenting. Each time an open brace is used, the source code is spaced to the right by two spaces. In this way, it is easy to see at a glance the brace pairs. Examples of this approach to tabbing are the **`Bit_Put`** function within Listing 2.18 and the median function in Listing 2.9.

Brackets

Brackets enclose array dimensions and subscripts

Brackets enclose array *dimensions* (in declarations) and *subscripts* (in expressions). Thus,

```
short FIFO[100];
```

declares an integer array named `FIFO` consisting of 100 words numbered from 0 through 99, and

```
PutPt = FIFO;
```

assigns the variable `PutPt` to the address of the first entry of the array.

Parentheses

Parentheses enclose argument lists and control the order of expression evaluation

Parentheses enclose argument lists that are associated with function declarations and calls. They are required even if there are no arguments.

As with all programming languages, C uses parentheses to control the order in which expressions are evaluated. Thus, (11+3)/2 yields 7, whereas 11+3/2 yields 12. Parentheses are very important when writing expressions.

2.2.6 Operators

The special characters used as *expression operators* are covered in the operator section further on in this document. There are many operators, some of which are single characters,

Special characters are used for expression operators

```
~  !  @  %  ^  &  *  -  +  =  |  /  :  ?  <  > ,
```

while others require two characters,

```
++  --  <<  >>  <=  +=  -=  *=  /=  ==  |=  %=  &=  ^=  ||  &&  !=
```

and some even require three characters,

```
<<=  >>=
```

The multiple-character operators cannot have white spaces or comments between the characters.

The C syntax can be confusing to the beginning programmer. For example

```
z = x + y;   // sets z equal to the sum of x and y
z = x_y;     // sets z equal to the value of x_y
```

It is therefore advisable to separate operators with white space.

2.3 Numbers, Characters and Strings

This section defines the various data types supported by the compiler. Since the objective of most computer systems is to process data, it is important to understand how data is stored and interpreted by the software. We define a literal as the direct specification of the number, character, or string. For example,

The three types of literals – numeric, character and string

```
100 'a' "Hello World"
```

are examples of a number literal, a character literal and a string literal respectively. The following sections discuss the way data is stored in the computer as well as the C syntax for creating the literals. The MX compiler recognizes three types of literals (*numeric, character, string*). Numbers can be written in three bases (decimal, octal, and hexadecimal). Although the programmer can choose to specify numbers in these three bases, once loaded into the computer, all numbers are stored and processed as unsigned or signed binary. Although the C standard does not support binary literals, if you wanted to specify a binary number, you should have no trouble using either the octal or hexadecimal format. However, the latest versions of GCC support binary literals using a syntax similar to the hex syntax, except the prefix is `0b` instead of `0x`.

2.3.1 Binary representation

Binary information is represented by 1's and 0's

Numbers are stored in the computer in binary form. In other words, information is encoded as a sequence of 1's and 0's. On most computers, the memory is organized into 8-bit bytes. This means each 8-bit byte stored in memory will have a separate address. Precision is the number of distinct or different values. We express precision in "alternatives", "decimal digits", "bytes", or "binary bits". *Alternatives* are defined as the total number of possibilities. For example, an 8-bit number scheme can represent 256 different numbers. An 8-bit *digital to analog converter* (DAC) can generate 256 different analog outputs. An 8-bit *analog to digital* converter (ADC) can measure 256 different analog inputs. We use the expression 4½ decimal digits to mean about 20,000 alternatives and the expression 4¾ decimal digits to mean more than 20,000 alternatives but less than 100,000 alternatives. The ½ decimal digit means twice the number of alternatives or one additional binary bit. For example, a voltmeter with a range of 0.00 to 9.99V has a three decimal digit precision. Let the operation $\lceil x \rceil$ be the greatest integer of x. E.g., $\lceil 2.1 \rceil$ is rounded up to 3. Table 2.7 and Table 2.8 illustrate various representations of precision.

Various representations of precision

Binary Bits	Bytes	Alternatives
8	1	256
10		1,024
12		4,096
16	2	65,536
20		1,048,576
24	3	16,777,216
30		1,073,741,824
32	4	4,294,967,296
n	$\lceil n/8 \rceil$	2^n

Table 2.7 – Relationships between various representations of precision

Decimal Digits	Alternatives
3	1,000
3½	2,000
3¾	4,000
4	10,000
4½	20,000
4¾	40,000
5	100,000
n	10^n

Table 2.8 – Relationships between various representations of precision

For large numbers we use abbreviations. A binary prefix is a prefix attached before a unit symbol to multiply it by a power of 2. In computing, such a prefix is seen in combination with a unit of information (bit, byte, etc.), to indicate a power of 1024. IEC 80000-13:2008 is an international standard that defines quantities and units used in information science, and specifies names and symbols for these quantities and units, as shown in the following table.

Abbreviation	Pronunciation	Value
Ki	"kibi"	$2^{10} = 1{,}024$
Mi	"mebi"	$2^{20} = 1{,}048{,}576$
Gi	"gibi"	$2^{30} = 1{,}073{,}741{,}824$
Ti	"tebi"	$2^{40} = 1{,}099{,}511{,}627{,}776$
Pi	"pebi"	$2^{50} = 1{,}125{,}899{,}906{,}843{,}624$
Ei	"exbi"	$2^{60} = 1{,}152{,}921{,}504{,}606{,}846{,}976$

Table 2.9 – Common abbreviations for large binary numbers

2.3.2 8-bit unsigned numbers

A byte contains 8 bits

A byte is 8 bits

b7	b6	b5	b4	b3	b2	b1	b0

where each bit b7, ..., b0 is binary and has the value 1 or 0. We specify b7 as the *most significant bit* or MSB, and b0 as the *least significant bit* or LSB. If a byte is used to represent an unsigned number, then the value of the number is

The value of an unsigned byte

$$N = 128 \cdot b7 + 64 \cdot b6 + 32 \cdot b5 + 16 \cdot b4 + 8 \cdot b3 + 4 \cdot b2 + 2 \cdot b1 + b0$$

There are 256 different unsigned 8-bit numbers. The smallest unsigned 8-bit number is 0 and the largest is 255. For example, 00001010_2 is 8 + 2 or 10.

Other examples are shown in the following table.

Binary	Hex	Calculation	Decimal
00000000	0x00	0	0
01000001	0x41	64 + 1	65
00010110	0x16	16 + 4 + 2	22
10000111	0x87	128 + 4 + 2 + 1	135
11111111	0xff	128 + 64 + 32 + 16 + 8 + 4 + 2 + 1	255

Table 2.10 – Example conversions of unsigned 8-bit binary numbers

The *basis* of a number system is a subset from which linear combinations of the basis elements can be used to construct the entire set. For the unsigned 8-bit number system, the basis is

The basis of an unsigned byte

$$\{ 128, 64, 32, 16, 8, 4, 2, 1 \}$$

Converting a decimal number to binary using the unsigned basis

One way for us to convert a decimal number into binary is to use the basis elements. The overall approach is to start with the largest basis element and work towards the smallest. One by one we see whether or not we need that basis element to create our number. If we do, then we set the corresponding bit in our binary result and subtract the basis element from our number. If we do not need it, then we clear the corresponding bit in our binary result. We will work through the algorithm with the example of converting 100 to 8-bit binary. We begin with the largest basis element (in this case 128) and see whether or not we need to include it to make 100. Since our number is less than 128, we do not need it so bit 7 is zero. We go to the next largest basis element, 64 and see if we need it. We do need 64 to generate 100, so bit 6 is one and we subtract 64 from 100 to get 36. We go to the next basis element, 32 and see if we need it. Again, we do need 32 to generate 36, so bit 5 is one and we perform 36 minus 32 to get 4. Continuing along, we need basis element 4 but not 16, 8, 2 or 1, so bits 43210 are 00100 respectively. Putting it together we get 01100100_2 (which means 64 + 32 + 4).

This operation can be visualized using the table below.

Number	Basis	Need it?	Bit	Operation
100	128	no	bit7 = 0	none
100	64	yes	bit6 = 1	100 – 64 = 36
36	32	yes	bit5 = 1	36 – 32 = 4
4	16	no	bit4 = 0	none
4	8	no	bit3 = 0	none
4	4	yes	bit2 = 1	4 – 4 = 0
0	2	no	bit1 = 0	none
0	1	no	bit0 = 0	none

Table 2.11 – Example conversion from decimal to unsigned 8-bit binary

$$\boxed{\textbf{If the least significant bit is zero, then the number is even.}} \tag{2.4}$$

$$\boxed{\textbf{If the right-most } n \textbf{ bits (least significant) are zero, then the number is divisible by } 2^n.} \tag{2.5}$$

We define an unsigned 8-bit number using the `unsigned char` format. When a number is stored into an `unsigned char` it is converted to an 8-bit unsigned value. For example

Defining an unsigned byte in C

```
unsigned char data; // 0 to 255
unsigned char function(unsigned char input)
{
  data = input + 1;
  return data;
}
```

2.3.3 8-bit signed numbers

If a byte is used to represent a *signed 2's complement* number, then the value of the number is

$$N = -128 \cdot b7 + 64 \cdot b6 + 32 \cdot b5 + 16 \cdot b4 + 8 \cdot b3 + 4 \cdot b2 + 2 \cdot b1 + b0$$

The value of a signed byte

There are also 256 different signed 8 bit numbers. The smallest signed 8-bit number is -128 and the largest is 127. For example, 10000010_2 is $-128+2$ or -126. Other examples are shown in the following table.

Binary	Hex	Calculation	Decimal
00000000	0x00	0	0
01000001	0x41	64 + 1	65
00010110	0x16	16 + 4 + 2	22
10000111	0x87	-128 + 4 + 2 + 1	-121
11111111	0xff	-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1	-1

Table 2.12 – Example conversions of signed 8-bit binary numbers

For the signed 8-bit number system the basis is

{ -128, 64, 32, 16, 8, 4, 2, 1 }

The basis of a signed byte

The most significant bit in a 2's complement signed number will specify the sign. (2.6)

Notice that the same binary pattern of 11111111_2 could represent either 255 or -1. It is very important for the software developer to keep track of the number format. The computer cannot determine whether the 8-bit number is signed or unsigned. You, as the programmer, will determine whether the number is signed or unsigned by the specific assembly or C instructions you select to operate on the number. Some operations like addition, subtraction, and shift left (multiply by 2) use the same hardware (instructions) for both unsigned and signed operations. On the other hand, multiply, divide, and shift right (divide by 2) require separate hardware (instructions) for unsigned and signed operations. For example, the K64 has both unsigned `umull`, and signed `smull`, multiply instructions. So if you use the `smull` instruction, you are implementing signed arithmetic. The compiler will automatically choose the proper implementation.

It is always good programming practice to have a clear understanding of the data type for each number, variable, parameter, etc. For some operations there is a difference between the signed and unsigned numbers while for others it does not matter.

Signed and unsigned operations may be different

	Signed different from unsigned
/ %	division
*	multiplication
>	greater than
<	less than
>=	Greater than or equal to
<=	Less than or equal to
>>	right shift

	Signed same as unsigned
+	addition
-	subtraction
==	is equal to
|	logical OR
&	logical AND
^	logical XOR
<<	left shift

Table 2.13 – Operations on signed and unsigned numbers differ

Care must be taken when dealing with a mixture of numbers of different sizes and types.

Converting a decimal number to binary using the signed basis

Similar to the unsigned algorithm, we can use the basis to convert a decimal number into signed binary. We will work through the algorithm with the example of converting -100 to 8-bit binary. We start with the largest basis element (in this case -128) and decide if we need to include it to make -100. Without -128, we would be unable to add the other basis elements together to get any negative result, so we set bit 7 and subtract the basis element from our value. Our new value is -100 minus -128, which is 28. We go to the next largest basis element, 64 and see if we need it. We do not need 64 to generate 28, so bit6 is zero. We go to the next basis element, 32 and see if we need it. We do not need 32 to generate 28, so bit5 is zero. Now we need the basis element 16, so we set bit4, and subtract 16 from 28 (28 – 16 = 12). Continuing along, we need basis elements 8 and 4 but not 2 and 1, so bits 3210 are 1100. Putting it together we get 10011100_2 (which means -128+16+8+4).

This operation can be visualized using the table below.

Number	Basis	Need it?	Bit	Operation
100	-128	yes	bit7 = 1	-100 – (-128) = 28
28	64	no	bit6 = 0	none
28	32	no	bit5 = 0	none
28	16	yes	bit4 = 1	28 – 16 = 12
12	8	yes	bit3 = 1	12 – 8 = 4
4	4	yes	bit2 = 1	4 – 4 = 0
0	2	no	bit1 = 0	none
0	1	no	bit0 = 0	none

Table 2.14 – Example conversion from decimal to signed 8-bit binary

To make the negative of a 2's complement signed number we first complement (toggle) all the bits, then add 1. (2.7)

Converting a decimal number to binary using 2's complement

A second way to convert negative numbers into binary is to first convert them into unsigned binary, then do a 2's complement negate. For example, we earlier found that +100 is 01100100_2. The 2's complement negate is a two step process. First, we do a logic complement (toggle all bits) to get 10011011_2. Then, add one to the result to get 10011100_2.

Converting a decimal number to binary using modulo arithmetic

A third way to convert negative numbers into binary is to first add the number to 256, then convert the unsigned result to binary using the unsigned method. For example, to find -100, we add -100 to 256 to get 156. Then we convert 156 to binary resulting in 10011100_2. This method works because in 8-bit binary maths adding 256 to a number does not change the value.

An error will occur if you use signed operations on unsigned numbers, or use unsigned operations on signed numbers. (2.8)

To improve the clarity of software, always specify the format of data (signed versus unsigned) when defining or accessing the data.

(2.9)

We define a signed 8-bit number using the **char** format. When a number is stored into a **char** it is converted to an 8-bit signed value. For example

Defining a signed byte in C

```
char data; // -128 to 127
char function(char input)
{
  data = input + 1;
  return data;
}
```

2.3.4 16 bit unsigned numbers

A half-word or double byte contains 16 bits

A half-word is 16 bits

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

where each bit b15, ..., b0 is binary and has the value 1 or 0. If a half-word is used to represent an unsigned number, then the value of the number is

The value of an unsigned half-word

$$\begin{aligned} N = {} & 32768 \cdot b15 + 16384 \cdot b14 + 8192 \cdot b13 + 4096 \cdot b12 \\ & + 2048 \cdot b11 + 1024 \cdot b10 + 512 \cdot b9 + 256 \cdot b8 \\ & + 128 \cdot b7 + 64 \cdot b6 + 32 \cdot b5 + 16 \cdot b4 \\ & + 8 \cdot b3 + 4 \cdot b2 + 2 \cdot b1 + b0 \end{aligned}$$

There are 65,536 different unsigned 16-bit numbers. The smallest unsigned 16-bit number is 0 and the largest is 65535. For example, $0010\ 0001\ 1000\ 0100_2$ or 0x2184 is 8192 + 256 + 128 + 4 or 8580.

Other examples are shown in the following table.

Binary	Hex	Calculation	Decimal
0000 0000 0000 0000	0x0000	0	0
0000 0100 0000 0001	0x0401	1024 + 1	1025
0000 1100 1010 0000	0x0CA0	2048 + 1024 + 128 + 32	3232
1000 1110 0000 0010	0x8E02	32768 + 2048 + 1024 + 512 + 2	36354
1111 1111 1111 1111	0xFFFF	32768+16384+8192+4096+2048+1024 +512+256+128+64+32+16+8+ 4+2+1	65535

Table 2.15 – Example conversions of unsigned 16-bit binary numbers

For the unsigned 16-bit number system the basis is

{ 32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1 }

The basis of an unsigned half-word

We define an unsigned 16-bit number using the **`unsigned short`** format. When a number is stored into an `unsigned short` it is converted to a 16-bit unsigned value. For example

```
unsigned short data; // 0 to 65535
unsigned short function(unsigned short input)
{
  data = input + 1;
  return data;
}
```

Defining an unsigned word in C

2.3.5 16-bit signed numbers

If a half-word is used to represent a signed 2's complement number, then the value of the number is

$$
\begin{aligned}
N = -32768 \cdot b15 + 16384 \cdot b14 + 8192 \cdot b13 + 4096 \cdot b12 \\
+ 2048 \cdot b11 + 1024 \cdot b10 + 512 \cdot b9 + 256 \cdot b8 \\
+ 128 \cdot b7 + 64 \cdot b6 + 32 \cdot b5 + 16 \cdot b4 \\
+ 8 \cdot b3 + 4 \cdot b2 + 2 \cdot b1 + b0
\end{aligned}
$$

The value of a signed half-word

There are also 65,536 different signed 16-bit numbers. The smallest signed 16-bit number is -32768 and the largest is 32767.

For example, $1101\ 0000\ 0000\ 0100_2$ or 0xD004 is -32768 + 16384 + 4096 + 4 or -12284. Other examples are shown in the following table.

Binary	Hex	Calculation	Decimal
0000 0000 0000 0000	0x0000	0	0
0000 0100 0000 0001	0x0401	1024 + 1	1025
0000 1100 1010 0000	0x0CA0	2048 + 1024 + 128 + 32	3232
1000 1110 0000 0010	0x8E02	-32768 + 2048 + 1024 + 512 + 2	-31742
1111 1111 1111 1111	0xFFFF	-32768+16384+8192+4096+2048+1024 +512+256+128+64+32+16+8+ 4+2+1	-1

Table 2.16 – Example conversions of signed 16-bit binary numbers

For the signed 16-bit number system the basis is

The basis of a signed half-word

{ -32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1 }

We define a signed 16-bit number using the **`short`** format. When a number is stored into a **`short`** it is converted to a 16-bit signed value. For example

Defining a signed half-word in C

```
short data; // -32768 to 32767
short function(short input)
{
  data = input + 1;
  return data;
}
```

2.3.6 Typedefs for Signed and Unsigned Data Types

To avoid confusion and make the signed and unsigned data types easy to recognise, `stdint.h` makes the following type definitions:

`typedef`s for signed and unsigned data

```
// Signed types
typedef char int8_t;
typedef int  int16_t;
typedef long int32_t;

// Unsigned types
typedef unsigned char uint8_t;
typedef unsigned int  uint16_t;
typedef unsigned long uint32_t;
```

2.3.7 Big- and Little-Endian

When we store 16-bit data into memory it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Some NXP microprocessors (those that came from Freescale, formerly Motorola) implement the *big-endian* approach that stores the most significant part first. Intel microprocessors implement the *little-endian* approach that stores the least significant part first. The PowerPC is *bi-endian*, because it can be configured to efficiently handle both big- and little-endian.

Big- and little-endian defined

For example, assume we wish to store the 16-bit number `1000` (`0x03E8`) at locations `0x50`, `0x51`, then

address	contents
0x0050	0x03
0x0051	0xE8

Big-endian

address	contents
0x0050	0xE8
0x0051	0x03

Little-endian

Example of big- and little-endian storage of a 16-bit number

We also can use either the big- or little-endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable. If we wish to store the 32-bit number `0x12345678` at locations `0x50-0x53` then

address	contents
0x0050	0x12
0x0051	0x34
0x0052	0x56
0x0053	0x78

Big-endian

address	contents
0x0050	0x78
0x0051	0x56
0x0052	0x34
0x0053	0x12

Little-endian

Example of big- and little-endian storage of a 32-bit number

In the above two examples we normally would not pick out individual bytes (e.g., the `0x12`), but rather capture the entire multiple byte data as one non-divisible piece of information. On the other hand, if each byte in a multiple byte data structure is individually addressable, then both the big- and little-endian schemes store the data in first to last sequence.

For example, if we wish to store the 4 ASCII characters `'6812'` which is `0x36383132` at locations `0x50-0x53`, then the ASCII `'6'` `=` `0x36` comes first in both big- and little-endian schemes.

Example of big- and little-endian storage of a multiple byte structure

address	contents
0x0050	0x36
0x0051	0x38
0x0052	0x31
0x0053	0x32

Big- and Little-endian

The term "Big-Endian" comes from Jonathan Swift's satiric novel *Gulliver's Travels*. In Swift's book, a Big-Endian refers to a person who cracks their egg on the big end. The Lilliputians considered the big-endians as inferiors. The big-endians fought a long and senseless war with the Lilliputians who insisted it was only proper to break an egg on the little end.

An error will occur when data is stored in Big-Endian format by one computer and read in Little-Endian format on another. (2.10)

2.3.8 Boolean information

A Boolean number has two states. The two values could represent the logical values of true or false. The positive logic representation defines true as a 1 or high, and false as a 0 or low. If you were controlling a motor, light, heater or air conditioner then Boolean could mean on or off. In communication systems, we represent the information as a sequence of Booleans, mark or space. For black or white graphic displays we use Booleans to specify the state of each pixel. The most efficient storage of Booleans on a computer is to map each Boolean into one memory bit. In this way, we could pack 8 Booleans into each byte. If we have just one Boolean to store in memory, out of convenience we allocate an entire byte or word for it. Most C compilers including GCC define:

A Boolean type has logical values of true or false

False to be **all zeros**

True to be any **nonzero value**. (2.11)

The Boolean type as implemented in C

Many programmers add the following macros to their code:

```
#define FALSE 0
#define TRUE  1
```

or the following enumeration, which is just a cleaner alternative to a series of `#define` statements:

```
typedef enum {FALSE, TRUE} BOOL;
```

2.3.9 Decimal Numbers

Decimal numbers are written as a sequence of decimal digits (0 through 9). The number may be preceded by a plus or minus sign or followed by an `L` or `U`. Lower case `l` or `u` could also be used. The minus sign gives the number a negative value, otherwise it is positive. The plus sign is optional for positive values. Unsigned integer literals should be followed by `U`. You can place an `L` at the end of the number to signify it to be a 32-bit signed number.

Decimal numbers have notation to specify the type

The range of a decimal number depends on the data type as shown in the following table.

Type	Range	Precision	Examples
`unsigned char`	0 to 255	8 bits	0, 10, 123
`char`	-128 to 127	8 bits	-123, 0, 10, +10
`unsigned short`	0 to 65535	16 bits	0, 2000, 2000U, 50000U
`short`	-32768 to 32767	16 bits	-1000, 0, 1000, +20000
`unsigned long`	0 to 4294967295	32 bits	0, 2000, 1234567U
`long`	-2147483648 to 2147483647	32 bits	-1234567L, 0L, 1234567L

Table 2.17 – The range of decimal numbers

In C, the `int` data type should be avoided because it is platform dependent

Because the K64 microcontroller architecture is based on 32-bit data and addresses (and not 16-bit), the `unsigned int` and `int` data types are 32 bits. On the other hand, on 16-bit microcontrollers, such as the NXP MC9S12, the `unsigned int` and `int` data types are 16 bits. In order to make your software more compatible with other machines, it is preferable to use the short type when needing 16 bit data and the long type for 32 bit data.

Type	NXP MC9S12	NXP Cortex-M4
`unsigned char`	8 bits	8 bits
`char`	8 bits	8 bits
`unsigned short`	16 bits	16 bits
`short`	16 bits	16 bits
`unsigned int`	16 bits	32 bits
`int`	16 bits	32 bits
`unsigned long`	32 bits	32 bits
`long`	32 bits	32 bits

Table 2.18 – Differences between a MC9S12 and a Cortex-M4

Minimize the use of 32-bit numbers on a 16-bit computer

Since the MC9S12 microcomputers do not have direct support of 32-bit numbers, the use of long data types on these devices should be minimized. On the other hand, a careful observation of the code generated yields the fact that the compilers are more efficient with 16-bit numbers than with 8-bit numbers.

Decimal numbers are reduced to their two's complement or unsigned binary equivalent and stored as 8/16/32-bit binary values.

The manner in which decimal literals are treated depends on the context. For example

```
short I;
unsigned short J;
char K;
unsigned char L;
long M;
void main(void)
{
  I = 97;    // 16 bits 0x0061
  J = 97;    // 16 bits 0x0061
  K = 97;    // 8 bits 0x61
  L = 97;    // 8 bits 0x61
  M = 97;    // 32 bits 0x00000061
}
```

2.3.10 Octal Numbers

Octal numbers begin with a leading 0

If a sequence of digits begins with a leading `0` (zero) it is interpreted as an octal value. There are only eight octal digits, 0 through 7. As with decimal numbers, octal numbers are converted to their binary equivalent in 8-bit, 16-bit or 32-bit words. The range of an octal number depends on the data type as shown in the following table.

Type	Range	Precision	Examples
`unsigned char`	0 to 0377	8 bits	0, 010, 0123
`char`	-0200 to 0177	8 bits	-0123, 0, 010, +010
`unsigned short`	0 to 0177777	16 bits	0, 02000, 0150000U
`short`	-0100000 to 077777	16 bits	-01000, 0, 01000, +020000
`unsigned long`	0 to 037777777777	32 bits	0, 02000, 015000000U
`long`	-020000000000 to 017777777777	32 bits	-01234567L, 0L, 01234567L

Table 2.19 – The range of octal numbers

Each octal digit maps to 3 bits

Notice that the octal values 0 through 07 are equivalent to the decimal values 0 through 7. One of the advantages of this format is that it is very easy to convert back and forth between octal and binary. Each octal digit maps directly to/from 3 binary digits.

2.3.11 Hexadecimal Numbers

The hexadecimal number system uses base 16 as opposed to our regular decimal number system that uses base 10. Like the octal format, the hexadecimal format is also a convenient mechanism for humans to represent binary information, because it is extremely simple to convert back and forth between binary and hexadecimal. A *nibble* is defined as 4 bits. Each value of the 4-bit nibble is mapped into a unique hex digit.

Hexadecimal numbers are base 16, and code into a 4-bit nibble

Hex digit	**Decimal Value**	**Binary Value**
0	`0`	`0000`
1	`1`	`0001`
2	`2`	`0010`
3	`3`	`0011`
4	`4`	`0100`
5	`5`	`0101`
6	`6`	`0110`
7	`7`	`0111`
8	`8`	`1000`
9	`9`	`1001`
a or A	`10`	`1010`
b or B	`11`	`1011`
c or C	`12`	`1100`
d or D	`13`	`1101`
e or E	`14`	`1110`
f or F	`15`	`1111`

Table 2.20 – Definition of hexadecimal representation

Computer programming environments use a wide variety of symbolic notations to specify the numbers in various bases. The following table illustrates various formats for numbers.

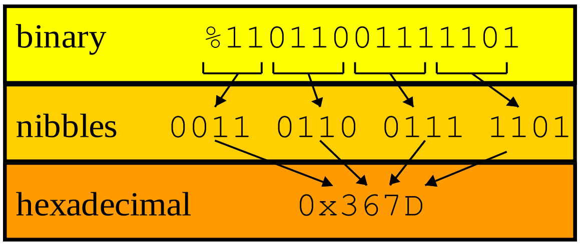
Environment	Binary format	Hexadecimal format	Decimal format
NXP assembly language	%01111010	$7A	122
Intel and TI assembly language	01111010b	7Ah	122
C language	-	0x7A	122

Table 2.21 – Various hexadecimal formats

To convert from binary to hexadecimal we can:

1) divide the binary number into right justified nibbles;
2) convert each nibble into its corresponding hexadecimal digit.

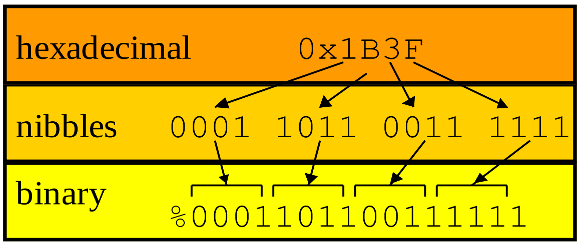
Converting from binary to hexadecimal



To convert from hexadecimal to binary we can:

1) convert each hexadecimal digit into its corresponding 4-bit binary nibble;
2) combine the nibbles into a single binary number.

Converting from hexadecimal to binary



In C hexadecimal values begin with `0x`

If a sequence of digits begins with `0x` or `0X` then it is taken as a hexadecimal value. In this case the word digits refers to hexadecimal digits (0 through F). As with decimal numbers, hexadecimal numbers are converted to their binary equivalent in 8-bit bytes, 16-bit half-words or 32-bit words.

The range of a hexadecimal number depends on the data type as shown in the following table.

Type	Range	Precision	Examples
`unsigned char`	0x00 to 0xFF	8 bits	0x01, 0x3a, 0xb3
`char`	-0x80 to 0x7F	8 bits	-0x01, 0x3a, -0x7b
`unsigned short`	0x0000 to 0xFFFF	16 bits	0x22, 0xABCD, 0xF0A6
`short`	-0x8000 to 0x7FFF	16 bits	-0x1234, 0x0, 0x7ABC
`unsigned long`	0x00000000 to 0xFFFFFFFF	32 bits	0x00000001, 0xABCDEF
`long`	-0x80000000 to 0x7FFFFFFF	32 bits	-0x1234567, 0xABCDEF

Table 2.22 – The range of hexadecimal numbers

2.3.12 Character Literals

Character literals consist of one or two characters surrounded by apostrophes. The manner in which character literals are treated depends on the context. For example:

```
short I;
unsigned short J;
char K;
unsigned char L;
long M;
void main(void)
{
  I = 'a';     // 16 bits 0x0061
  J = 'a';     // 16 bits 0x0061
  K = 'a';     // 8 bits 0x61
  L = 'a';     // 8 bits 0x61
  M = 'a';     // 32 bits 0x00000061
}
```

Character literals are surrounded by apostrophes

All standard ASCII characters are positive because the high-order bit is zero. In most cases it doesn't matter if we declare character variables as signed or unsigned. On the other hand, we have seen earlier that the compiler treats signed and unsigned numbers differently. Unless a character variable is specifically declared to be unsigned, its high-order bit will be taken as a sign bit. Therefore, we should not expect a character variable, which is not declared unsigned, to compare equal to the same character literal if the high-order bit is set.

The C `char` type is signed and is therefore different to unsigned ASCII literals if the sign bit is set

2.3.13 String Literals

Strings are really character arrays

Strictly speaking, C does not recognize character strings, but it does recognize arrays of characters and provides a way to write character arrays, which we call *strings*. Surrounding a character sequence with quotation marks, e.g., `"John"`, sets up an array of characters and generates the address of the array. In other words, at the point in a program where it appears, a string literal produces the address of the specified array of character literals. The array itself is located elsewhere. MX will place the strings into the text area, i.e., the string literals are considered constant and will be defined in the Flash memory of an embedded system. This is very important to remember. Notice that this differs from a character literal which generates the value of the literal directly. Just to be sure that this distinct feature of the C language is not overlooked, consider the following example:

Strings are surrounded by quotes

```
char *pt;
void main(void)
{
  pt = "John"; // pointer to the string
  printf(pt);  // passes the pointer, not the data itself
}
```

The compiler places the string in memory and uses a pointer to it when calling `printf`. MX pushes the parameter on the stack.

Strings in Flash memory are referenced by a pointer in RAM

Notice that the pointer, `pt`, is allocated in RAM (`.bss`) and the string is stored in Flash memory (`.text`). The assignment statement `pt = "John";` copies the address, not the data. Similarly, the function `printf()` must receive the address of a string as its first (in this case, only) argument. First, the address of the string is assigned to the character pointer `pt`. Unlike other languages, the string itself is not assigned to `pt`, only its address is. After all, `pt` is a 32-bit object and, therefore, cannot hold the string itself. The same program could be written better as

```
void main(void)
{
  printf("John"); // passes the pointer, not the data itself
}
```

Notice again that the program passes a pointer to the string into `printf()`, and not the string itself.

In this case, it is tempting to think that the string itself is being passed to `printf()`; but, as before, only its address is.

Since strings may contain as few as one or two characters, they provide an alternative way of writing character literals in situations where the address, rather than the character itself, is needed.

It is a convention in C to identify the end of a character string with a null (zero) character. Therefore, C compilers automatically suffix character strings with such a terminator. Thus, the string `"John"` sets up an array of five characters (`'J'`, `'o'`, `'h'`, `'n'`, and zero) and generates the address of the first character, for use by the program.

In C, strings are null terminated (end with a 0x00 byte)

Remember that `'J'` is different from `"A"`. Consider the following example:

```
char letter, *pt;
void main(void)
{
  pt = "A";       // pointer to the string
  letter = 'A';  // the data itself ('A' ASCII 65=$41)
}
```

2.3.14 Escape Sequences

Sometimes it is desirable to code nongraphic characters in a character or string literal. This can be done by using an *escape sequence* – a sequence of two or more characters in which the first (escape) character changes the meaning of the following character(s). When this is done the entire sequence generates only one character. C uses the backslash (\) for the escape character.

Escape sequences access the special ASCII codes and begin with a \

The following escape sequences are recognized by the GCC compiler:

Sequence	Name	Value
`\n`	newline, linefeed	0x0A = 10
`\t`	tab	0x09 = 9
`\b`	backspace	0x08 = 8
`\f`	form feed	0x0C = 12
`\a`	bell	0x07 = 7
`\r`	return	0x0D = 13
`\v`	vertical tab	0x0B = 11
`\0`	null	0x00 = 0
`\"`	ASCII double quote	0x22 = 34
`\\`	ASCII back slash	0x5C = 92
`\'`	ASCII single quote	0x27 = 39

Table 2.23 – The escape sequences supported by MX

Other nonprinting characters can also be defined using the `\ooo` octal format. The digits `ooo` can define any 8-bit octal number. The following three lines are equivalent:

```
printf("\tJohn\n");
printf("\11John\12");
printf("\011John\012");
```

A newline character is represented by `\n`

The term *newline* refers to a single character which, when written to an output device, starts a new line. Some hardware devices use the ASCII carriage return (13) as the newline character while others use the ASCII line feed (10). It really doesn't matter which is the case as long as we write `\n` in our programs. Avoid using the ASCII value directly since that could produce compatibility problems between different compilers.

Backslash is also used to specify literal characters

There is one other type of escape sequence: anything undefined. If the backslash is followed by any character other than the ones described above, then the backslash is ignored and the following character is taken literally. So the way to code the backslash is by writing a pair of backslashes and the way to code an apostrophe or a quote is by writing `\'` or `\"` respectively.

2.4 Variables and Constants

The purpose of this section is to explain how to create and access variables and constants. The storage and retrieval of information are critical operations of any computer system. This section will also present the C syntax and resulting assembly code generated by the MX compiler.

A *variable* is a named object that resides in RAM memory and is capable of being examined and modified. A variable is used to hold information critical to the operation of the embedded system. A *constant* is a named object that resides in memory (usually in Flash memory) and is only capable of being examined. As we saw in the last section, a *literal* is the direct specification of a number, character or string. The difference between a literal and a constant is that constants are given names so that they are easier to remember and can be accessed more than once.

The difference between a *variable*, a *constant* and a *literal*

For example:

```
short MyVariable;             // variable allows read/write access
const short MY_CONSTANT = 50; // constant allows only read access
#define FIFTY 50
void main(void)
{
  MyVariable = 50;        // write access to the variable
  OutSDec(MyVariable);    // read access to the variable
  OutSDec(MY_CONSTANT);   // read access to the constant
  OutSDec(50);            // "50" is a literal
  OutSDec(FIFTY);         // FIFTY is also a literal
}
```

Listing 2.26 – Example showing a variable, a constant and some literals

The concepts of precision and type (unsigned vs. signed) developed for numbers in the last section apply to variables and constants as well. In this section we will begin the discussion of variables that contain integers and characters. Even though pointers are similar in many ways to 32-bit unsigned integers, pointers will be treated in a later section. Although arrays and structures also fit the definition of a variable, they are regarded as collections of variables and will be discussed in later sections.

The distinction between global and local variables

The term *storage class* refers to the method by which an object is assigned space in memory. The MX compiler recognizes three storage classes – static, automatic, and external. In this document we will use the term *global variable* to mean a regular static variable that can be accessed by all other functions. Similarly, we will use the term *local variable* to mean an automatic variable that can be accessed only by the function that created it. As we will see in the following sections there are other possibilities like a static global and static local.

2.4.1 Statics

Static variables are given space in memory at some fixed location within the program. They exist when the program starts to execute and continue to exist throughout the program's entire lifetime. The value of a static variable is faithfully maintained until we change it deliberately (or remove power from the memory). A constant, which we define by adding the modifier `const`, can be read but not changed.

In an embedded system we normally wish to place all variables in RAM and all constants in Flash memory.

In the MX IDE, we set the starting memory address for the static variables in the linker parameter `*.ld` file by specifying the `m_data` *user segment*. The `m_data` segment is just the entire RAM of the microcontroller and is used to store data. The program instructions will be placed in the `m_text` user segment, which is normally a page of Flash memory reserved for instructions. The constants will also be placed in the `m_text` user segment in an area of Flash memory reserved for constants and string literals.

"bss" stands for "Block Started by Symbol", and is a leftover acronym from an early assembler written for an IBM mainframe computer in the 1950's. It is the name of the data section containing uninitialized variables

The MX compiler places static variables in the `.bss` section, which we can view in the linker output `*.map` file in the "SECTIONS" section. It also places the program in the `.text` section, and constants in the `.rodata` (read only data) section. The MX linker automatically places sections into their correct segments.

The MX compiler uses the name of each static variable to define an assembler label. The following example sets a global, called `TheGlobal`, to the value `1000`. This global can be referenced by any function from any file in the software system. It is truly global.

```
int TheGlobal;   // a regular global variable
void main(void)
{
  TheGlobal = 1000;
}
```

Listing 2.27 – Example showing a regular global variable

The K64 code generated by the MX compiler is as follows

```
main:
     ldr   r3, TheGlobal
     mov   r2, #1000
     str   r2, [r3]
     bx    lr
```

Listing 2.28 – Example showing a global variable in assembly language

The fact that these types of variables exist in permanently reserved memory means that static variables exist for the entire life of the program. When the power is first applied to an embedded computer, the values in its RAM are usually undefined. Therefore, initializing global variables requires special run-time software. The MX compiler will attach the C code in the `startup.c` file to the beginning of every program. This software is executed first, before our `main()` program is started. We can see by observing the `startup.c` file that the MX compiler will clear all static variables to zero (`ZeroOut`) immediately after a hardware reset, and then copy all the values of initialized static variables from Flash to RAM (`CopyDown`).

A *static global* is very similar to a regular global. In both cases, the variable is defined in RAM permanently. The assembly language access is identical. The only difference is the scope. The static global can only be accessed within the file where it is defined.

A static global can only be accessed within the file where it is defined

The following example also sets a global, called `TheGlobal`, to the value `1000`.

```
static int TheGlobal;   // a static global variable
void main(void)
{
  TheGlobal = 1000;
}
```

Listing 2.29 – Example showing a static global variable

This static global cannot be referenced outside the scope of this file.

The K64 code generated by the MX compiler is the same as a regular global. MX limits access to the static global to functions defined in the same file.

```
main:
     ldr   r3, TheGlobal
     mov   r2, #1000
     str   r2, [r3]
     bx    lr
```

Listing 2.30 – Example showing a static global in assembly language

A static local retains its value from one function call to another, and can only be accessed within the function where it is defined

A *static local* is similar to a static global. Just as with the other statics, the variable is defined in RAM permanently. The assembly language code generated by the compiler that accesses the variable is identical. The only difference is the scope. The static local can only be accessed within the function where it is defined. The following example sets a static local, called `TheLocal`, to the value `1000`.

```
void main(void)
{
  static int TheLocal;   // a static local variable
  TheLocal = 1000;
}
```

Listing 2.31 – Example showing a static local variable

Again the K64 code generated by the MX compiler is the same as a regular global. MX limits access to the static local to the function in which it is defined.

```
main:
     ldr   r3, TheGlobal
     mov   r2, #1000
     str   r2, [r3]
     bx    lr
```

Listing 2.32 – Example showing a static local variable in assembly

A static local can be used to save information from one instance of the function call to the next. Assume each function wished to know how many times it has been called. Remember upon reset, the startup code will initialize all statics to zero (including static locals).

All static variables are initialized to zero by code created by the compiler

The following functions maintain such a count, and these counts cannot be accessed by other functions. Even though the names are the same, the two static locals are in fact distinct.

```
void function1(void)
{
  static int TheCount;
  TheCount++;
}

void function2(void)
{
  static int TheCount;
  TheCount++;
}
```

Listing 2.33 – Two static local variables with the same name

In each function, the address of `TheCount` will resolve to a unique address in RAM, so the K64 code generated by the MX compiler will be something like:

```
function1:
    ldr   r3, .L2
    ldr   r3, [r3]
    adds  r2, r3, #1
    ldr   r3, .L2
    str   r2, [r3]
    bx    lr
.L2:
    .word TheCount.3933
function2:
    ldr   r3, .L5
    ldr   r3, [r3]
    adds  r2, r3, #1
    ldr   r3, .L5
    str   r2, [r3]
    bx    lr
.L5:
    .word TheCount.3937
```

Listing 2.34 – Two static local variables with the same name in assembly

The MX compiler limits the scope of the local variables to within their functions only.

2.4.2 Volatile

A **`volatile`** modifier is used to indicate that a variable can change due to external influences

We add the *volatile* modifier to a variable that can change value outside the scope of the function. Usually the value of a global variable changes only as a result of explicit statements in the C function that is currently executing. This paradigm results when a single program executes from start to finish, and everything that happens is an explicit result of actions taken by the program. There are two situations that break this simple paradigm in which the value of a memory location might change outside the scope of a particular function currently executing:

1) interrupts and

2) input/output ports.

An interrupt is a hardware-requested software action. Consider the following multithreaded interrupt example. There is a foreground thread called **`main()`**, which we set up as the usual main program that all C programs have. Then, there is a background thread called **`TickHandler()`**, which we setup to be executed on a periodic basis (e.g., every 10 ms). Both threads access the global variable `Time`. The interrupt thread increments the global variable, and the foreground thread waits for time to reach `100`. Notice that `Time` changes value outside the influence of the **`main()`** program.

```c
volatile char Time;
void __attribute__ ((interrupt)) TickHandler(void)
{
  // every 10 ms
  Time++;
}

void main(void)
{
  // Disable SysTick
  SYST_CSR = 0;

  // Set reload value for a 10 ms period
  SYST_RVR = CPU_CORE_CLK_HZ / 100 - 1;

  // Clear current value as well as count flag
  SYST_CVR = 0;

  // Enable SysTick
  SYST_CSR = 0x00000007;

  // Initialise time
  Time = 0;

  // Wait for 100 counts of the 10 ms timer
  while (Time < 100);
}
```

Listing 2.35 – Code showing shared access to a common global variable

Without the `volatile` modifier the compiler might look at the two statements:

```c
Time = 0;
while (Time < 100);
```

and conclude that since the `while` loop does not modify `Time`, it could never reach `100`. Some compilers might attempt to move the read `Time` operation, performing it once before the `while` loop is executed. The `volatile` modifier disables the optimization, forcing the program to fetch a new value from the variable each time the variable is accessed.

In the next K64 example, assume `GPIOA_PDIR` is an input port containing the current status of some important external signals. The program wishes to collect status versus time data of these external signals.

```
#define GPIOA_PDIR *(uint32_t volatile *)(0x400FF010)
#define GPIOA_PDDR *(uint32_t volatile *)(0x400FF014)

unsigned char Data[100];

void main(void)
{
  int i;

  // Make Port A an input
  GPIOA_PDDR = 0x0000;
  // Collect 100 measurements
  for (i = 0; i < 100; i++)
  {
    // Collect ith measurement
    Data[i] = GPIOA_PDIR;
  }
}
```

Listing 2.36 – Code showing shared access to a common global variable

Without the **`volatile`** modifier in the `GPIOA_PDIR` definition, the compiler might optimize the **`for`** loop, reading `GPIOA_PDIR` once, then storing 100 identical copies into the data array.

2.4.3 Automatics

Automatic variables do not have fixed memory locations. They are dynamically allocated when the block in which they are defined is entered, and they are discarded upon leaving that block. Specifically, they are allocated on the K64 stack by subtracting a value (one for characters, two for short integers and four for long integers) from the stack pointer register (`SP`). Since automatic objects exist only within blocks, they can only be declared locally. An automatic variable can only be referenced (read or written to) by the function that created it. In this way, the information is protected or local to the function.

Automatic variables are created on the stack, and only exist locally

When a local variable is created it has no dependable initial value. It must be set to an initial value by means of an assignment operation. C provides for automatic variables to be initialized in their declarations, like globals. It does this by generating "hidden" code that assigns values automatically after variables are allocated space.

It is tempting to forget that automatic variables go away when the block in which they are defined exits. This sometimes leads new C programmers to fall into the "dangling reference" trap in which a function returns a pointer to a local variable, as illustrated by

```
int* BadFunction(void)
{
  int z;
  z = 1000;
  return(&z);
}
```

Listing 2.37 – Example showing an illegal reference to a local variable

When callers use the returned address of `z` they will find themselves messing around with the stack space that `z` used to occupy. This type of error is NOT flagged as a syntax error, but rather will cause unexpected behaviour during execution.

2.4.4 Implementation of Automatic Variables

Automatic variables are defined with respect to a local stack frame

If locals are dynamically allocated at unspecified memory (stack) locations, then how does the program find them? This is done by using the stack pointer (`SP`) to designate a stack frame for the currently active function. The MX compiler generates code that references variables with respect to this stack frame. When the C function is entered, space is allocated by decrementing the stack pointer (the stack grows downwards in memory). This new value of `SP` then becomes the base for references to local variables that are declared within the function. The K64 `SP` register points to the top data byte that has already been pushed – it is a "last-used" stack as opposed to a "next-available" stack.

In order to understand both the machine architecture and the C compiler, we can look at the assembly code generated. For the MX compiler, the linker/loader allocates 3 segmented memory sections: code pointed to by the `PC` (*.text section*); globals accessed with absolute addressing (*.data section*); and locals pointed to by the stack pointer `SP`. This example shows a simple C program with three local variables. Although the function doesn't do much (and will be in general be optimised out of any object code) it will serve to illustrate how local variables are created (allocation), accessed (read and write) and destroyed (deallocated).

```
void sub(void)
{
  short y1, y2, y3;   // 3 local variables
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

Listing 2.38 – Example showing three local variables

The disassembled output of the MX compiler shown below has been highlighted to clarify its operation. In the K64 the program counter (`PC`) always points to the next instruction to be executed.

Code and K64 registers and stack showing the creation and use of automatic variables

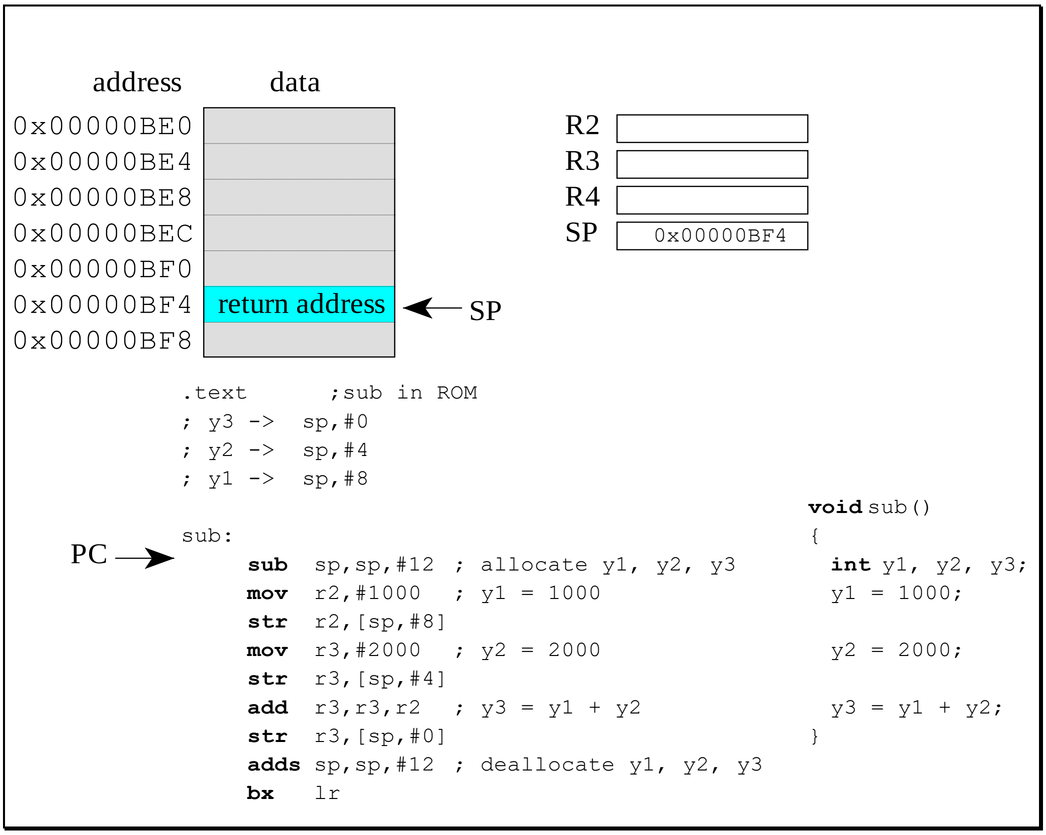


Figure 2.3 – K64 implementation of three local variables – step 1

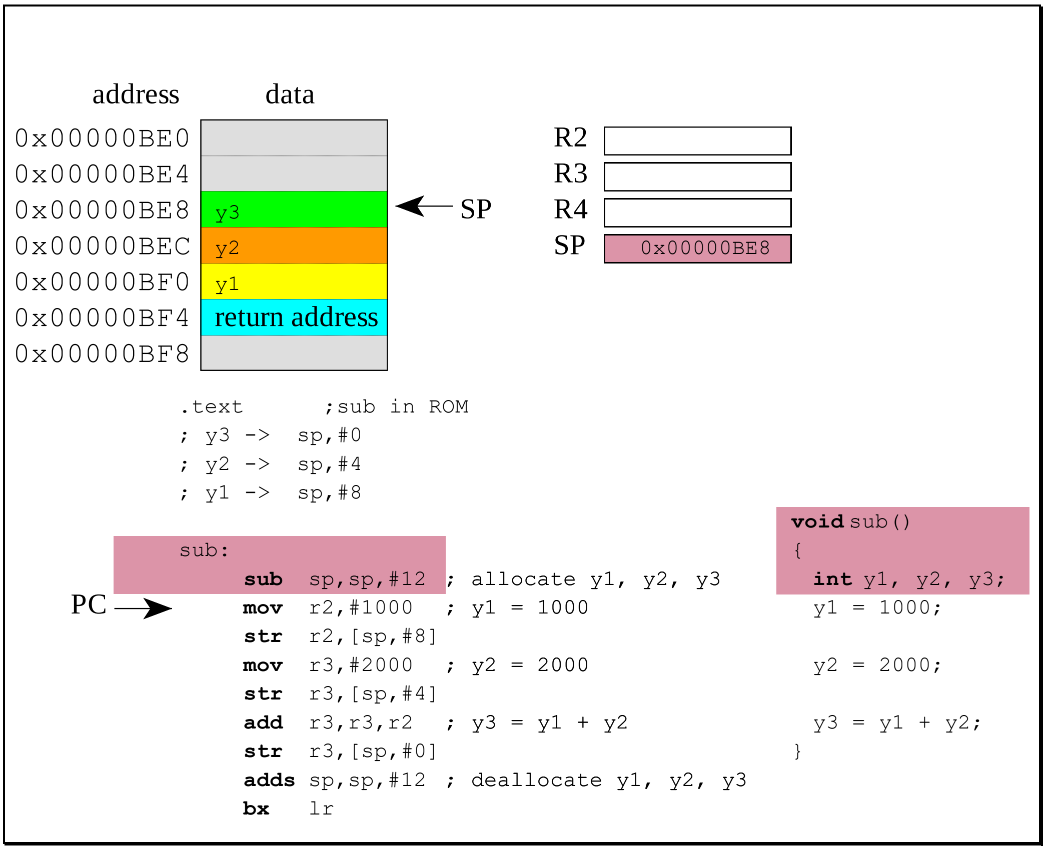


Figure 2.4 – K64 implementation of three local variables – step 2

2.80

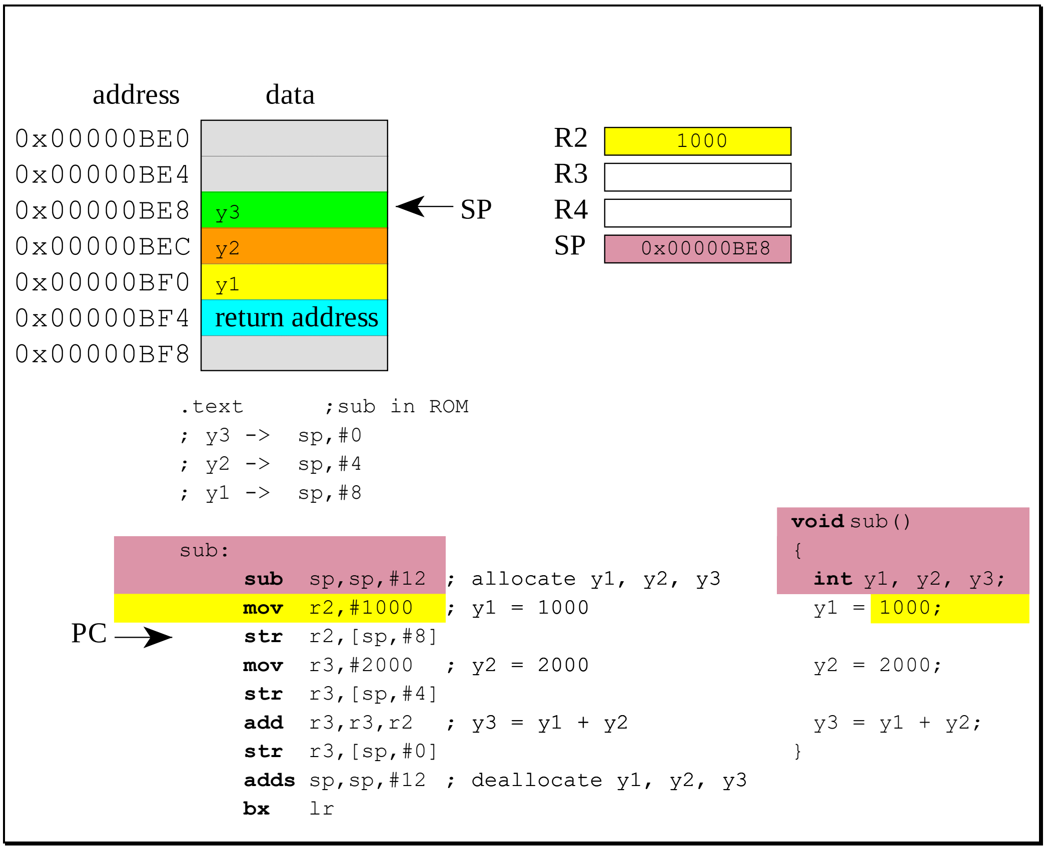


Figure 2.5 – K64 implementation of three local variables – step 3

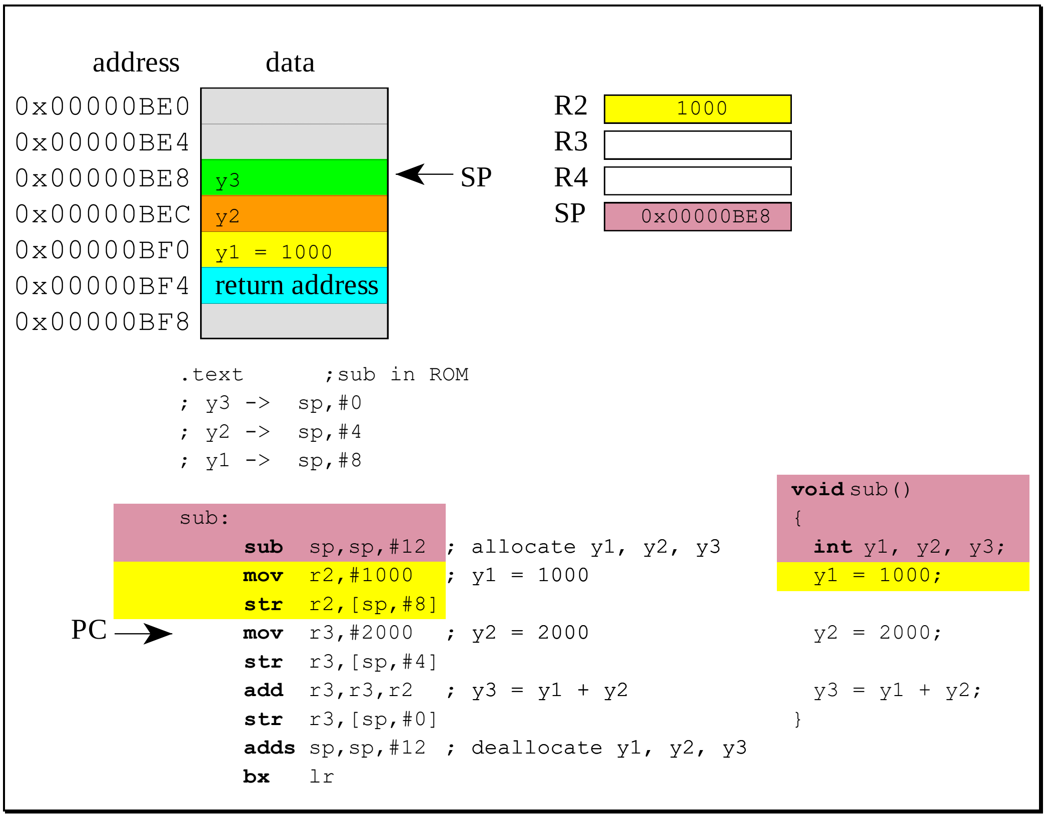


Figure 2.6 – K64 implementation of three local variables – step 4

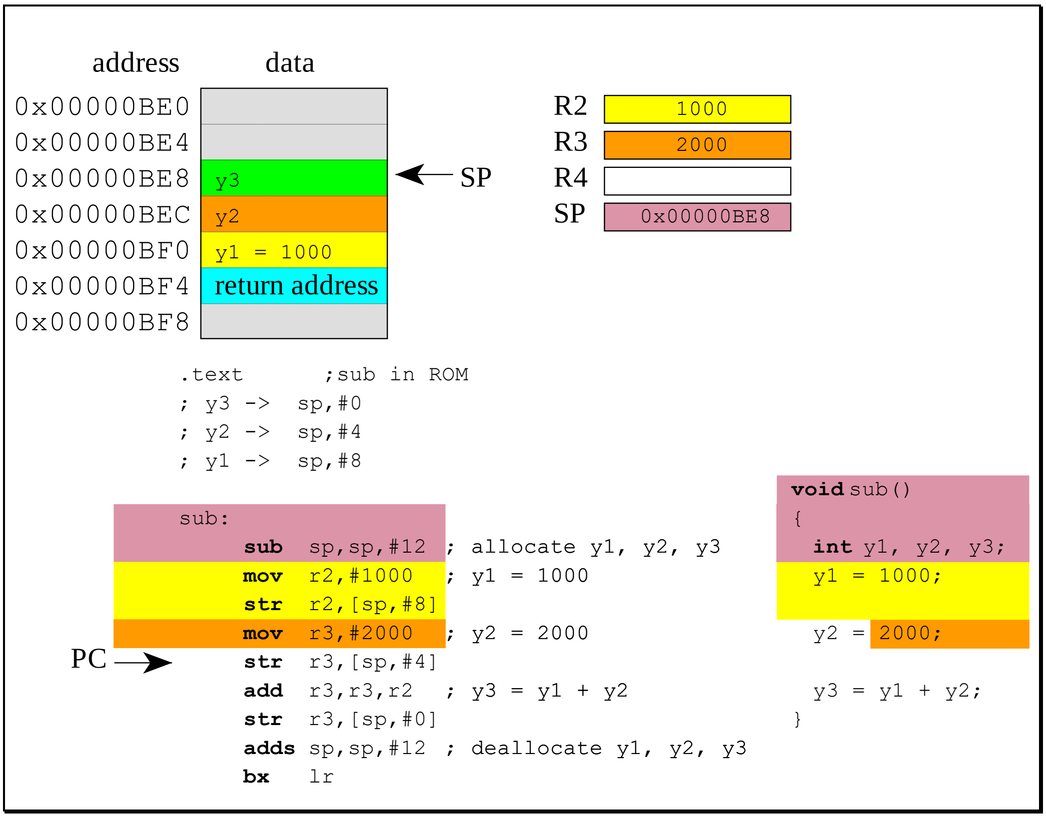


Figure 2.7 – K64 implementation of three local variables – step 5

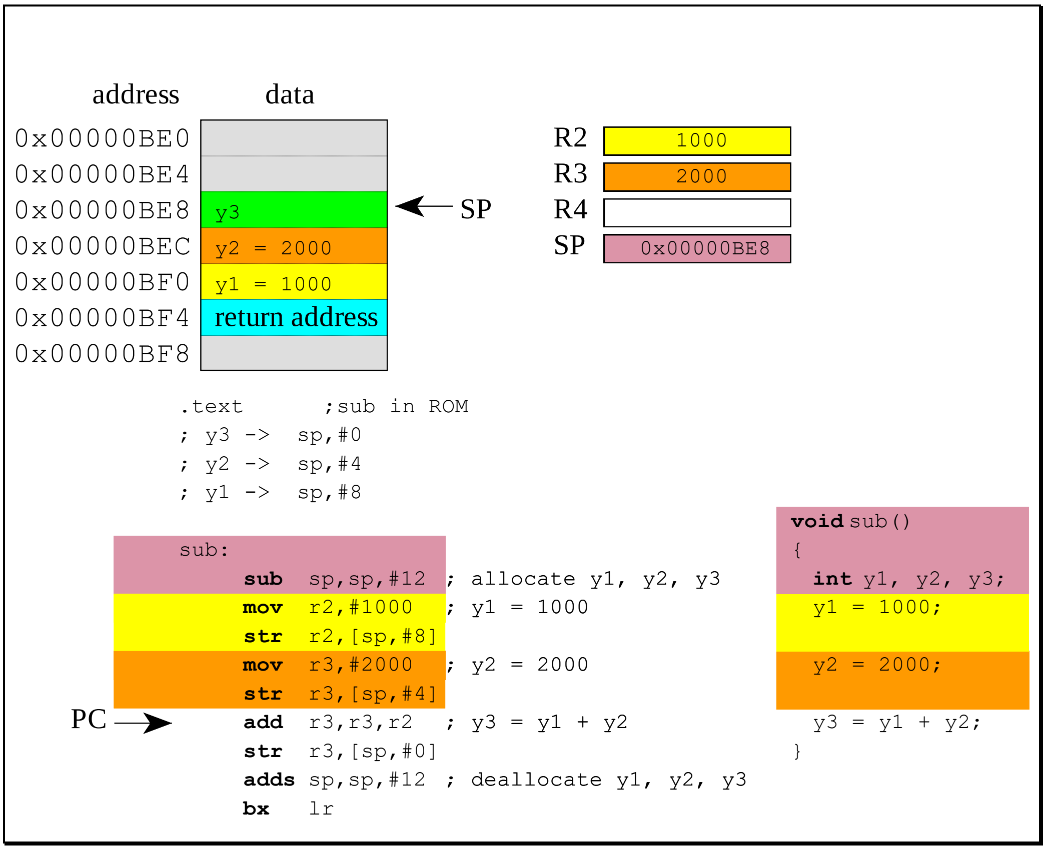


Figure 2.8 – K64 implementation of three local variables – step 6

2.82

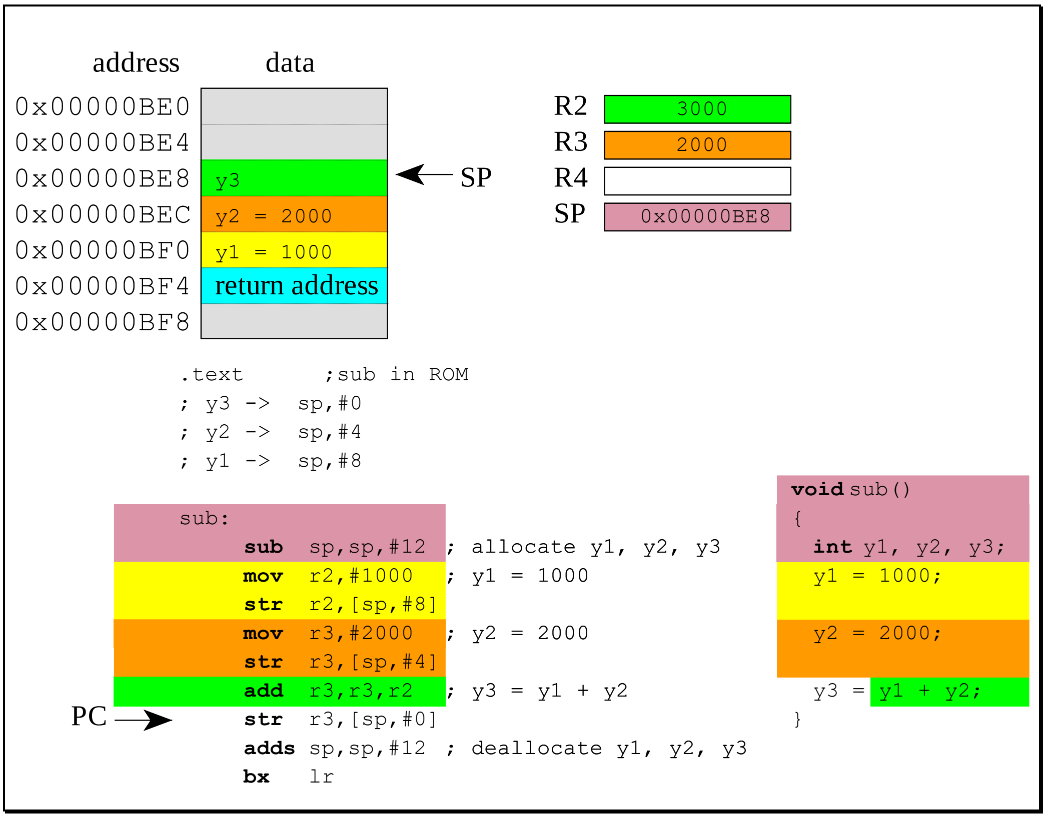


Figure 2.9 – K64 implementation of three local variables – step 7

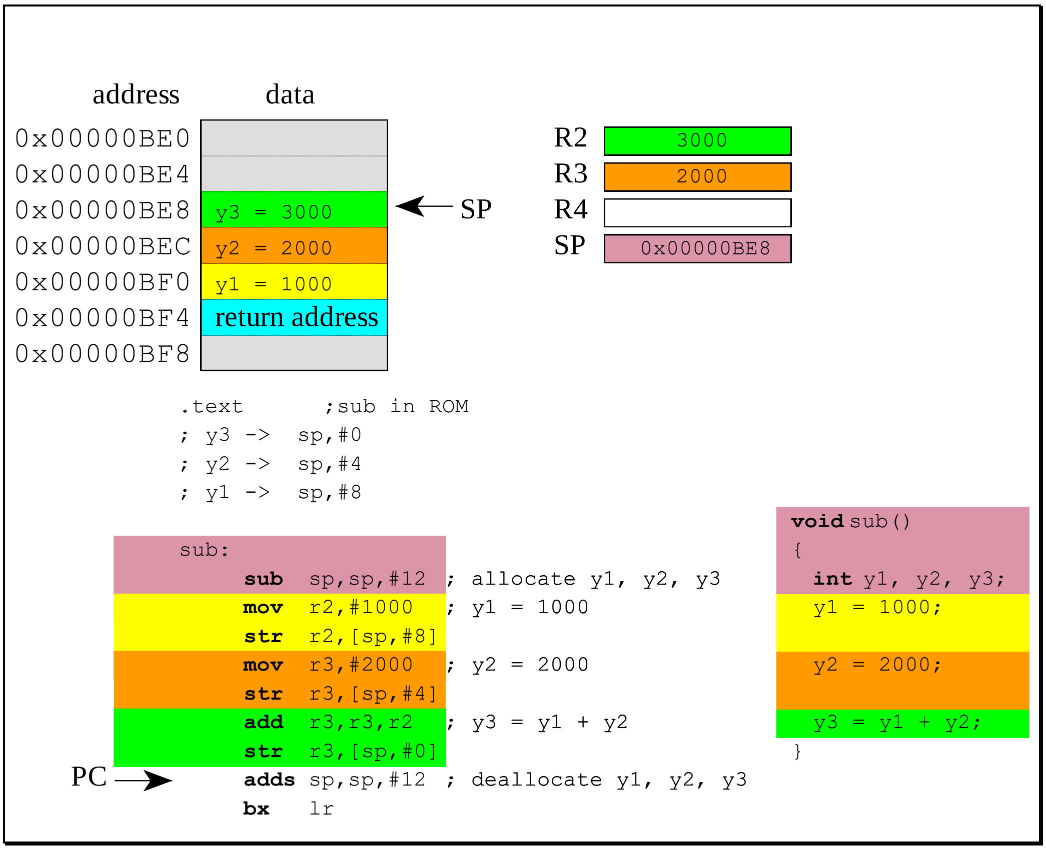


Figure 2.10 – K64 implementation of three local variables – step 8

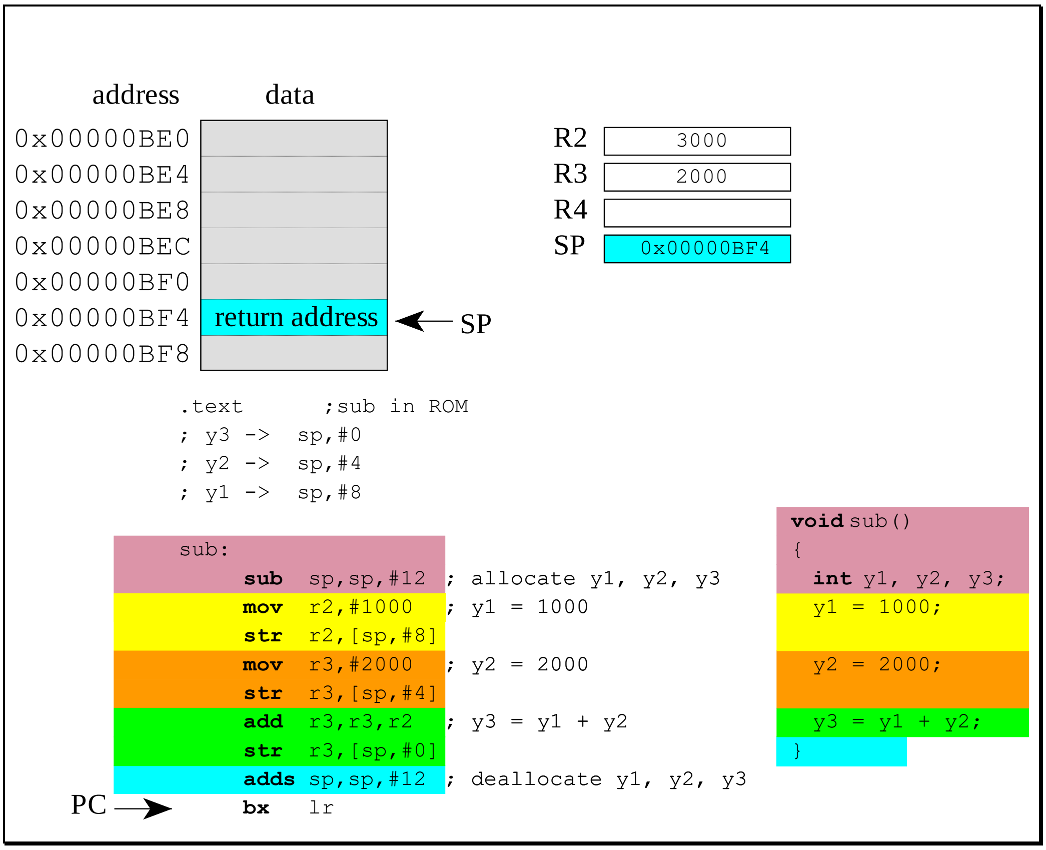
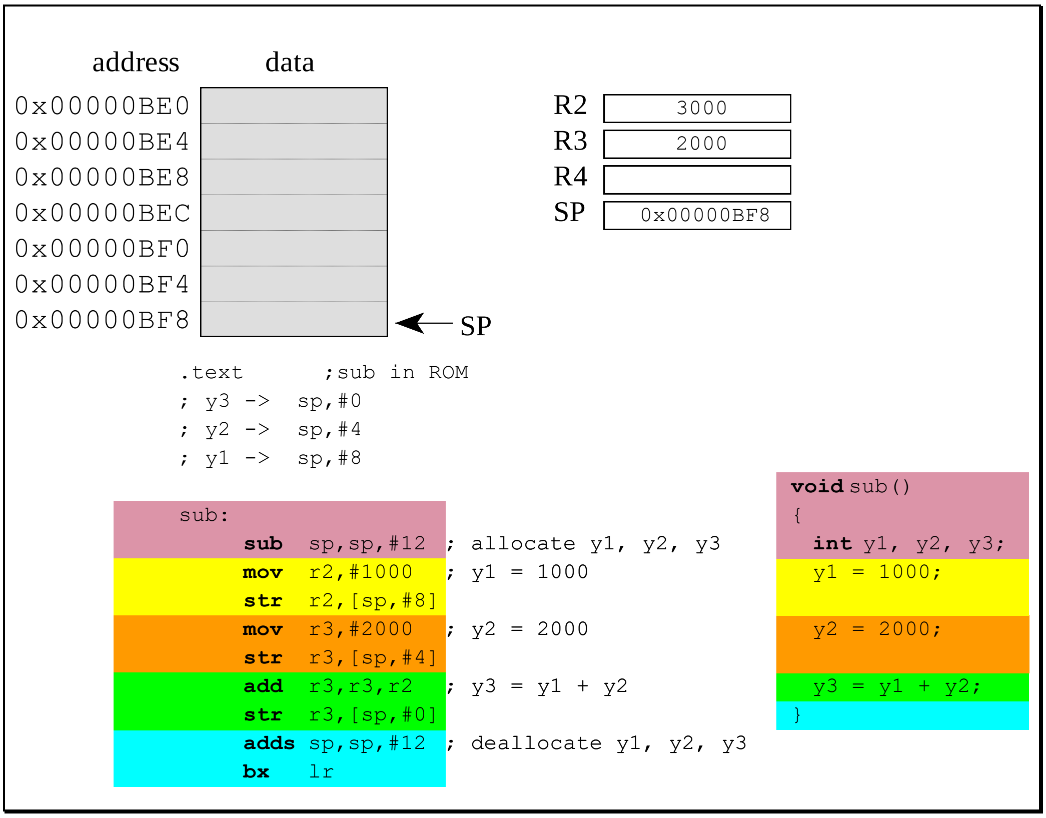


Figure 2.11 – K64 implementation of three local variables – step 9



Constant locals are defined temporarily on the stack

Figure 2.12 – K64 implementation of three local variables – step 10

The **sub** sp,sp,#12 instruction allocates the local variables, and thereafter they are accessed by indexing the stack pointer. Within the subroutine the local variables of other functions are not accessible. If a function is called from within another function, the new function will allocate its own local variable space on the stack, without disturbing the existing data.

2.4.5 Implementation of Constant Locals

A *constant local* is different to a regular local. Unlike the other locals, the constant is *not* defined temporarily on the stack. Since it cannot be changed, the assembly language code generated by the MX compiler that references the constant local replaces the reference with the actual value.

```
int TheGlobal;   // a regular global variable
void sub(void)
{
  const int THE_CONSTANT = 1000;   // a constant local
  TheGlobal = THE_CONSTANT;
}
```

Listing 2.39 – Example showing a constant local

The K64 code generated by the MX compiler is as follows (notice the reservation of space in the .bss section for the global variable)

```
    .text
sub:
    mov   r3,#1000
    ldr   r2,TheGlobal
    str   r3,[r2]
    bx    lr

    .bss
TheGlobal
```

2.4.6 Externals

Objects that are defined outside of the present source module have the *external* storage class. This means that, although the compiler knows *what* they are (signed / unsigned, 8-bit, 16-bit, 32-bit, etc.), it has no idea *where* they are. It simply refers to them by name without reserving space for them. Then, when the linker brings together the object modules, it resolves these "pending" references by finding the external objects and inserting their addresses into the instructions that refer to them. The compiler knows an external variable by the keyword `extern` that must precede its declaration.

Externals are variables defined elsewhere

Only global declarations can be designated `extern` and only globals in other modules can be referenced as external.

The following example sets an external global, called `ExtGlobal`, to the value `1000`. This global can be referenced by any function from any file in the software system. It is truly global.

```
extern int ExtGlobal;   // an external global variable
void main(void)
{
  ExtGlobal = 1000;
}
```

Listing 2.40 – Example showing an external global

The assembly language the MX compiler generates does *not* include the definition of `ExtGlobal`. The K64 code generated by the MX compiler is as follows

```
     .text
main:
     mov    r3,#1000
     ldr    r2,ExtGlobal
     str    r3,[r2]
     bx     lr
```

2.4.7 Scope

Scope refers to where a variable can be "seen"

The *scope* of a variable is the portion of the program from which it can be referenced. We might say that a variable's scope is the part of the program that "knows" or "sees" the variable. As we shall see, different rules determine the scopes of global and local objects.

Global variables have a scope that extends from the declaration to the end of the file

When a variable is declared globally (outside of a function) its scope is the part of the source file that follows the declaration – any function following the declaration can refer to it. Functions that precede the declaration cannot refer to it. Most C compilers would issue an error message in that case.

Local variables have a scope that is restricted to the function where they are defined

The scope of local variables is the block in which they are declared. Local declarations *must* be grouped together before the first executable statement in the block – at the head of the block. This is different from C++ and C11 that allow local variables to be declared anywhere in the function. It follows that the scope of a local variable effectively includes all of the block in which it is declared. Since blocks can be nested, it also follows that local variables are seen in all blocks that are contained in the one that declares the variables.

If we declare a local variable with the same name as a global object or another local in a superior block, the new variable temporarily supersedes the higher level declarations. Consider the following program.

```
unsigned char x;        // a regular global variable
void sub(void)
{
  x = 1;
  {
    unsigned char x;    // a local variable
    x = 2;
    {
      unsigned char x;  // a local variable
      x = 3;
      GPIOA_PDOR = x;
    }
    GPIOA_PDOR = x;
  }
  GPIOA_PDOR = x;
}
```

Listing 2.41 – Example showing the scope of local variables

Local variables can have nested scope

This program declares variables with the name x, assigns values to them, and outputs them to GPIOA_PDOR in such a way that, when we consider its output, the scope of its declarations becomes clear. When this program runs, it outputs 321. This only makes sense if the x declared in the inner most block masks the higher level declarations so that it receives the value '3' without destroying the higher level variables. Likewise the second x is assigned '2' which it retains throughout the execution of the inner-most block. Finally, the global x, which is assigned '1', is not affected by the execution of the two inner blocks. Notice, too, that the placement of the last two GPIOA_PDOR = x; statements demonstrates that leaving a block effectively unmasks objects that were hidden by declarations in the block. The second GPIOA_PDOR = x; sees the middle x and the last GPIOA_PDOR = x; sees the global x.

This masking of higher level declarations is an advantage, since it allows the programmer to declare local variables for temporary use without regard for other uses of the same names.

One of the mistakes a C++ programmer makes when writing C code is trying to define local variables in the middle of a block. In C local variables must be defined at the beginning of a block. The following example is proper C++ or C11 code, but results in a syntax error in C.

```
void sub(void)
{
  int x;   // a valid local variable declaration
  x = 1;
  int y;   // this declaration is improper in C
  y = 2;
}
```

Listing 2.42 – Example showing an illegal local variable declaration

We will be using the MX GCC compiler in C11 mode, so we can make declarations like this without causing errors.

2.4.8 Declarations

Every variable in C must be *declared* before it is used. Declarations force us to consider the precision (8-bit, 16-bit etc.) and format (unsigned vs. signed) of each variable.

Describing a variable involves two actions. The first action is declaring its type and the second action is defining it in memory (reserving a place for it). Although both of these may be involved, we refer to the C construct that accomplishes them as a *declaration*. As we saw previously, if the declaration is preceded by `extern` it only declares the type of the variable, without reserving space for it. In such cases, the definition must exist in another source file. Failure to do so will result in an unresolved reference error at link time.

A declaration defines the type of a variable and where it is located in memory

Table 2.24 contains examples of legitimate variable declarations. Notice that the declarations are introduced by one or two type keywords that state the data types of the variables listed. The keyword `char` declares 8-bit values, `short` declares 16-bit values, `int` declares 32-bit values, and `long` declares 32-bit values. Unless the modifier `unsigned` is present, the variables declared by these statements are assumed by the compiler to contain signed values. You could add the keyword `signed` before the data type to clarify its type.

Variables have a **`signed`** modifier by default

When more than one variable is being declared, they are written as a list with the individual names separated by commas. Each declaration is terminated with a semicolon, as are all simple C statements.

Declaration	Comment	Range
`unsigned char uc;`	8-bit unsigned number	0 to +255
`char c1, c2, c3;`	three 8-bit signed numbers	-128 to +127
`unsigned short us;`	16-bit unsigned number	0 to +65535
`short s1, s2;`	two 16-bit signed numbers	-32768 to +32767
`unsigned int ui;`	32-bit unsigned number	0 to +65535
`int i1, i2;`	two 32-bit signed numbers	-32768 to +32767
`long l1, l2, l3, l4;`	four signed 32-bit integers	-2147483648L to 2147483647L
`float f1, f2;`	two 32-bit floating-point numbers	$\approx \pm 3.402823 \times 10^{38}$
`double d1, d2;`	two 64-bit floating-point numbers	$\approx \pm 1.797693 \times 10^{308}$

Table 2.24 – Variable declarations

Variable Storage Classes and Modifiers

The following tables shows the available storage classes and modifiers for variables.

Storage class indicates where variables should be placed in memory

Storage Class	Comment
`auto`	automatic, allocated on the stack
`extern`	defined in some other program file
`static`	permanently allocated
`register`	attempt to implement an automatic using a register instead of on the stack

Table 2.25 – Variable storage classes

The MX compiler allows the `register` modifier for automatic variables, but this is usually unnecessary as the compiler will use registers in preference to locals on the stack (for speed reasons).

A modifier is used to further classify a variable's type

Modifier	Comment
`volatile`	can change value by means other than the current program
`const`	fixed value, defined in the source code and cannot be changed during execution
`unsigned`	range starts with 0, includes only positive values
`signed`	range includes both negative and positive values

Table 2.26 – Variable modifiers

A **`const`** modifier means a variable cannot be changed

In all cases `const` means the variable has a fixed value and cannot be changed. When modifying a global on an embedded system like the K64, it also means the parameter will be allocated in Flash memory. In the following example, `CR` is allocated in Flash memory. When `const` is added to a parameter or a local variable, it means that parameter cannot be modified by the function. It does not change where the parameter is allocated. For example, this example is legal:

```
unsigned char const CR = 13;
void LegalFunction(short count)
{
  while (count)
  {
    UART_OutChar(CR);
    count--;
  }
}
```

On the other hand, the following example is not legal because the function attempts to modify the input parameter. `count` in this example would have been allocated on the stack or in a register.

```
void IllegalFunction(const short count)
{
  while (count)
  {
    UART_OutChar(13);
    count--;  // this operation is illegal
  }
}
```

Similarly, the following example is not legal because the function attempts to modify the local variable. `COUNT` in this example would have been substituted by the value 5.

```
void IllegalFuntion2(void)
{
  const short COUNT = 5;
  while (COUNT)
  {
    UART_OutChar(13);
    COUNT--;  // this operation is illegal
  }
}
```

2.4.9 Character Variables

Character variables are stored as 8-bit quantities. When they are fetched from memory, they are always promoted automatically to 32-bit integers. Unsigned 8-bit values are promoted by adding 24 zeros into the most significant bits. Signed values are promoted by copying the sign bit (bit7) into the 24 most significant bits.

Characters are 8-bit quantities promoted to 32-bits

2.4.10 Mixing Signed and Unsigned Variables

There is a confusion when signed and unsigned variables are mixed into the same expression. It is good programming practice to avoid such confusions. As with integers, when a signed character enters into an operation with an unsigned quantity, the character is interpreted as though it was unsigned. The result of such operations is also unsigned. When a signed character joins with another signed quantity, the result is also signed.

Do not mix signed and unsigned variables in expressions

```
char x;   // signed 8-bit global
unsigned short y;   // unsigned 16-bit global
void sub(void)
{
  y = y + x;
  // x treated as unsigned even though defined as signed
}
```

Listing 2.43 – Code showing the mixture of signed and unsigned variables

There is also a need to change the size of characters when they are stored, since they are represented in the CPU as 32-bit values. In this case, however, it does not matter whether they are signed or unsigned. Obviously there is only one reasonable way to put a 32-bit quantity into an 8-bit location. When the high-order byte is chopped off, an error might occur. It is the programmer's responsibility to ensure that significant bits are not lost when characters are stored.

2.4.11 When Do We Use Automatics Versus Statics?

Because their contents are allowed to change, all variables must be allocated in RAM and not Flash memory. An *automatic variable* contains temporary information used only by one software module. Automatic variables are typically allocated, used, then deallocated from the stack. Since an interrupt will save registers and create its own stack frame, the use of automatic variables is important for creating re-entrant software. Automatic variables provide protection, limiting the scope of access in such a way that only the program that created the local variable can access it. The information stored in an automatic variable is not permanent. This means if we store a value into an automatic variable during one execution of the module, the next time that module is executed the previous value is not available. Typically we use automatics for loop counters and temporary sums. We use an automatic variable to store data that is temporary in nature. In summary, reasons why we place automatic variables on the stack include:

Automatic variables contain temporary information used only by one software module, and they are interrupt proof

- dynamic allocation release allows for reuse of memory
- limited scope of access provides for data protection
- can be made re-entrant
- since absolute addressing is not used, the code is relocatable
- the number of variables is only limited by the size of the stack allocation

A *static variable* is information shared by more than one program module. For example, we use globals to pass data between the main (or background) process and an interrupt (or foreground) process. Static variables are not deallocated. The information they store is permanent. We can use static variables for the time of day, date, user name, temperature, pointers to shared data, etc. The MX compiler uses absolute addressing (direct or extended) to access the static variables.

Static variables contain information that is shared between software modules

2.4.12 Initialization of variables and constants

Most programming languages provide ways of specifying *initial values*; that is, the values that variables have when program execution begins. The MX compiler will initially set all static variables to zero. Constants must be initialized at the time they are declared, and we have the option of initializing the variables.

Static variables are initialized to zero

Specifying initial values is simple. In its declaration, we follow a variable's name with an equals sign and a constant expression for the desired value. Thus

Constants must be initialized when they are declared

```
short Temperature = -55;
```

declares `Temperature` to be a 16-bit signed integer, and gives it an initial value of `−55`. Character constants with backslash-escape sequences are permitted. Thus

```
char Letter = '\t';
```

declares `Letter` to be a character, and gives it the value of the tab character. If array elements are being initialized, a list of constant expressions, separated by commas and enclosed in braces, is written. For example,

```
const unsigned short Steps[4] = {10, 9, 6, 5};
```

declares `Steps` to be an unsigned 16-bit constant integer array, and gives its elements the values `10`, `9`, `6`, and `5` respectively. If the size of the array is not specified, it is determined by the number of initializers. Thus

```
char Waveform[] = {28, 27, 60, 30, 40, 50, 60};
```

declares `Waveform` to be a signed 8-bit array of 7 elements which are initialized to `28`, `27`, `60`, `30`, `40`, `50`, `60`. On the other hand, if the size of the array is given and if it exceeds the number of initializers, the leading elements are initialized and the trailing elements default to zero. Therefore,

```
char Waveform[100] = {28, 27, 60, 30, 40, 50, 60};
```

declares `Waveform` to be an integer array of 100 elements, the first 7 elements of which are initialized to `28`, `27`, `60`, `30`, `40`, `50`, `60` and the others to zero. Finally, if the size of an array is given and there are too many initializers, the compiler generates an error message.

Character arrays can be initialized with a character string

Character arrays and character pointers may be initialized with a character string. In these cases, a terminating zero is automatically generated. For example,

```
char Name[5] = "John";
```

declares `Name` to be a character array of five elements with the first four initialized to `'J'`, `'o'`, `'h'` and `'n'` respectively. The fifth element contains zero. If the size of the array is not given, it will be set to the size of the string plus one. Thus

```
char Name[] = "John";
```

also contains the same five elements. If the size is given and the string is shorter, trailing elements default to zero. For example, the array declared by

```
char Name[7] = "John";
```

contains zeroes in its last three elements. If the string is longer than the specified size of the array, the array size is increased to match.

If we write

```
char *NamePtr = "John";
```

Pointers can be initialized to point to a constant character string

the effect is quite different from initializing an array. First a word (32 bits) is set aside for the pointer itself. This pointer is then given the address of the string. Then, beginning with that byte, the string and its zero terminator are assembled. The result is that `NamePtr` contains the address of the string `"John"`. The MX compiler accepts initializers for character variables, pointers, and arrays; and for integer variables and arrays. The initializers themselves may be either constant expressions, lists of constant expressions, or strings.

2.4.13 Implementation of the initialization

The compiler initializes static constants simply by defining its value in Flash memory (normally as part of the instruction for small values). In the following example, `J` is a static constant, and `K` is a literal.

```
short I;              // 16-bit global
const short J = 96;   // 16-bit constant
#define K 97;
void main(void)
{
  I = J;
  I = K;
}
```

Listing 2.44 – Example showing the initialization of a static constant

The K64 code generated by the MX compiler is as follows

```
    .text
main:
    movs  r2,#96   ;constant J=96
    ldr   r3,I
    strh  r2,[r3]  ;I=J (store half-word, 16-bits)
    ldr   r3,I
    movs  r2,#97
    strh  r2,[r3]  ;I=K (store half-word, 16-bits)
    bx    lr

    .word I
```

Notice the use of the **`#define`** macro which is used to implement an operation that is equivalent to `I = 97;`.

The compiler initializes a static variable by defining its initial value in Flash memory. It creates another segment called *.rodata* (in addition to the *.data* and *.text* sections). It places the initial values in the *.rodata* segment, then copies the data dynamically from *.rodata* Flash memory into *.data* RAM variables at the start of the program (before **`main`** is started). For example

```
short I = 95;         // 16-bit global
void main(void)
{
  ...
}
```

For the MX compiler, code in the `startup.c` file will copy the `95` from `.rodata` (Flash memory) into `I` in `.bss` (RAM) upon a hardware reset.

This copy is performed transparently before the main program is started.

```
        .text
main:
        ...
        bx      lr

        .global     I
        .section    .data.I,"aw",%progbits
        .align      1
I:
        .short      95
```

Even though the following two initializations of a global variable are technically proper, the explicit initialization of a global variable is a better style.

```
// good style
int I;
void main(void)
{
  I = 95;
}
```

```
// poor style
int I = 95;
void main(void)
{

}
```

A good understanding of the assembly code generated by our compiler makes us better programmers. (2.12)

2.4.14 Summary of Variable Attributes

Every variable possesses a number of different attributes, as summarized in the table below:

Attribute	**Description**
Type	`char`, `int`, `unsigned int`, etc. (*also implies size, range and resolution*)
Name	The identifier used to access the variable.
Value	The data held within the variable.
Address	The location in memory where the variable resides.
Scope	That part of the source code where the variable's name is recognized.
Lifetime	A notion of when the variable is created and destroyed, and thus when it is available for use.

Table 2.27 – Attributes of variables stored in memory

2.4.15 Summary of Variable Lifetimes

C offers three basic types of memory allocation as summarized below:

Method	**Variable is created…**	**Variable is initialized…**	**Variable is destroyed…**
Automatic	*Each* time the program enters the *function* in which it is declared.	If specified in the declaration, initialization occurs *each* time the program enters the *block*.	*Each* time the function *returns*.
Static	*Once*: When the program is first loaded into memory.	*Once*: Just before the program starts to run.	*Once*: When the program stops.
Dynamic	By calling the library function `malloc`.	By writing executable statements that modify its content.	By calling the library function `free`.

Table 2.28 – Types of memory allocation available in C

Each was designed for a different purpose; understanding their behaviour is crucial in order to take advantage of their capability.

2.5 Expressions

An expression is a combination of constants, variables array elements and function calls joined by operators

Most programming languages support the traditional concept of an expression as a combination of constants, variables, array elements, and function calls joined by various operators (+, -, etc.) to produce a single numeric value. Each operator is applied to one or two operands (the values operated on) to produce a single value which may itself be an operand for another operator. This idea is generalized in C by including non-traditional data types and a rich set of operators. Pointers, unsubscripted array names, and function names are allowed as operands. As Table 2.29 through to Table 2.34 illustrate, many operators are available. All of these operators can be combined in any useful manner in an expression. As a result, C allows the writing of very compact and efficient expressions which at first glance may seem a bit strange. Another unusual feature of C is that anywhere the syntax calls for an expression, a list of expressions, with comma separators, may appear.

2.5.1 Precedence and Associativity

The basic problem in evaluating expressions is deciding which parts of an expression are to be associated with which operators. To eliminate ambiguity, operators are given three properties: *operand count*, *precedence*, and *associativity*.

Operand count refers to how many variables the operator is applied to

Operand count refers to the classification of operators as unary, binary, or ternary according to whether they operate on one, two, or three operands. The unary minus sign, for instance, reverses the sign of the following operand, whereas the binary minus sign subtracts one operand from another.

The following example converts the distance x in inches to a distance y in cm. Without parentheses the following statement seems ambiguous:

```
y = 254 * x / 100;
```

If we divide first, then y can only take on values that are multiples of 254 (e.g., 0, 254, 508 etc.), so the following statement is incorrect:

```
y = 254 * (x / 100);
```

The proper approach is to multiply first then divide. To multiply first we must guarantee that the product `254 * x` will not overflow the precision of the computer. How do we know what precision the compiler used for the intermediate result `254 * x`? To answer this question, we must observe the assembly code generated by the compiler. Since multiplication and division associate left to right, the first statement without parentheses, although ambiguous will actually calculate the correct answer. It is good programming style to use parentheses to clarify the expression. The following statement has both good style and proper calculation:

Precedence and associativity are confusing – use parentheses

```
y = (254 * x) / 100;
```

The issues of precedence and associativity were explained in an earlier section. Precedence defines the evaluation order. For example the expression `3+4*2` will be `11` because multiplication has precedence over addition. Associativity determines the order of execution for operators that have the same precedence. For example, the expression `10-3-2` will be `5`, because subtraction associates left to right. On the other hand, if `x` and `y` are initially `10`, then the expression `x+=y+=1` will first make `y=y+1` (`11`), then make `x=x+y` (`21`) because the operator `+=` associates right to left. Refer to Table 2.4 for a list of operators and their precedence and associativity.

Associativity determines order of execution for operators with the same precedence

2.5.2 Unary operators

Unary operators have a single input, and a single output

Unary operators take a single input and give a single output. In the following examples, assume all numbers are 16-bit signed (`short`). The following variables are listed:

```
short data; // -32768 to +32767
short *pt;  // pointer to memory
short flag; // 0 is false, not zero is true
```

Operator	Meaning	Example	Result
`~`	binary complement	`~0x1234`	`0xEDCB`
`!`	logical complement	`!flag`	flip `0` to `1` and not zero to `0`
`&`	address of	`&data`	address in memory where data is stored
`-`	negate	`-100`	negative `100`
`+`	positive	`+100`	`100`
`++`	preincrement	`++data`	`data=data+1`, then result is `data`
`--`	predecrement	`--data`	`data=data-1`, then result is `data`
`*`	dereference	`*pt`	16-bit information pointed to by `pt`

Table 2.29 – Unary prefix operators

Operator	Meaning	Example	Result
`++`	Postincrement	`data++`	result is `data`, then `data=data+1`
`--`	Postdecrement	`data--`	result is `data`, then `data=data-1`

Table 2.30 – Unary postfix operators

2.5.3 Binary operators

Binary arithmetic operators operate on two number inputs giving a single number result. The operations of addition, subtraction and shift left are the same independent of whether the numbers are signed or unsigned. As we will see later, overflow and underflow after an addition, subtraction and shift left are different for signed and unsigned numbers, but the operation itself is the same. On the other hand multiplication, division, and shift right have different functions depending on whether the numbers are signed or unsigned. It will be important, therefore, to avoid multiplying or dividing an unsigned number with a signed number.

Binary operators have two inputs, and one output

Operator	Meaning	Example	Result
`+`	addition	`100 + 300`	`400`
`-`	subtraction	`100 - 300`	`-200`
`*`	multiplication	`10 * 300`	`3000`
`/`	division	`123 / 10`	`12`
`%`	remainder	`123 % 10`	`3`
`<<`	shift left	`102 << 2`	`408`
`>>`	shift right	`102 >> 2`	`25`

Table 2.31 – Binary arithmetic operators

The binary bitwise logical operators take two inputs and give a single result.

Operator	Meaning	Example	Result
`&`	bitwise AND	`0x1234 & 0x00FF`	`0x0034`
`|`	bitwise OR	`0x1234 | 0x00FF`	`0x12FF`
`^`	bitwise XOR	`0x1234 ^ 0x00FF`	`0x12CB`

Table 2.32 – Binary bitwise logical operators

The binary Boolean operators take two Boolean inputs and give a single Boolean result.

Operator	Meaning	Example	Result
&&	AND	`0 && 1`	`0 (false)`
||	OR	`0 || 1`	`1 (true)`

Table 2.33 – Binary Boolean operators

Don't confuse bitwise logical operators with Boolean operators

Many programmers confuse the logical operators with the Boolean operators. Logical operators take two numbers and perform a bitwise logical operation. Boolean operators take two Boolean inputs (`0` and not zero) and return a Boolean (`0` or `1`). In the program below, the operation `c = a & b;` will perform a bitwise logical AND of `0x0F0F` and `0xF0F0` resulting in `0x0000`. In the `d = a && b;` expression, the value a is considered as a `TRUE` (because it is not zero) and the value b also is considered a `TRUE` (not zero). The Boolean operation of `TRUE` AND `TRUE` gives a `TRUE` result (`1`).

```
short a, b, c, d;
void main(void)
{
  a = 0x0F0F;
  b = 0xF0F0;
  c = a & b;  // logical result c will be 0x0000
  d = a && b; // Boolean result d will be 1 (true)
}
```

Listing 2.45 – The difference between logical and Boolean operators

The binary relational operators take two number inputs and give a single Boolean result.

Binary relational operators

Operator	Meaning	Example	Result
`==`	is equal to	`100 == 200`	`0` (false)
`!=`	is not equal to	`100 != 200`	`1` (true)
`<`	less than	`100 < 200`	`1` (true)
`<=`	less than or equal to	`100 <= 200`	`1` (true)
`>`	greater than	`100 > 200`	`0` (false)
`>=`	greater than or equal to	`100 >= 200`	`0` (false)

Table 2.34 – Binary relational operators

Some programmers confuse *assignment equals* (`=`) with the *relational equals* (`==`). In the following example, the first `if` will execute the `subfunction()` if `a` is equal to zero (`a` is not modified). In the second case, the variable `b` is set to zero, and the `subfunction()` will never be executed because the result of the equals assignment is the value (in this case the `0` means false).

Don't confuse assignment equals (=) with relational equals (==)

```
short a, b;
void program(void)
{
  if (a == 0)
    subfunction(); // execute subfunction if a is zero
  if (b = 0)
    subfunction();//set b to zero, never execute subfunction
}
```

Listing 2.46 – The difference between relational and assignment equals

Before looking at the kinds of expressions we can write in C, we will first consider the process of evaluating expressions and some general properties of operators.

2.5.4 Assignment Operators

The assignment operator

The *assignment* operator is used to store data into variables. The syntax is `variable = expression;` where `variable` has been previously defined. At run-time, the result of the expression is saved into the variable. If the type of the expression is different from the variable, then the result is automatically converted. The assignment operation itself has a result, so the assignment operation can be nested.

```
short a, b;
void initialize(void)
{
  a = b = 0; // set both variables to zero
}
```

Listing 2.47 – Example of a nested assignment operation

The read / modify / write assignment operators are convenient. Examples are shown below.

Read / modify / write assignment operators

```
short a, b;
void initialize(void)
{
  a += b;  // same as a = a + b
  a -= b;  // same as a = a - b
  a *= b;  // same as a = a * b
  a /= b;  // same as a = a / b
  a %= b;  // same as a = a % b
  a <<= b; // same as a = a << b
  a >>= b; // same as a = a >> b
  a |= b;  // same as a = a | b
  a &= b;  // same as a = a & b
  a ^= b;  // same as a = a ^ b
}
```

Listing 2.48 – List of all read / modify / write assignment operations

Most compilers will produce the same code for the short and long version of the operation. Therefore you should use the read / modify / write operations only in situations that make the software easier to understand.

```
void function(void)
{
  GPIOA_PDOR |= 0x01;  // set PA0 high
  GPIOB_PDOR &= ~0x80; // clear PB7 low
  GPIOC_PDOR ^= 0x40;  // toggle PC6
}
```

Listing 2.49 – Good examples of read/modify/write assignment operations

2.5.5 Expression Types and Explicit Casting

We saw earlier that numbers are represented in the computer using a wide range of formats. A list of these formats is given in Table 2.35. Notice that for the MC9S12, the `int` and `short` types are the same. On the other hand, with the K64, the `int` and `long` types are the same. This difference may cause confusion when porting code from one system to another. You should use the `short` type when you are interested in efficiency and don't care about precision, and use the `long` type when you want a variable with a 32-bit precision.

Declare the size of an integer type explicitly

Type	Range	Precision	Example Variable
`unsigned char`	`0` to `255`	8 bits	`unsigned char uc;`
`char`	`-128` to `127`	8 bits	`char sc;`
`unsigned short`	`0` to `65535U`	16 bits	`unsigned short us;`
`short`	`-32768` to `32767`	16 bits	`short ss;`
`unsigned long`	`0` to `4294967295UL`	32 bits	`unsigned long ui;`
`long`	`-2147483648L` to `2147483647L`	32 bits	`long sl;`

Table 2.35 – Available number formats for the MX compiler

What happens when two numbers of different types are operated on? Before operation, the C compiler will first convert one or both numbers so they have the same type. The conversion of one type into another has many names:

Names for type conversion

- automatic conversion;
- implicit conversion;
- coercion;
- promotion; or
- widening.

Promotion of type

There are three ways to consider this issue. The first way to think about this is if the range of one type completely fits within the range of the other, then the number with the smaller range is converted (promoted) to the type of the number with the larger range. In the following examples, a number of *type1* is added to a number of *type2*. In each case, the number range of *type1* fits into the range of *type2*, so the parameter of *type1* is first promoted to *type2* before the addition.

Type1		Type2	Example
`unsigned char`	fits inside	`unsigned short`	`uc + us` is of type `unsigned short`
`unsigned char`	fits inside	`short`	`uc + ss` is of type `short`
`unsigned char`	fits inside	`long`	`uc + sl` is of type `long`
`char`	fits inside	`short`	`sc + ss` is of type `short`
`char`	fits inside	`long`	`sc + sl` is of type `long`
`unsigned short`	fits inside	`long`	`us + sl` is of type `long`
`short`	fits inside	`long`	`ss + sl` is of type `long`

Table 2.36 – Conversion when the range of one type fits inside another

The second way to consider mixed precision operations is that in most cases the compiler will promote the number with the smaller precision into the other type before operation. If the two numbers are of the same precision, then the signed number is converted to unsigned. These automatic conversions may not yield correct results. The third and best way to deal with mixed type operations is to perform the conversions explicitly using the *cast* operation. We can force the type of an expression by explicitly defining its type. This approach allows the programmer to explicitly choose the type of the operation.

Use a typecast to force a particular type conversion

Consider the following digital filter with mixed type operations. In this example, we explicitly convert `x` and `y` to signed 16-bit numbers and perform 16-bit signed arithmetic. Note that the assignment of the result into `y`, will require a demotion of the 16-bit signed number into an 8-bit signed number. Unfortunately, C does not provide any simple mechanisms for error detection / correction.

```c
char y; // output of the filter
unsigned char x; // input of the filter
void filter(void)
{
  y = (12 * (short)x + 56 * (short)y) / 100;
}
```

Listing 2.50 – Example of converting types with the cast operator

An explicit cast is achieved by preceding the expression with parentheses surrounding the type

We apply an explicit *cast* simply by preceding the number or expression with parentheses surrounding the type.

In the next digital filter all numbers are of the same type. Even so, we are worried that the intermediate result of the multiplications and additions might overflow the 16-bit arithmetic. We know from digital signal processing that the final result will always fit into the 16-bit variable. In this example, the cast `(long)` will specify the calculations be performed in 32-bit precision.

```c
// y(n) = [113 * x(n) + 113 * x(n-2) - 98 * y(n-2)] / 128
// channel specifies the A/D channel
// arrays containing current and previous values
short x[3], y[3];
void Filter (void)
{
  // shift arrays
  y[2] = y[1];
  y[1] = y[0];
  x[2] = x[1];
  x[1] = x[0];
  x[0] = ADC_Get(channel); // new data
  y[0] = (113 * ((long)x[0] + (long)x[2]) - 98
          * (long)y[2]) / 128;
}
```

A cast is used to give a symbolic name to a microcontroller port or register

Listing 2.51 – We can use a cast to force higher precision arithmetic

We saw previously that casting was used to assign a symbolic name to an I/O port. In particular the following `#define` casts the number `0x400FF000` as a pointer type, which points to volatile unsigned 32-bit data.

```c
#define GPIOA_PDOR *(uint32_t volatile *)(0x400FF000)
```

2.5.6 Selection operator

The selection operator

The selection operator takes three input parameters and yields one output result. The format is

```
Expr1 ? Expr2 : Expr3
```

The first input parameter is an expression, `Expr1`, which yields a Boolean (`0` for false, not zero for true). `Expr2` and `Expr3` return values that are regular numbers. The selection operator will return the result of `Expr2` if the value of `Expr1` is true, and will return the result of `Expr3` if the value of `Expr1` is false. The type of the expression is determined by the types of `Expr2` and `Expr3`. If `Expr2` and `Expr3` have different types, then the usual promotion is applied. The resulting type is determined at compile time, in a similar manner as the `Expr2 + Expr3` operation, and not at run-time depending on the value of `Expr1`. The following two subroutines have identical functions.

```c
short a, b;
void sub1(void)
{
  a = (b==1) ? 10 : 1;
}

void sub2(void)
{
  if (b == 1)
    a = 10;
  else
    a = 1;
}
```

Listing 2.52 – Example of the selection operator

2.5.7 Arithmetic Overflow and Underflow

An important issue when performing arithmetic calculations on integer values is the problem of *underflow* and *overflow*. Arithmetic operations include addition, subtraction, multiplication, division and shifting. Overflow and underflow errors can occur during all of these operations. In assembly language the programmer is warned that an error has occurred because the processor will set condition code bits after each of these operations. Unfortunately, the C compiler provides no direct access to these error codes, so we must develop careful strategies for dealing with overflow and underflow. It is important to remember that arithmetic operations (addition, subtraction, multiplication, division, and shifting) have constraints when performed with finite precision on a microcomputer. An overflow error occurs when the result of an arithmetic operation cannot fit into the finite precision of the result. We will study addition and subtraction operations in detail, but the techniques for dealing with overflow and underflow will apply to the other arithmetic operations as well. We will consider two approaches:

Overflow and underflow can occur when performing arithmetic

An overflow occurs when the result of an operation cannot fit into the finite precision of the result

- avoiding the error
- detecting the error then correcting the result

For example when two 8-bit numbers are added, the sum may not fit back into the 8-bit result. We saw earlier that the same digital hardware (instructions) could be used to add and subtract unsigned and signed numbers. Unfortunately, we will have to design separate overflow detection for signed and unsigned addition and subtraction.

All microcomputers have a condition code register which contain bits which specify the status of the most recent operation. In this section, we will introduce 4 condition code bits common to most microcomputers. If the two inputs to an addition or subtraction operation are considered as unsigned, then the `C` bit (carry) will be set if the result does not fit. In other words, after an unsigned addition, the `C` bit is set if the answer is wrong. If the two inputs to an addition or subtraction operation are considered as signed, then the `V` bit (overflow) will

Condition code bits are used to detect an overflow or underflow be set if the result does not fit. In other words, after a signed addition, the `V` bit is set if the answer is wrong.

bit	name	Meaning after addition or subtraction
`N`	negative	result is negative
`Z`	zero	result is zero
`V`	overflow	signed overflow
`C`	carry	unsigned overflow

Table 2.37 – Condition code bits contain the status of the previous arithmetic or logical operation

For unsigned numbers, errors occur when crossing the 0|255 boundary

For an 8-bit unsigned number, there are only 256 possible values, 0 to 255. We can think of the numbers as positions along a circle. There is a discontinuity at the 0|255 interface, everywhere else adjacent numbers differ by ± 1. If we add two unsigned numbers, we start at the position of the first number and move in a clockwise direction the number of steps equal to the second number. For example, if `96 + 64` is performed in 8-bit unsigned precision, the correct result of `160` is obtained. In this case, the carry bit will be `0` signifying the answer is correct. On the other hand, if `224 + 64` is performed in 8-bit unsigned precision, the incorrect result of `32` is obtained. In this case, the carry bit will be `1`, signifying the answer is wrong.

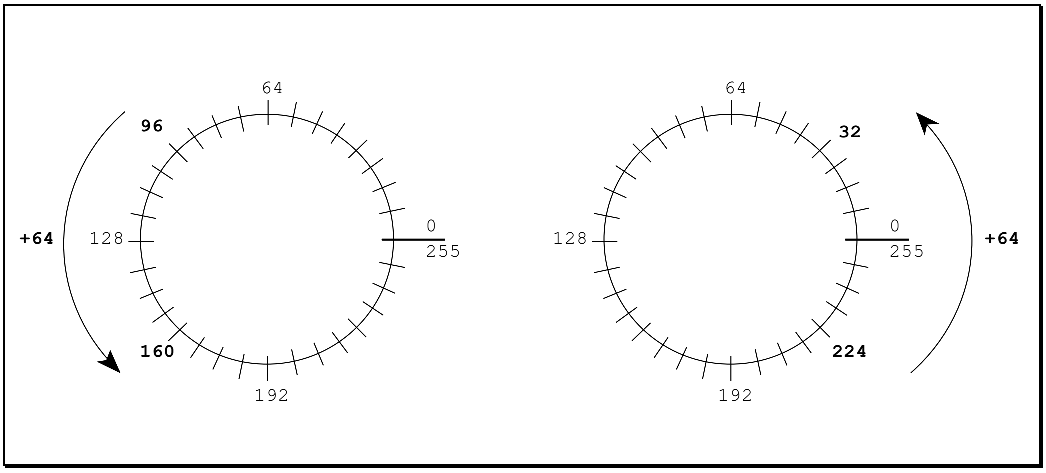


Figure 2.13 – 8-bit unsigned addition

For subtraction, we start at the position of the first number and move in a counter clockwise direction the number of steps equal to the second number. For example, if `160 - 64` is performed in 8-bit unsigned precision, the correct result of `96` is obtained (carry bit will be `0`). On the other hand, if `32 - 64` is performed in 8-bit unsigned precision, the incorrect result of `224` is obtained (carry bit will be `1`) .

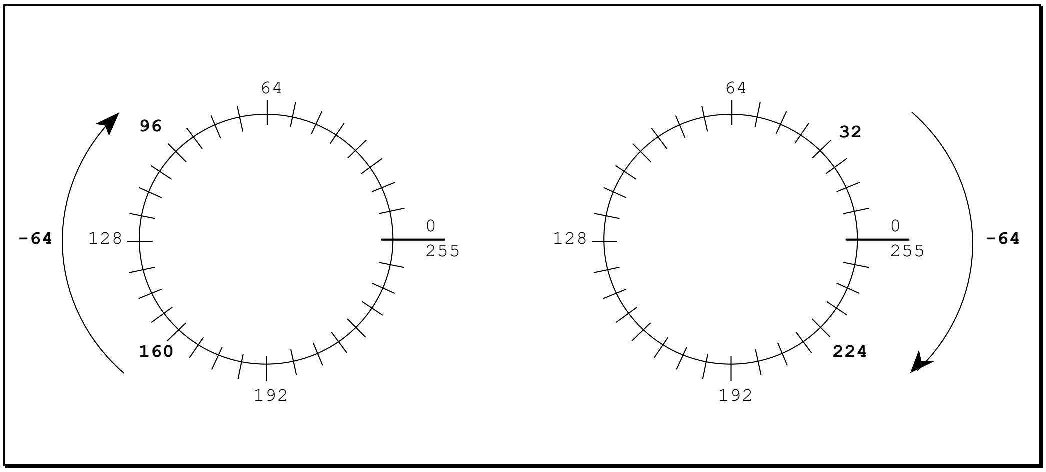


Figure 2.14 – 8-bit unsigned subtraction

In general, we see that the carry bit is set when we cross over from `255` to `0` while adding or cross over from `0` to `255` while subtracting.

The carry bit, `C`, is set after an unsigned add or subtract when the result is incorrect.	(2.13)

For an 8-bit signed number, the possible values range from -128 to 127. Again there is a discontinuity, but this time it exists at the -128|127 interface, everywhere else adjacent numbers differ by ±1. The meanings of the numbers with bit 7 = 1 are different from unsigned, but we add and subtract signed numbers on the number wheel in a similar way (e.g., addition of a positive number moves counterclockwise.) Adding a negative number is the same as subtracting a positive number hence this operation would cause a clockwise motion. For example, if `-32 + 64` is performed, the correct result of `32` is obtained. In this case, the overflow bit will be `0` signifying the answer is correct. On the other hand, if `96 + 64` is performed, the incorrect result of

For signed numbers, errors occur when crossing the -128|127 boundary

`-96` is obtained. In this case, the overflow bit will be `1` signifying the answer is wrong.

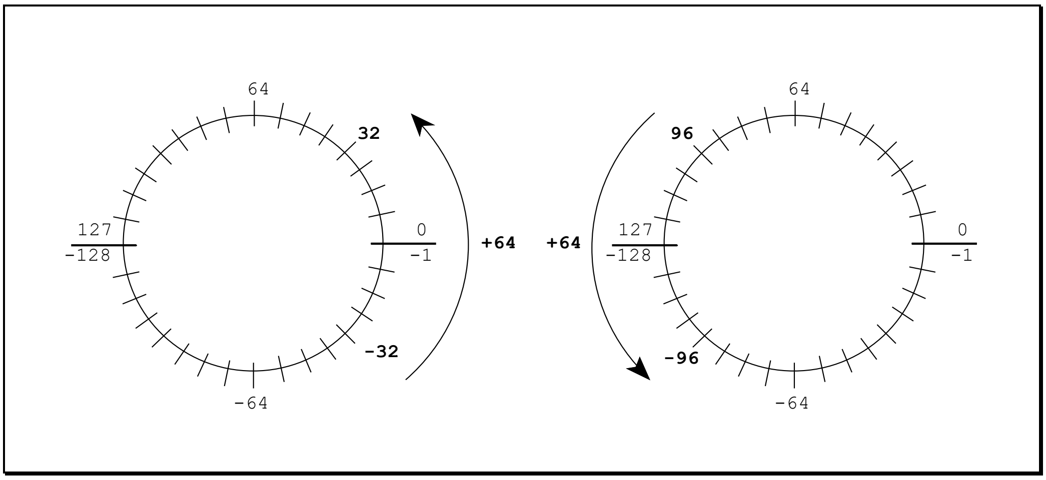


Figure 2.15 – 8-bit signed addition

For subtracting signed numbers, we again move in a clockwise direction. Subtracting a negative number is the same as adding a positive number hence this operation would cause a counterclockwise motion. For example, if `32 - 64` is performed, the correct result of `-32` is obtained (overflow bit will be `0`). On the other hand, if `-96 - 64` is performed, the incorrect result of `96` is obtained (overflow bit will be `1`).

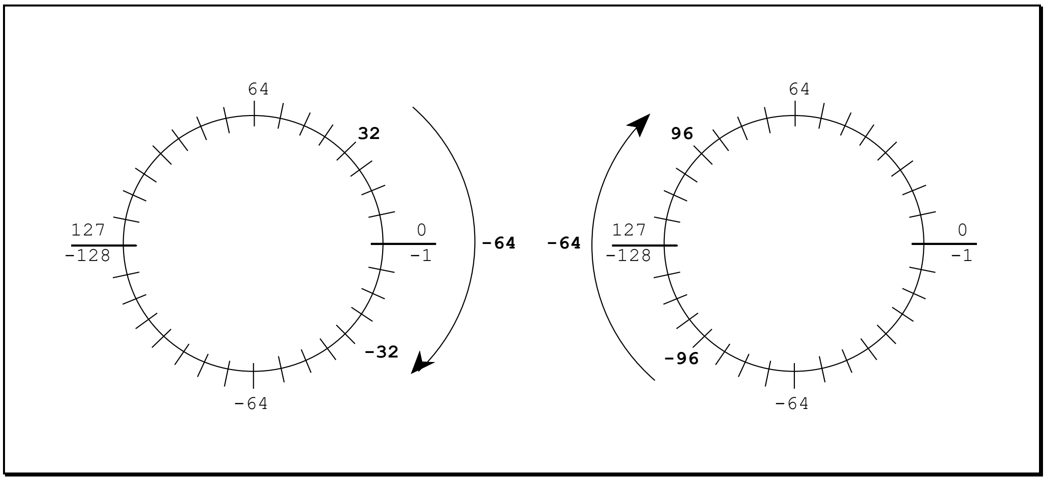


Figure 2.16 – 8-bit signed subtraction

In general, we see that the overflow bit is set when we cross over from `127` to `-128` while adding or cross over from `-128` to `127` while subtracting.

The overflow bit, `V`, is set after a signed add or subtract when the result is incorrect.

(2.14)

Overflow can be detected by Boolean operations on the individual bits

Another way to determine the overflow bit after an addition is to consider the carry out of bit 6. The `V` bit will be set of there is a carry out of bit 6 (into bit 7) but no carry out of bit 7 (into the `C` bit). It is also set if there is no carry out of bit 6 but there is a carry out of bit 7. Let *X*7-*X*0 and *M*7-*M*0 be the individual binary bits of the two 8-bit numbers which are to be added, and let *R*7-*R*0 be individual binary bits of the 8-bit sum. Then, the 4 condition code bits after an addition are shown in Table 2.38.

`N`	R7	if unsigned result above 127, if signed result is negative
`Z`	$\overline{R7}\cdot\overline{R6}\cdot\overline{R5}\cdot\overline{R4}\cdot\overline{R3}\cdot\overline{R2}\cdot\overline{R1}\cdot\overline{R0}$	result is zero
`V`	$\overline{X7}\cdot\overline{M7}\cdot R7$ $+X7\cdot M7\cdot\overline{R7}$	add two positives get a negative result; or add two negatives get a positive result
`C`	$X7\cdot M7$ $+M7\cdot\overline{R7}$ $+X7\cdot\overline{R7}$	add two numbers both above 127; or add one number above 127 and get a number below 128; or add one number above 127 and get a number below 128

Table 2.38 – Condition code bits after an 8-bit addition operation

Let the result R be the result of the subtraction X - M. Then the 4 condition code bits are shown in Table 2.39.

N	R7	if unsigned result above 127, if signed result is negative
Z	$\overline{R7}\cdot\overline{R6}\cdot\overline{R5}\cdot\overline{R4}\cdot\overline{R3}\cdot\overline{R2}\cdot\overline{R1}\cdot\overline{R0}$	result is zero
V	$\overline{X7}\cdot M7\cdot R7$ $+X7\cdot\overline{M7}\cdot\overline{R7}$	a positive minus a negative and get a negative result; or a negative minus a positive and get a positive result
C	$\overline{X7}\cdot M7$ $+M7\cdot R7$ $+\overline{X7}\cdot R7$	a number below 128 minus a number above 127; or subtracted a number above 127 and get one above 127; or started with a number below 127 and get one above 127

Table 2.39 – Condition code bits after an 8-bit subtraction operation

Ignoring overflow (signed or unsigned) can result in significant errors. (2.15)

Computers have two sets of conditional branch instructions (if statements) which make program decisions based on either the C or V bit. (2.16)

An error will occur if you use unsigned conditional branch instructions (if statements) after operating on signed numbers, and vice-versa. (2.17)

There are some applications where arithmetic errors are not possible. For example, if we had two 8-bit unsigned numbers that we knew were in the range of `0` to `100`, then no overflow is possible when they are added together.

Typically the numbers we are processing are either signed or unsigned (but not both), so we need only consider the corresponding `C` or `V` bit (but not both the `C` and `V` bits at the same time.) In other words, if the two numbers are unsigned, then we look at the `C` bit and ignore the `V` bit. Conversely, if the two numbers are signed, then we look at the `V` bit and ignore the `C` bit. There are two appropriate mechanisms to deal with the potential for arithmetic errors when adding and subtracting. The first mechanism, used by most compilers, is called promotion. Promotion involves increasing the precision of the input numbers, and performing the operation at that higher precision. An error can still occur if the result is stored back into the smaller precision. Fortunately, the program has the ability to test the intermediate result to see if it will fit into the smaller precision. To promote an unsigned number we add zero's to the left side.

Promotion is used by compilers to avoid overflow and underflow problems

2.116

Promotion of unsigned numbers to avoid overflow and underflow

In a previous example, we added the unsigned 8-bit `224` to `64`, and got the wrong result of `32`. With promotion we first convert the two 8-bit numbers to 16-bits, then add. We can check the 16-bit intermediate result (e.g., `228`) to see if the answer will fit back into the 8-bit result. In the following flowchart, X and M are 8-bit unsigned inputs, X_{16}, M_{16}, and R_{16} are 16-bit intermediate values, and R is an 8-bit unsigned output.

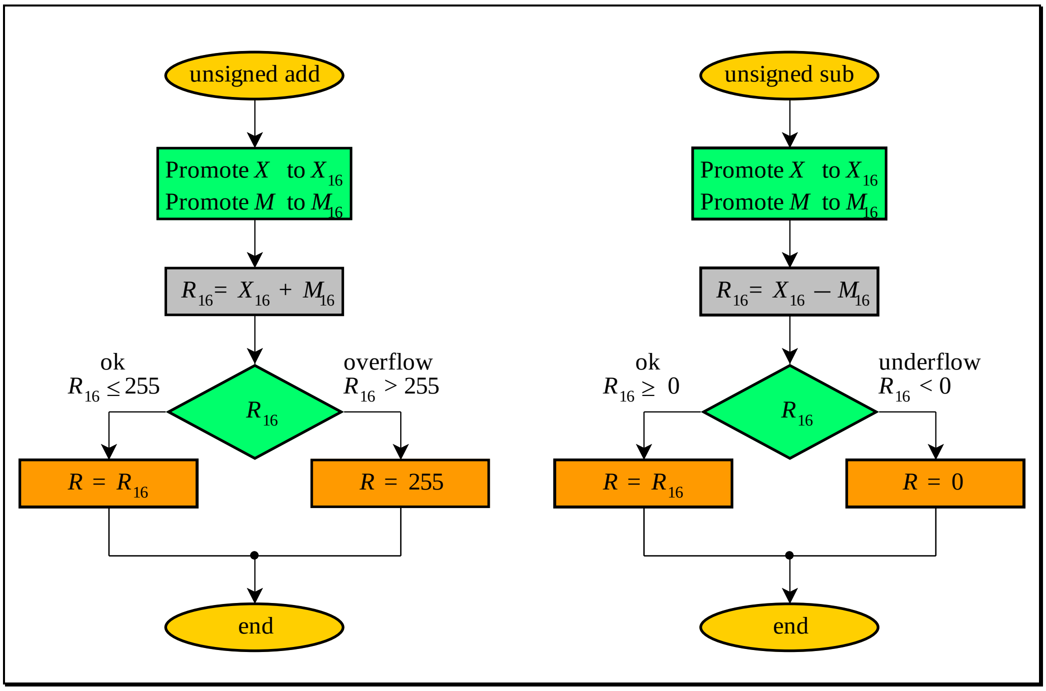


Figure 2.17 – Promotion can be used to avoid overflow and underflow

Signed numbers are promoted by extending the sign bit

To promote a signed number, we duplicate the sign bit as we add binary digits to the left side. Earlier, we performed the 8-bit signed operation `-96 - 64` and got a signed overflow. With promotion we first convert the two numbers to 16-bits, then subtract.

decimal	8-bit	16-bit
-96 -64 -160	1010 0000 $-$0100 0000 0110 0000	1111 1111 1010 0000 $-$0000 0000 0100 0000 1111 1111 0110 0000

We can check the 16-bit intermediate result (e.g., `-160`) to see if the answer will fit back into the 8-bit result. In the following flowchart, X and M are 8-bit signed inputs, X_{16}, M_{16}, and R_{16} are 16-bit signed intermediate values, and R is an 8-bit signed output.

Promotion of signed numbers to avoid overflow and underflow

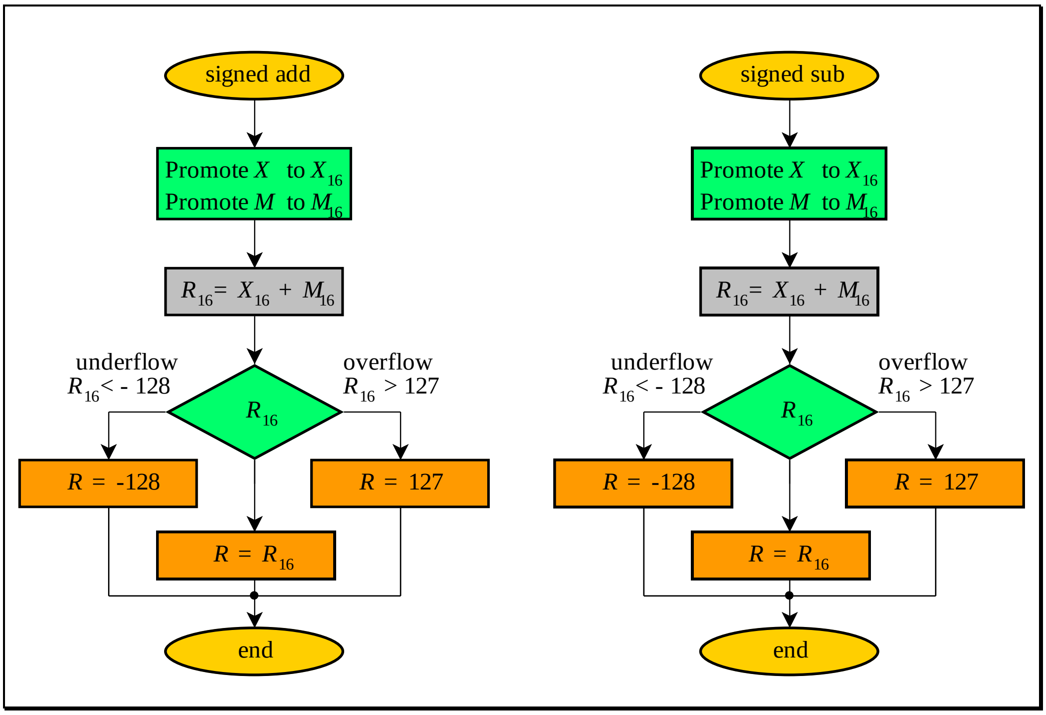


Figure 2.18 – Promotion can be used to avoid overflow and underflow

The other mechanism for handling addition and subtraction errors is called ceiling and floor. It is analogous to movements inside a room. If we try to move up (add a positive number or subtract a negative number) the ceiling will prevent us from exceeding the bounds of the room. Similarly, if we try to move down (subtract a positive number or add a negative number) the floor will prevent us from going too low. For our 8-bit addition and subtraction, we will prevent the `0` to `255` and `255` to `0` crossovers for unsigned operations and `-128` to `+127` and `+127` to `-128` crossovers for signed operations. These operations are described by the following flowcharts. If the carry bit is set after an unsigned addition the result is adjusted to the largest possible unsigned number (ceiling). If the carry bit is set after an unsigned subtraction, the result is adjusted to the smallest possible unsigned number (floor.)

Ceiling and floor can be used to avoid overflow and underflow

Using ceiling and floor of unsigned numbers to avoid overflow and underflow

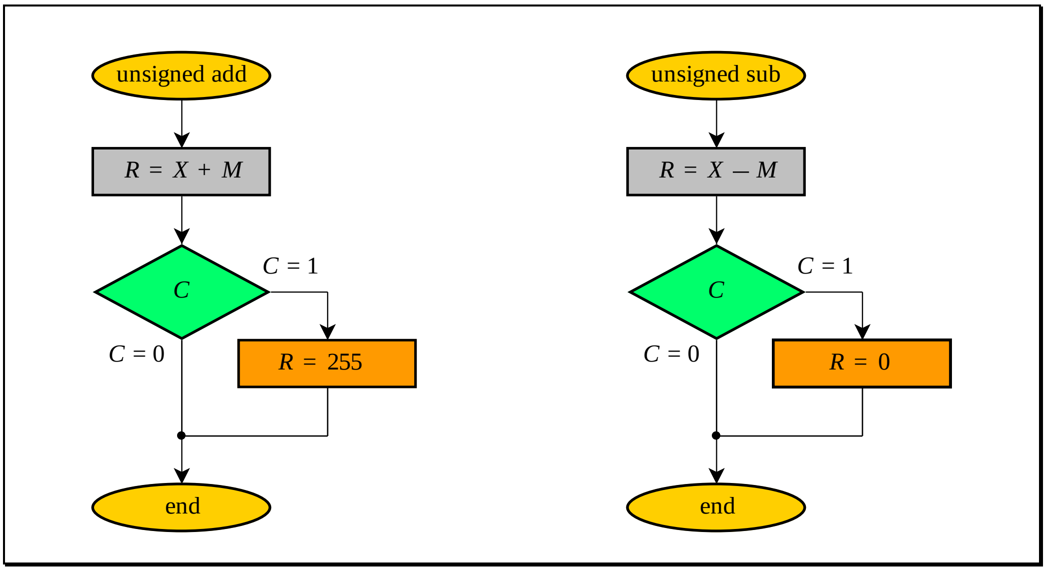


Figure 2.19 – In assembly language we can detect overflow and underflow

If the overflow bit is set after a signed operation the result is adjusted to the largest (ceiling) or smallest (floor) possible signed number depending on whether it was a `-128` to `127` cross over (`N` = 0) or `127` to `-128` cross over (`N` = 1). Notice that after a signed overflow, bit 7 of the result is always wrong because there was a cross over.

Using ceiling and floor of signed numbers to avoid overflow and underflow

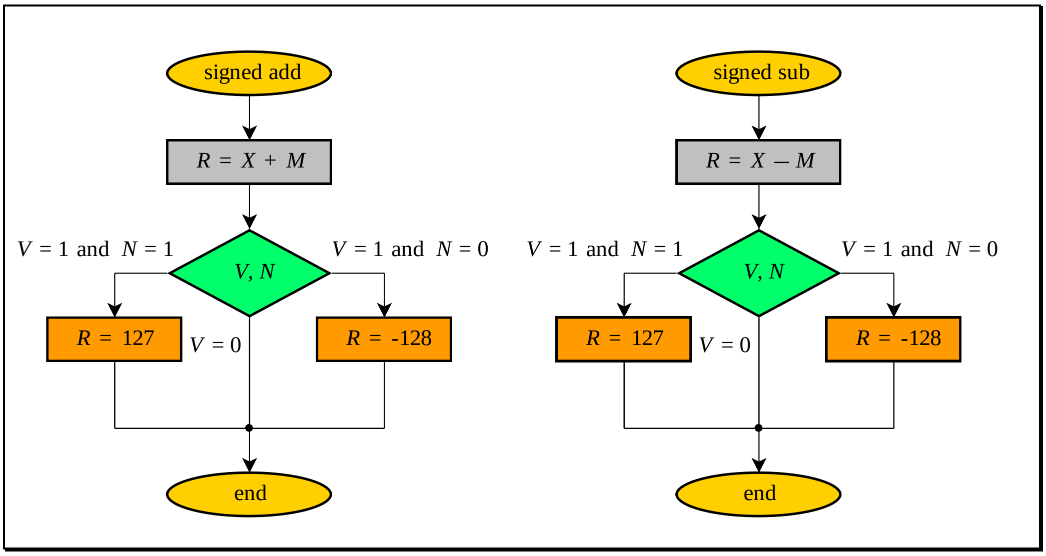


Figure 2.20 – In assembly language we can detect overflow and underflow

In summary, overflow and underflow occur *silently* during addition and subtraction of integer data types; there is no run-time checking provided by the microprocessor. It is entirely the programmer's responsibility to allocate a data type of the appropriate size for each variable and to avoid overflow.

2.6 Procedural Statements

Every procedural language provides statements for determining the flow of control within programs. Although declarations are a type of statement, in C the unqualified word *statement* usually refers to *procedural statements* rather than declarations. In this section we are concerned only with procedural statements.

In the C language, statements can be written only within the body of a function; more specifically, only within compound statements. The normal flow of control among statements is sequential, proceeding from one statement to the next. However, most of the statements in C are designed to alter this sequential flow so that algorithms of arbitrary complexity can be implemented. This is done with statements that control whether or not other statements execute and, if so, how many times. Furthermore, the ability to write compound statements permits the writing of a sequence of statements wherever a single, possibly controlled, statement is allowed. These two features provide the necessary generality to implement any algorithm, and to do it in a structured way.

Compound statements and decision statements are used to make algorithms of arbitrary complexity

2.6.1 Simple Statements

The C language uses semicolons as statement terminators. A semicolon follows every simple (non-compound) statement, even the last one in a sequence.

When one statement controls other statements, a terminator is applied only to the controlled statements. Thus we would write

Simple statements are terminated with a semicolon

```
if (x > 5)
  x = 0;
else
  x++;
```

with two semicolons, not three. Perhaps one good way to remember this is to think of statements that control other statements as "super" statements that "contain" ordinary (simple and compound) statements. Then remember that only simple statements are terminated. This implies, as stated above, that compound statements are not terminated with semicolons.

Thus

```
while (x < 5)
{
  func();
  x++;
}
```

is perfectly correct. Notice that each of the simple statements within the compound statement is terminated.

2.6.2 Compound Statements

The terms *compound statement* and *block* both refer to a collection of statements that are enclosed in braces to form a single unit. Compound statements have the form

```
{ ObjectDeclaration?... Statement?... }
```

`ObjectDeclaration?...` is an optional set of local declarations. If present, C requires that they precede the statements; in other words, they must be written at the head of the block. `Statement?...` is a series of zero or more simple or compound statements. Notice that there is no semicolon at the end of a block; the closing brace suffices to delimit the end. In the following example the local variable `temp` is only defined within the inner compound statement.

```c
void main(void)
{
  short n1, n2;
  n1 = 1;
  n2 = 2;
  {
    short temp;
    temp = n1;
    n1 = n2;
    n2 = temp; // switch n1, n2
  }
}
```

Compound statements are enclosed with braces { }

Listing 2.53 – Example of a compound statement

The power of compound statements derives from the fact that one may be placed anywhere the syntax calls for a statement. Thus any statement that controls other statements is able to control units of logic of any complexity.

When control passes into a compound statement, two things happen. First, space is reserved on the stack for the storage of local variables that are declared at the head of the block. Then the executable statements are processed.

One important limitation in C is that a block containing local declarations must be entered through its leading brace. This is because bypassing the head of a block effectively skips the logic that reserves space for local objects. The **`goto`** and **`switch`** statements (below) could violate this rule.

2.6.3 The `if` Statement

The `if` statement

`if` statements provide a non-iterative choice between alternate paths based on specified conditions. They have either of two forms

```
if (ExpressionList)
  Statement1
```

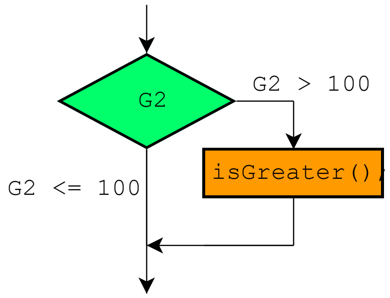
or

```
if (ExpressionList)
  Statement1
else
  Statement2
```

`ExpressionList` is a list of one or more expressions and `Statement` is any simple or compound statement. First, `ExpressionList` is evaluated and tested. If more than one expression is given, they are evaluated from left to right and the right-most expression is tested. If the result is true (non-zero), then the `Statement1` is executed and the `Statement2` (if present) is skipped. If it is false (zero), then `Statement1` is skipped and `Statement2` (if present) is executed.

In the following example, the function `isGreater()` is executed if `G2` is larger than `100`.

```
if (G2 > 100)
 isGreater();
```



Listing 2.54 – Example `if` statement

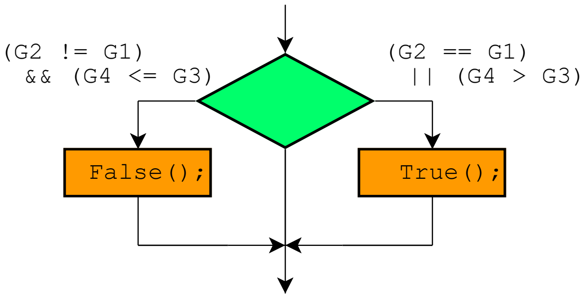
A 3-wide median filter can be designed using `if-else` conditional statements.

```
short Median(short u1, short u2, short u3)
{
  short result;
  if (u1 > u2)
    if (u2 > u3)
      result = u2;     // u1>u2,u2>u3       u1>u2>u3
    else
      if (u1 > u3)
        result = u3;   // u1>u2,u3>u2,u1>u3 u1>u3>u2
      else
        result = u1;   // u1>u2,u3>u2,u3>u1 u3>u1>u2
  else
    if (u3 > u2)
      result = u2;     // u2>u1,u3>u2       u3>u2>u1
    else
      if (u1 > u3)
        result = u1;   // u2>u1,u2>u3,u1>u3 u2>u1>u3
      else
        result = u3;   // u2>u1,u2>u3,u3>u1 u2>u3>u1
  return result;
}
```

Listing 2.55 – A 3-wide median function

Complex conditional testing can be implemented using the relational and Boolean operators described in the last section.

```
if ((G2 == G1) || (G4 > G3))
  True();
else
  False();
```



2.6.4 The `switch` Statement

`switch` statements provide a non-iterative choice between any number of paths based on specified conditions. They compare an expression to a set of constant values. Selected statements are then executed depending on which value, if any, matches the expression. Switch statements have the form

The `switch` statement

```
switch (ExpressionList)
{
 Statement?...
}
```

where `ExpressionList` is a list of one or more expressions. `Statement?...` represents the statements to be selected for execution. They are selected by means of `case` and `default` prefixes – special labels that are used only within `switch` statements. These prefixes locate points to which control jumps depending on the value of `ExpressionList`. They are to the `switch` statement what ordinary labels are to the `goto` statement. They may occur only within the braces that delimit the body of a `switch` statement.

The `case` prefix has the form

```
case ConstantExpression:
```

and the `default` prefix has the form

```
default:
```

The terminating colons are required; they heighten the analogy to ordinary statement labels. Any expression involving only numeric and character constants and operators is valid in the `case` prefix.

The `switch` statement uses `case` and `default` prefixes

After evaluating `ExpressionList`, a search is made for the first matching `case` prefix. Control then goes directly to that point and proceeds normally from there. Other `case` prefixes and the `default` prefix have no effect once a `case` has been selected; control flows through them just as though they were not even there. If no matching `case` is found, control goes to the `default` prefix, if there is one. In the absence of a `default` prefix, the entire compound statement is ignored and control resumes with whatever follows the `switch` statement. Only one `default` prefix may be used with each `switch`.

If it is not desirable to have control proceed from the selected prefix all the way to the end of the switch block, break statements may be used to exit the block. break statements have the form

The break statement is used to exit a block

```
break;
```

Some examples may help clarify these ideas. Assume Port A is specified as an output, and bits 3, 2, 1, and 0 are connected to a stepper motor. The switch statement will first read Port A and AND the data with 0x0000000F (GPIOA_PDOR & 0x0000000F). If the result is 5, then Port A is set to 6 and control is passed to the end of the switch (because of the break). Similarly for the other 3 possibilities.

```
#define GPIOA_PDOR *(uint32_t volatile *)(0x400FF000)
void step(void)
{
  // turn stepper motor one step
  switch (GPIOA_PDOR & 0x0000000F)
  {
    case 0x05:
      GPIOA_PDOR = 0x06; // 6 follows 5
      break;
    case 0x06:
      GPIOA_PDOR = 0x0A; // 10 follows 6
      break;
    case 0x0A:
      GPIOA_PDOR = 0x09; // 9 follows 10
      break;
    case 0x09:
      GPIOA_PDOR = 0x05; // 5 follows 9
      break;
    default:
      GPIOA_PDOR = 0x05; // start at 5
   }
}
```

Listing 2.56 – Example of the switch statement

This next example shows that multiple tests can be performed for the same condition.

```c
// ASCII to decimal digit conversion
unsigned char convert(unsigned char letter)
{
  unsigned char digit;
  switch (letter)
  {
    case 'A':
    case 'B':
    case 'C':
    case 'D':
    case 'E':
    case 'F':
      digit = letter + 10 - 'A';
      break;
    case 'a':
    case 'b':
    case 'c':
    case 'd':
    case 'e':
    case 'f':
      digit = letter + 10 - 'a';
      break;
    default:
      digit = letter - '0';
  }
  return digit;
}
```

Listing 2.57 – Example of the `switch` statement

The body of the `switch` is not a normal compound statement since local declarations are not allowed in it or in subordinate blocks. This restriction enforces the C rule that a block containing declarations must be entered through its leading brace.

2.6.5 The while Statement

The while statement is one of three statements that determine the repeated execution of a controlled statement. This statement alone is sufficient for all loop control needs. The other two merely provide an improved syntax and an execute-first feature. while statements have the form

```
while (ExpressionList) Statement
```

The while statement

where ExpressionList is a list of one or more expressions and Statement is a simple or compound statement. If more than one expression is given, the right-most expression yields the value to be tested. First, ExpressionList is evaluated. If it yields true (non-zero), then Statement is executed and ExpressionList is evaluated again. As long as it yields true, Statement executes repeatedly. When it yields false, Statement is skipped, and control continues with whatever follows.

In the example

```
i = 5;
while (i) array[--i] = 0;
```

elements 0 through 4 of array[] are set to zero. First i is set to 5. Then as long as it is not zero, the assignment statement is executed. With each execution i is decremented before being used as a subscript.

It is common to use the while statement to implement polling loops

```
#define RDRF 0x20 // Receive Data Register Full Bit
// Wait for new serial port input,
// return ASCII code for key typed
char UART_InChar(void)
{
  while ((UART2_S1 & RDRF) == 0);
  return UART2_D;
}

#define TDRE 0x80 // Transmit Data Register Empty Bit
// Wait for buffer to be empty, output ASCII to serial port
void UART_OutChar(char data)
{
  while ((UART2_S1 & TDRE) == 0);
  UART2_D = data;
}
```

Listing 2.58 – Examples of the while statement

`continue` and `break` statements are handy for use with the `while` statement (also helpful for the `do` and `for` loops). The `continue` statement has the form

The `continue` statement causes control to jump to the top of the control loop

```
continue;
```

It causes control to jump directly back to the top of the loop for the next evaluation of the controlling expression. If loop controlling statements are nested, then `continue` affects only the innermost surrounding statement. That is, the innermost loop statement containing the `continue` is the one that starts its next iteration.

The `break` statement (described earlier) may also be used to break out of loops. It causes control to pass on to whatever follows the loop controlling statement. If `while` (or any loop or `switch`) statements are nested, then `break` affects only the innermost statement containing the `break`. That is, it exits only one level of nesting.

2.6.6 The `for` Statement

The **`for`** statement also controls loops. It is really just an embellished **`while`** in which the three operations normally performed on loop-control variables (initialize, test, and modify) are brought together syntactically. It has the form

```
for (ExpressionList1?; ExpressionList2?; ExpressionList3?)
  Statement
```

The `for` statement

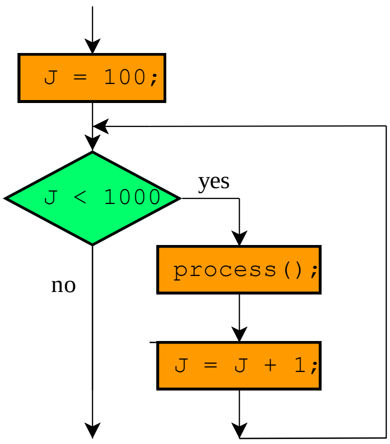
`for` statements are performed in the following steps:

`ExpressionList1` is evaluated. This is done only once to initialize the control variable(s).

`ExpressionList2` is evaluated to determine whether or not to perform `Statement`. If more than one expression is given, the right-most expression yields the value to be tested. If it yields false (zero), control passes on to whatever follows the `for` statement. But, if it yields true (non-zero), `Statement` executes.

`ExpressionList3` is then evaluated to adjust the control variable(s) for the next pass, and the process goes back to step 2. For example,

```
for (J = 100; J < 1000; J++)
{
  process();
}
```



A five-element array being set to zero could be written as

```
for (i = 4; i >= 0; --i)
  array[i] = 0;
```

or a little more efficiently as

```
for (i = 5; i; array[--i] = 0);
```

2.130

Any of the three expression lists may be omitted, but the semicolon separators must be kept. If the test expression is absent, the result is always true. Thus

```
for (;;)
{
  ...
  break;
  ...
}
```

will execute until the `break` is encountered.

As with the `while` statement, `break` and `continue` statements may be used with equivalent effects. A `break` statement makes control jump directly to whatever follows the `for` statement. A `continue` skips whatever remains in the controlled block so that the third `ExpressionList3` is evaluated, after which the second `ExpressionList2` is evaluated and tested. In other words, a `continue` has the same effect as transferring control directly to the end of the block controlled by the `for`.

2.6.7 The do Statement

The **do** statement is the third loop controlling statement in C. It is really just an execute-first **while** statement. It has the form

The do statement

```
do Statement while (ExpressionList);
```

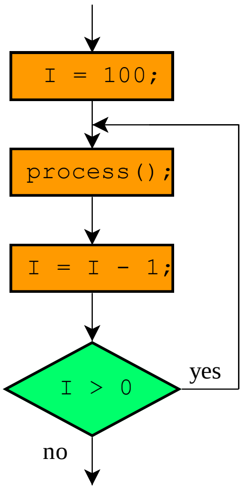
`Statement` is any simple or compound statement. The **do** statement executes in the following steps:

`Statement` is executed.

Then, `ExpressionList` is evaluated and tested. If more than one expression is given, the right most expression yields the value to be tested. If it yields true (non-zero), control goes back to step 1; otherwise, it goes on to whatever follows.

As with the `while` and `for` statements, `break` and `continue` statements may be used. In this case, a `continue` causes control to proceed directly down to the `while` part of the statement for another test of `ExpressionList`. A `break` makes control exit to whatever follows the `do` statement.

```
I=100;
do
{
  process();
  I--;
} while (I > 0);
```



The example of the five-element array could be written as

```
i = 4;
do
{
  array[i] = 0;
  --i;
} while (i >= 0);
i = 4;
```

or as

```
i = 4;
do
  array[i--] = 0;
while (i >= 0);
```

or as

```
i = 5;
do
  array[--i] = 0;
while (i);
```

2.6.8 The `return` Statement

The `return` statement is used within a function to return control to the caller. Return statements are not always required since reaching the end of a function always implies a return. But they are required when it becomes necessary to return from interior points within a function or when a useful value is to be returned to the caller. `return` statements have the form

The `return` statement

```
return ExpressionList?;
```

`ExpressionList?` is an optional list of expressions. If present, the last expression determines the value to be returned by the function. If absent, the returned value is unpredictable.

2.6.9 Null Statements

The simplest C statement is the null statement. It has no text, just a semicolon terminator. As its name implies, it does exactly nothing. Statements that do nothing can serve a purpose. As we saw previously, expressions in C can do work beyond that of simply yielding a value. In fact, in C programs, all of the work is accomplished by expressions; this includes assignments and calls to functions that invoke operating system services such as input/output operations. It follows that anything can be done at any point in the syntax that calls for an expression.

Nulls statements (statements that do nothing) can serve a purpose in a program

Take, for example, the statement

```
while ((UART2_S1 & TDRE) == 0); // Wait for TDRE to be set
```

in which the `((UART2_S1 & TDRE) == 0)` controls the execution of the null statement following. The null statement is just one way in which the C language follows a philosophy of attaching intuitive meanings to seemingly incomplete constructs. The idea is to make the language as general as possible by having the least number of disallowed constructs.

2.6.10 The goto Statement

goto statements break the sequential flow of execution by causing control to jump abruptly to designated points. They have the general form

The goto statement…

```
goto Name;
```

where Name is the name of a label which must appear in the same function. It must also be unique within the function.

```
short data[10];
void clear(void)
{
  short n;
  n = 0;
loop:
  data[n] = 0;
  n++;
  if (n == 10)
    goto done;
  goto loop;
done:
  // Semicolon needed because you can't have a label at the
  // end of a compound statement
  ;
}
```

Listing 2.59 – Example of a goto statement

Notice that labels are terminated with a colon. This highlights the fact that they are not statements but statement prefixes which serve to label points in the logic as targets for goto statements. When control reaches a goto, it proceeds directly from there to the designated label. Both forward and backward references are allowed, but the range of the jump is limited to the body of the function containing the goto statement.

goto statements cannot be used in functions which declare locals in blocks which are subordinate to the outermost block of the function.

…should never be used

Because they violate the structured programming paradigm, goto statements should not be used at all.

2.6.11 Missing Statements

The C language has no input/output, program control, or memory management statements. In the interest of portability these services have been relegated to a set of standard functions in the run-time library. Since they depend so heavily on the run-time environment, removing them from the language eliminates a major source of compatibility problems. Each implementation of C has its own library of standard functions that perform these operations. Since different compilers have libraries that are pretty much functionally equivalent, programs have very few problems when they are compiled by different compilers.

High-level functions like input / output are implemented in C libraries – this makes C very portable

2.7 Pointers

The ability to work with memory addresses is an important feature of the C language. This feature allows programmers the freedom to perform operations similar to assembly language. Unfortunately, along with the power comes the potential danger of hard-to-find and serious run-time errors. In many situations, array elements can be reached more efficiently through pointers than by subscripting. It also allows pointers and pointer chains to be used in data structures. Without pointers the run-time dynamic memory allocation and deallocation using the heap would not be possible. We will also use a format similar to pointers to develop mechanisms for accessing I/O ports. These added degrees of flexibility are absolutely essential for embedded systems.

2.7.1 Addresses and Pointers

Pointers are variables that store addresses

Addresses that can be stored and changed are called pointers. A pointer is really just a variable that contains an address. Although they can be used to reach objects in memory, their greatest advantage lies in their ability to enter into arithmetic (and other) operations, and to be changed. Just like other variables, pointers have a type. In other words, the compiler knows the format (8-bit, 16-bit, 32-bit, unsigned, signed) of the data pointed to by the address.

Not all addresses are pointers

Not every address is a pointer. For instance, we can write `&var` when we want the address of the variable `var`. The result will be an address that is not a pointer since it does not have a name or a place in memory. It cannot, therefore, have its value altered.

Other examples include an array or a structure name. As we shall see in the next sections, an unsubscripted array name yields the address of the array, and a structure name yields the address of the structure. But since arrays and structures cannot be moved around in memory, their addresses are not variable. So, although such addresses have a name, they do not exist as objects in memory (the array does, but its address does not) and cannot, therefore, be changed.

A third example is a character string. A character string yields the address of the character array specified by the string. In this case the address has neither a name or a place in memory, so it too is not a pointer.

2.7.2 Pointer Declarations

The syntax for declaring pointers is like that for variables except that pointers are distinguished by an asterisk that prefixes their names. Listing 2.60 illustrates several legitimate pointer declarations. Notice, in the third example, that we may mix pointers and variables in a single declaration, i.e. the variable `data` and the pointer `pt3` are declared in the same statement. Also notice that the data type of a pointer declaration specifies the type of object to which the pointer refers, not the type of the pointer itself. As we shall see, MX creates pointers containing 32-bit unsigned absolute addresses.

Pointers are declared by placing a `*` in front of the pointer name

```
// define pt1, declare as a pointer to a 16-bit integer
short *pt1;

// define pt2, declare as a pointer to an 8-bit character
char *pt2;

// define data and pt3, declare data as an unsigned 16-bit integer
// and declare pt3 as a pointer to a 16-bit unsigned integer
unsigned short data, *pt3;

// define pt4, declare as a pointer to a 32-bit integer
long *pt4;

// declare pt5 as a pointer to an integer
extern short *pt5;
```

Listing 2.60 – Examples of pointer declarations

The best way to think of the asterisk is to imagine that it stands for the phrase "object at" or "object pointed to by". The first declaration in Listing 2.60 then reads "the object at (pointed to by) `pt1` is a 16-bit signed integer".

2.7.3 Pointer Referencing

We can use the pointer to retrieve data from memory or to store data into memory. Both operations are classified as pointer references. The syntax for using pointers is like that for variables except that pointers are distinguished by an asterisk that prefixes their names. Figure 2.21 to Figure 2.33 illustrate several legitimate pointer references. In the first figure, the global variables contain unknown data (actually we know MX will zero global variables). The arrow identifies the execution location. Assume addresses `0x1FFF007C` through `0x1FFF0094` exist in RAM.

Example of legitimate pointer references in C

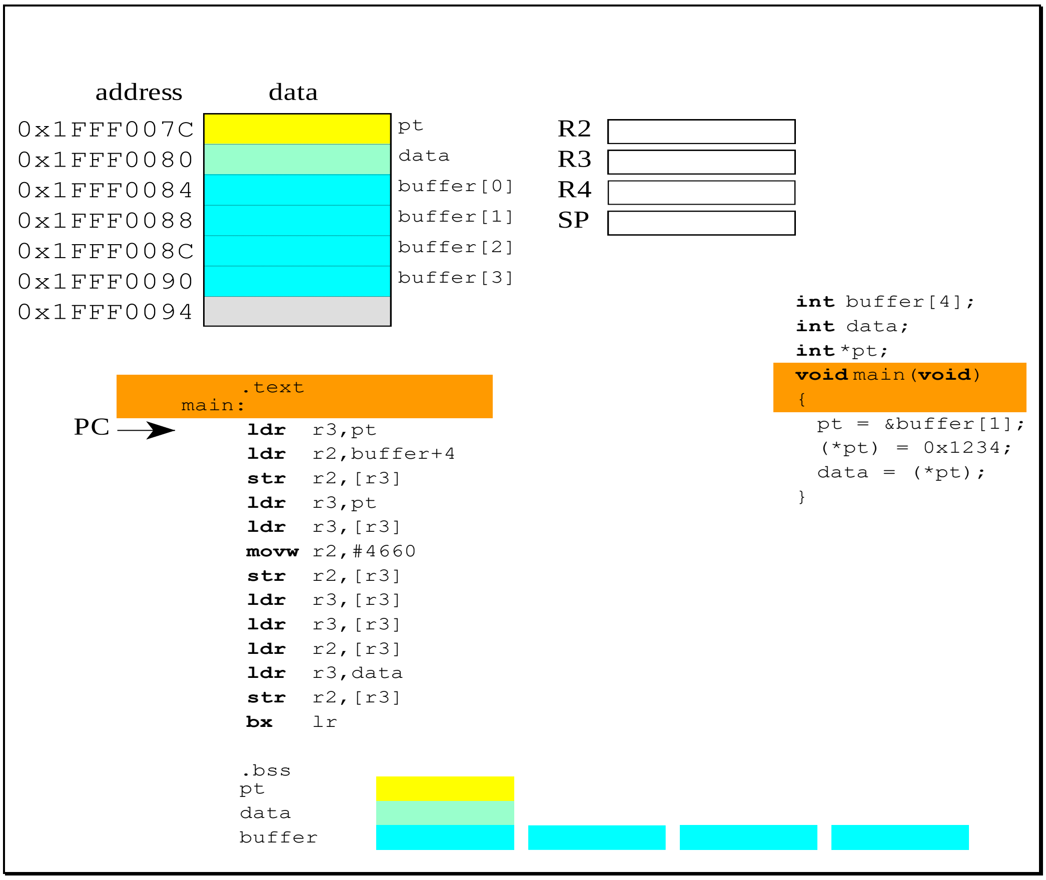


Figure 2.21 – K64 example of pointer references – step 1

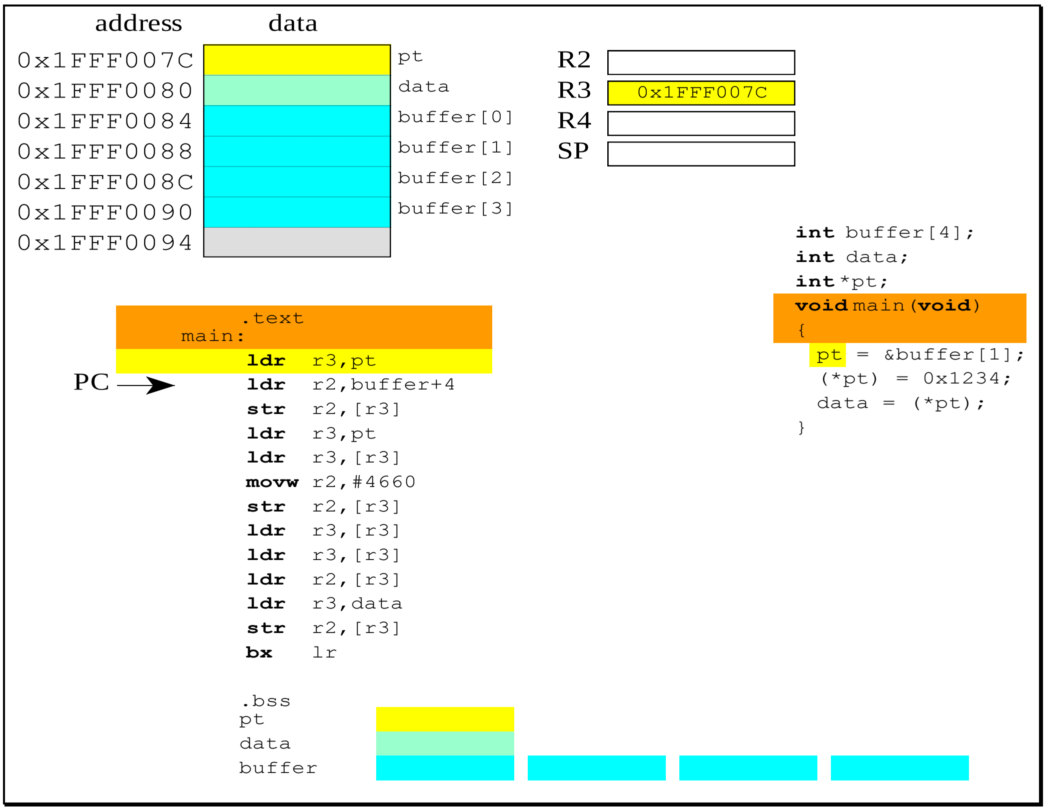


Figure 2.22 – K64 example of pointer references – step 2

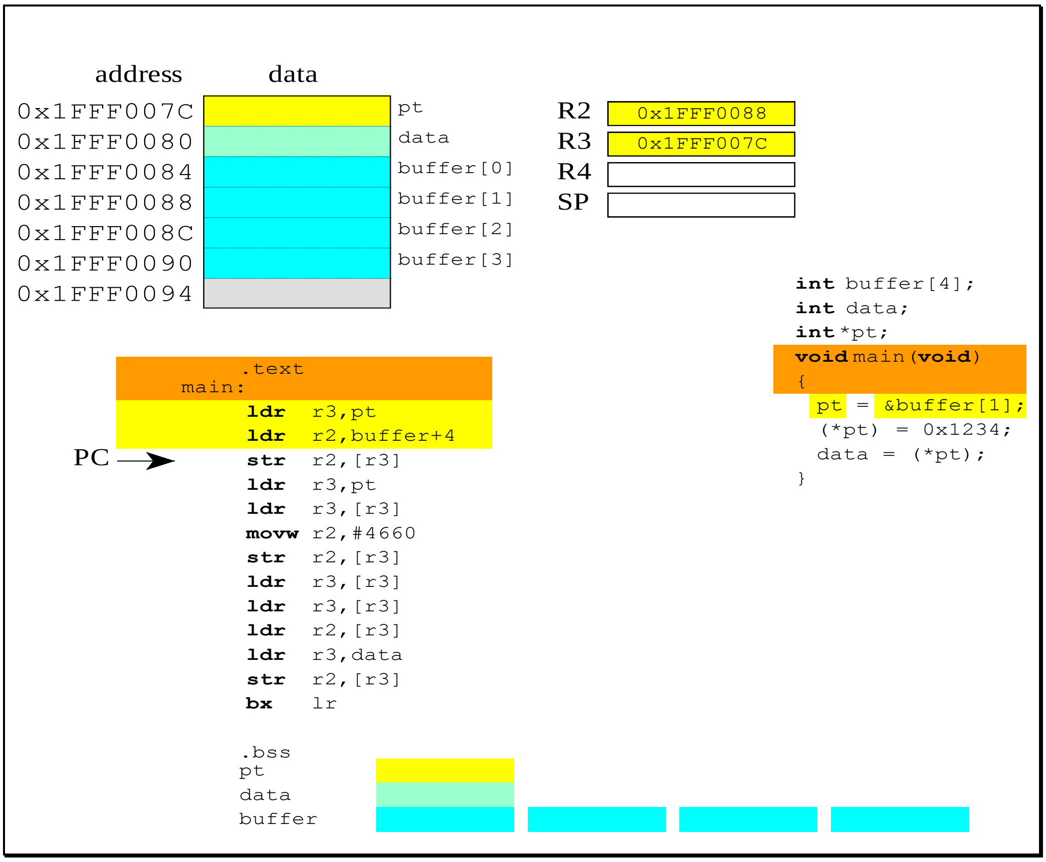


Figure 2.23 – K64 example of pointer references – step 3

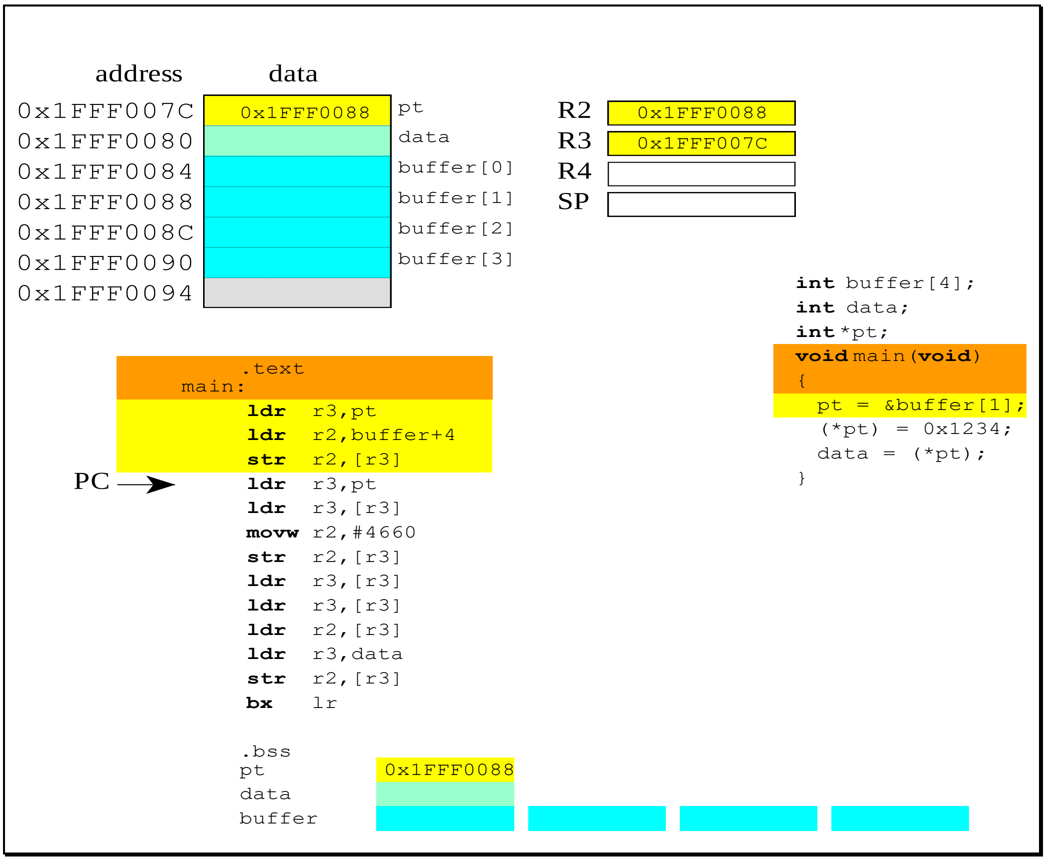


Figure 2.24 – K64 example of pointer references – step 4

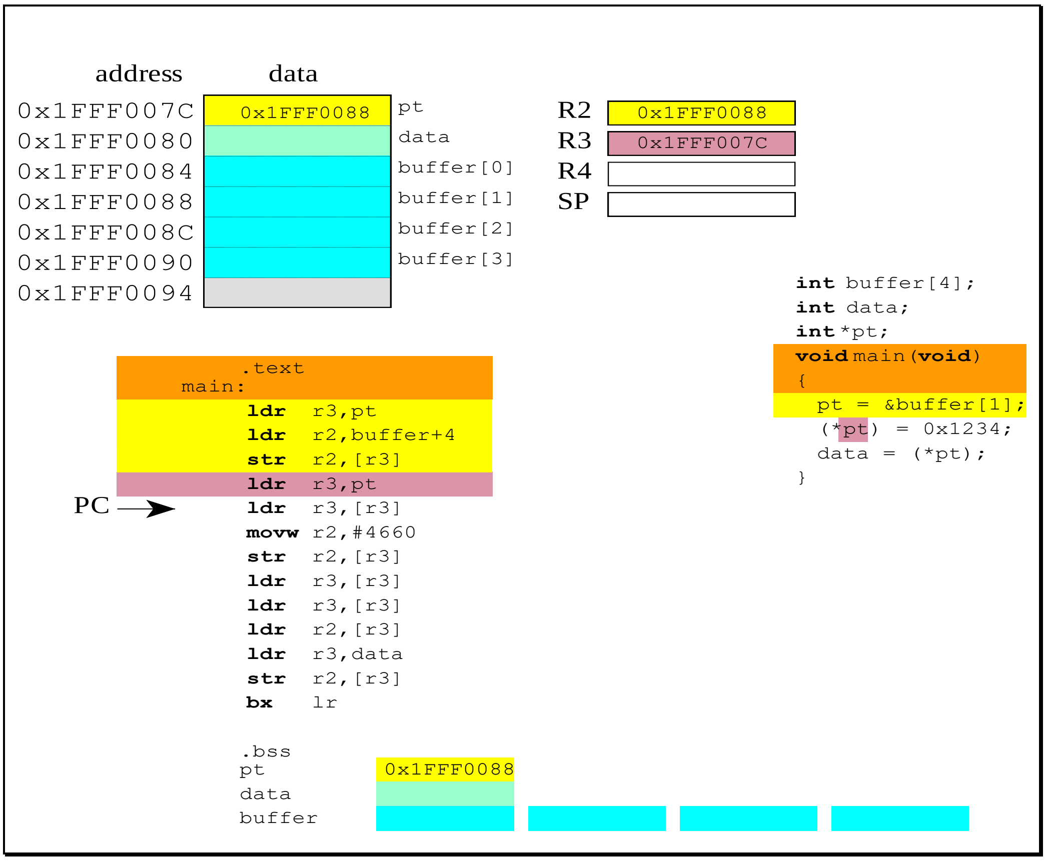


Figure 2.25 – K64 example of pointer references – step 5

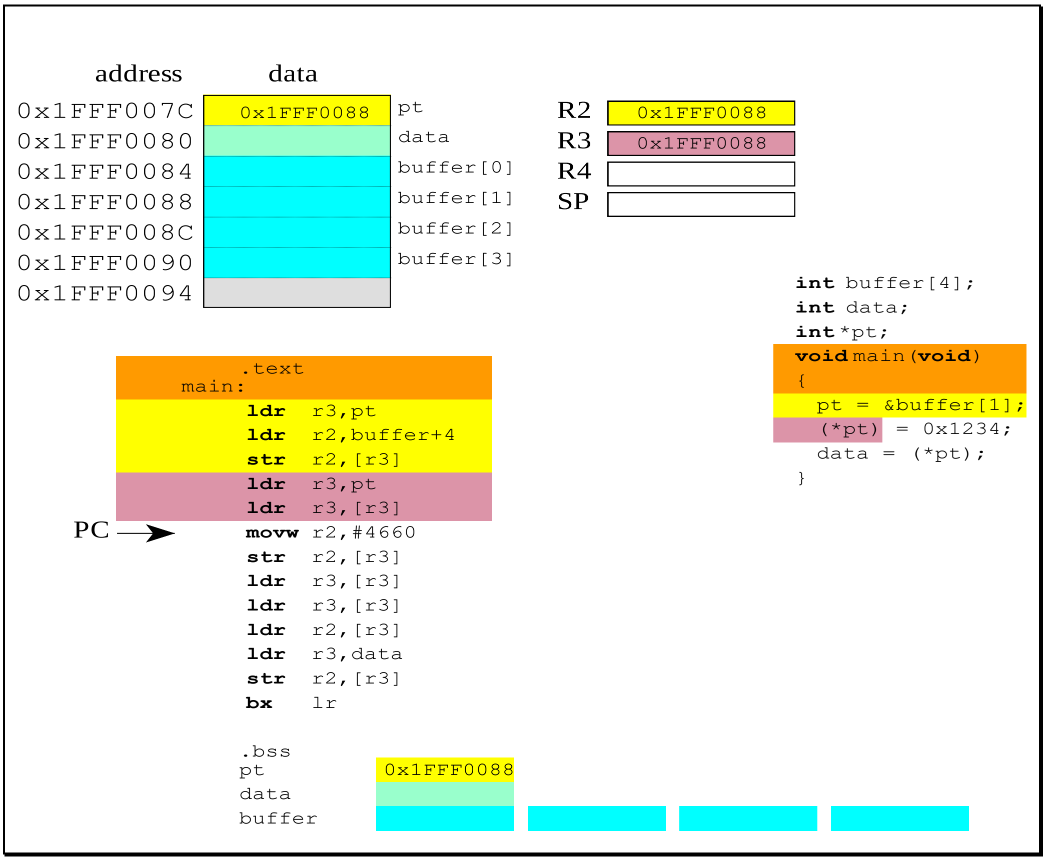


Figure 2.26 – K64 example of pointer references – step 6

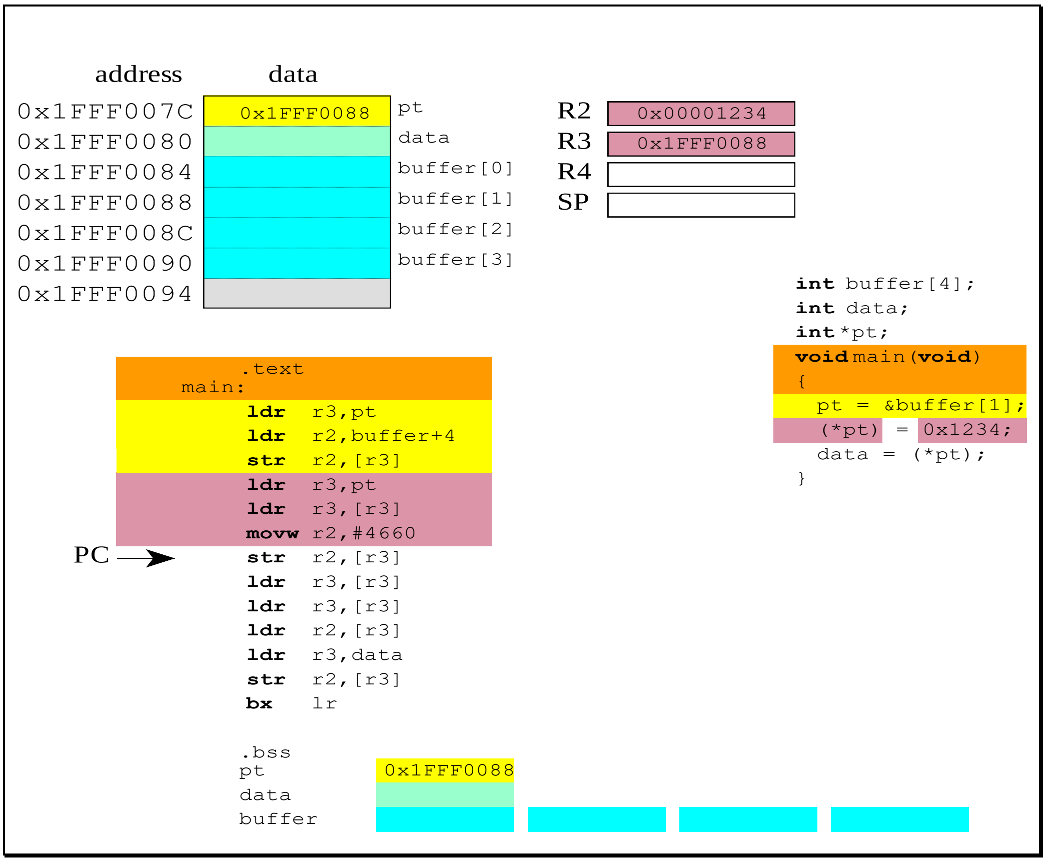


Figure 2.27 – K64 example of pointer references – step 7

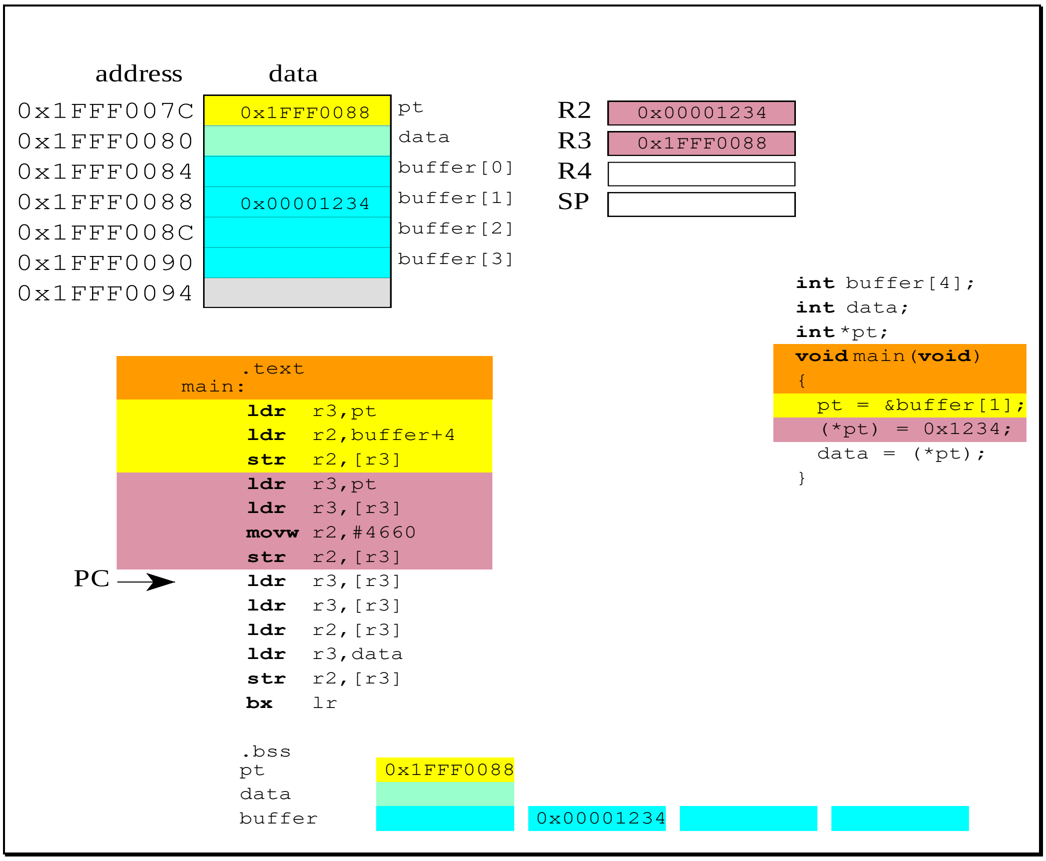


Figure 2.28 – K64 example of pointer references – step 8

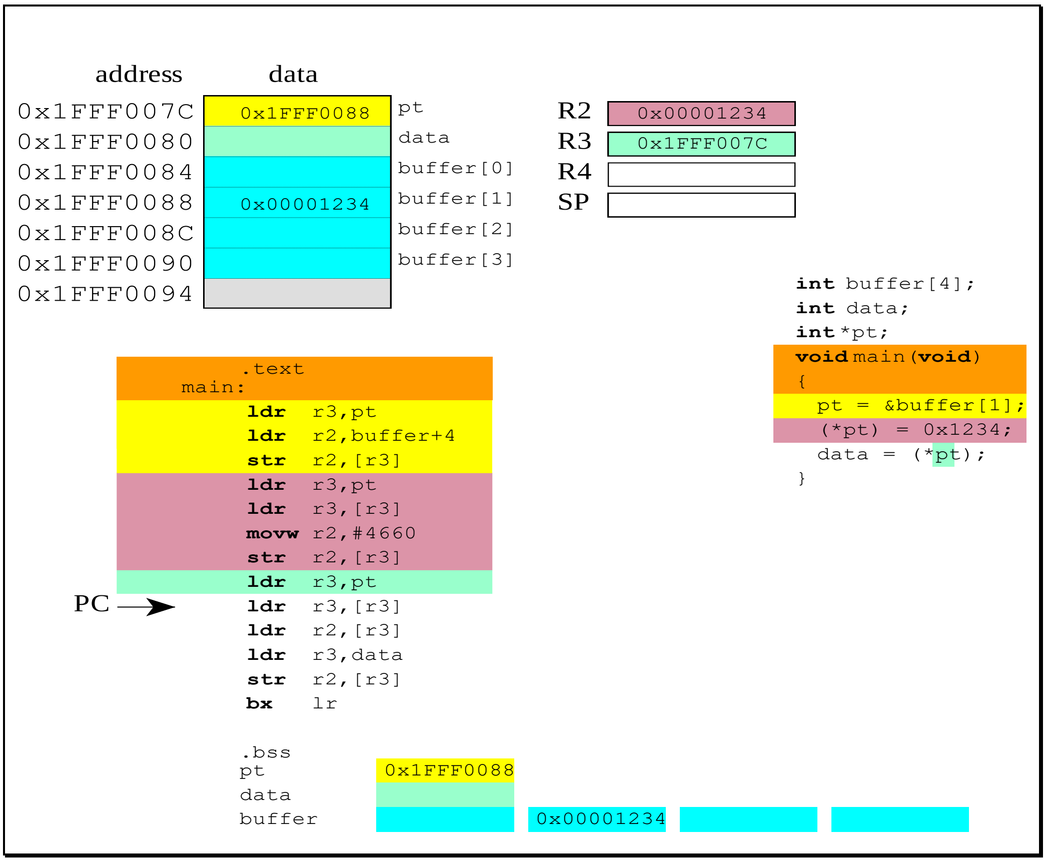


Figure 2.29 – K64 example of pointer references – step 9

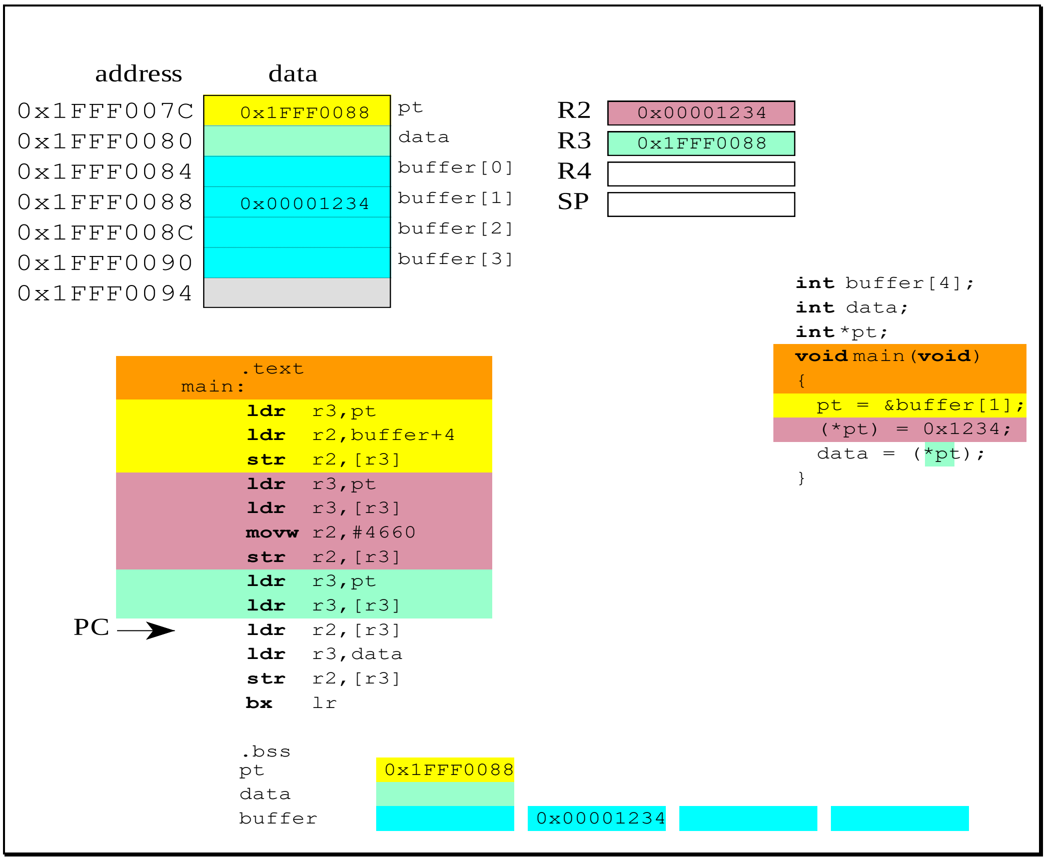


Figure 2.30 – K64 example of pointer references – step 10

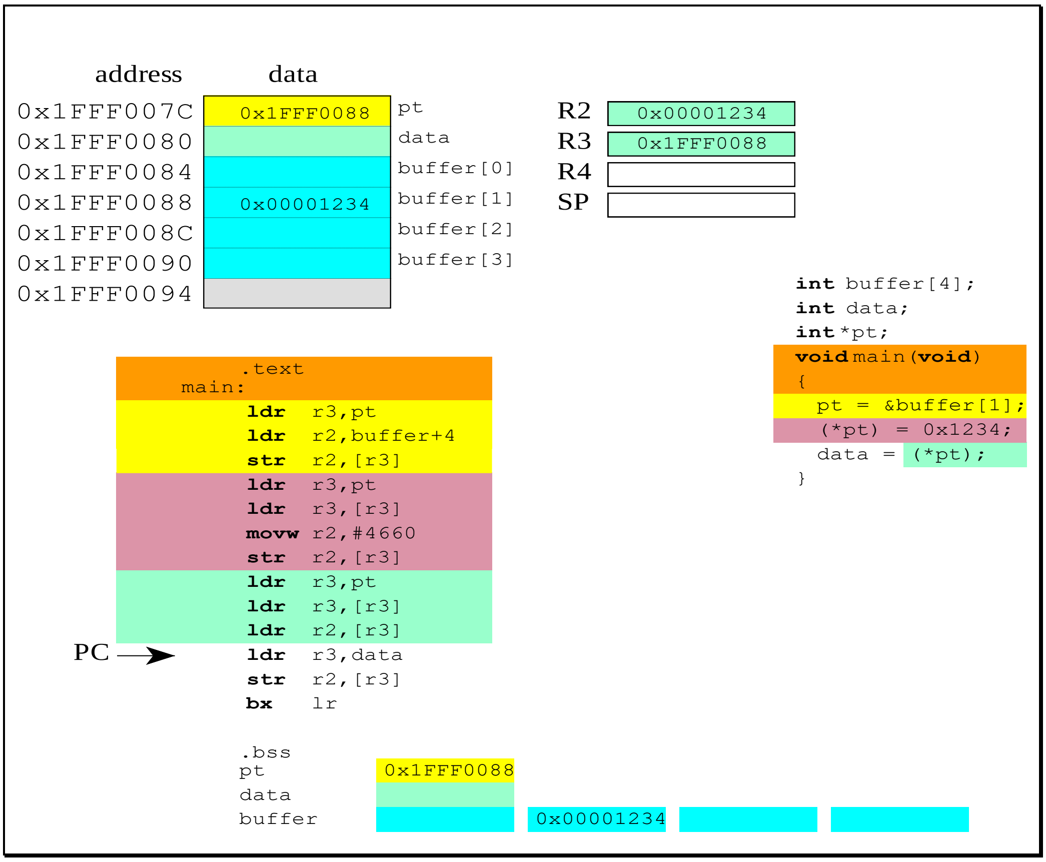


Figure 2.31 – K64 example of pointer references – step 11

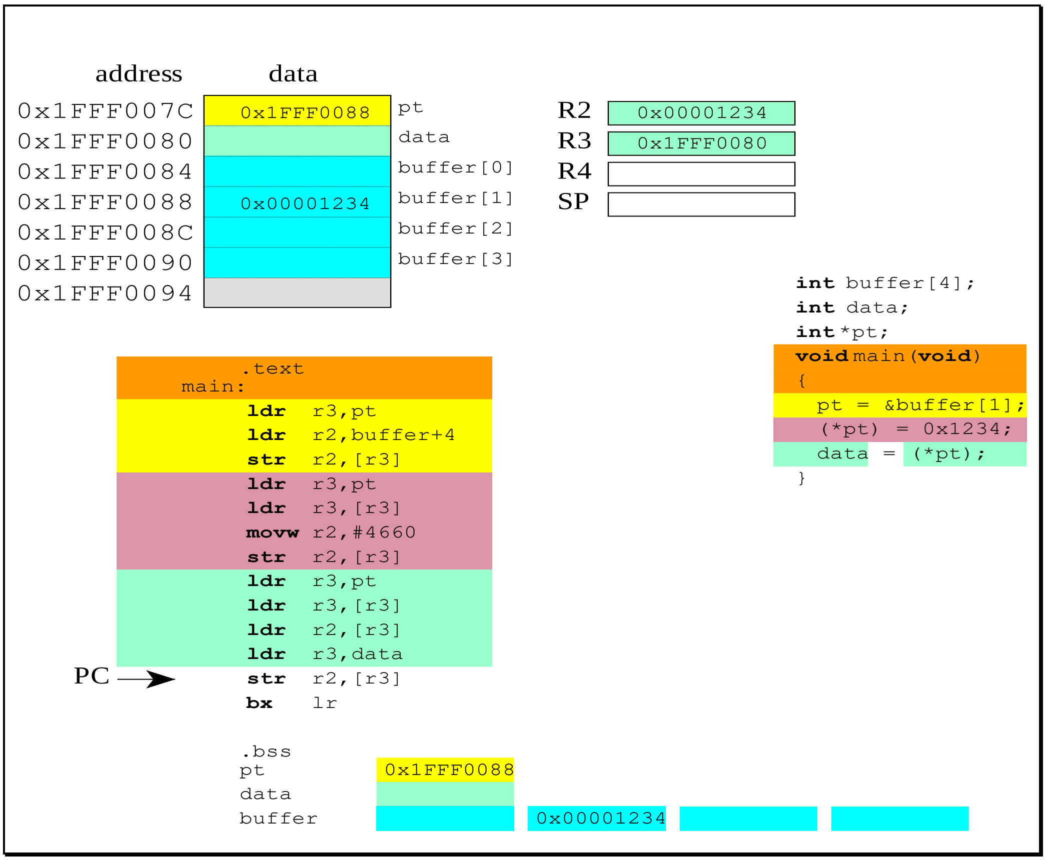


Figure 2.32 – K64 example of pointer references – step 12

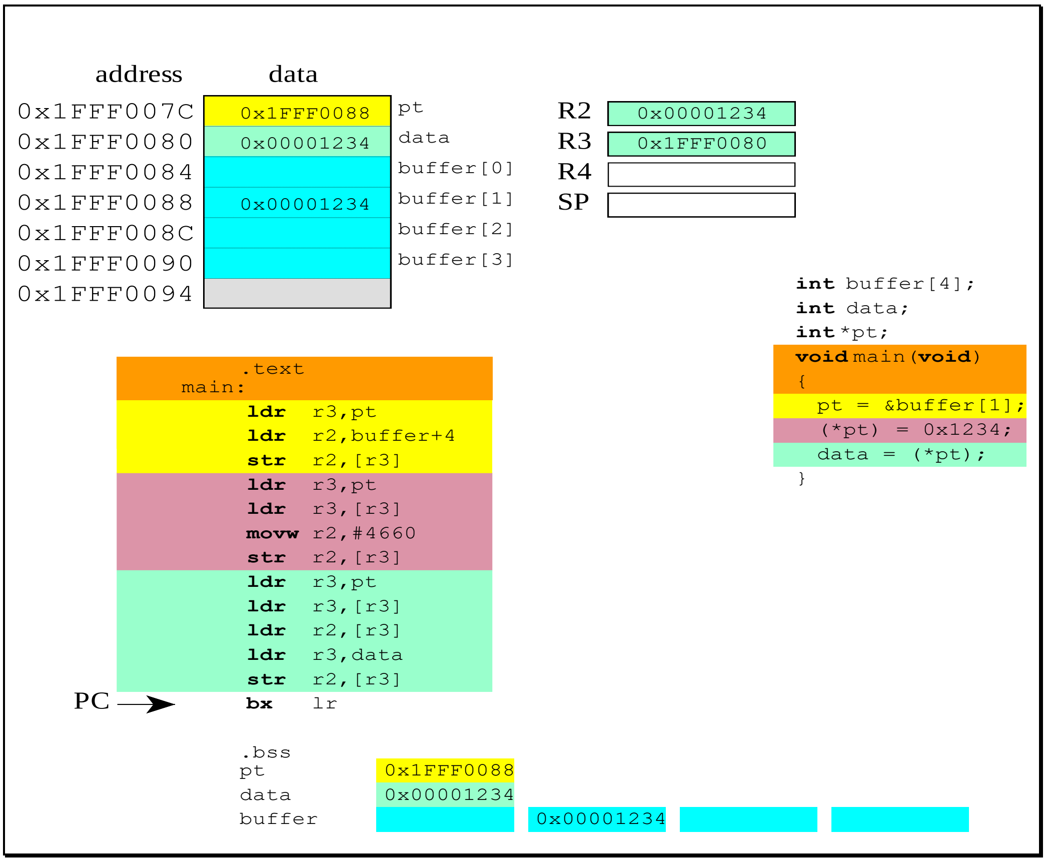


Figure 2.33 – K64 example of pointer references – step 13

The expression `&buffer[1]` returns the address of the second 32-bit element of the buffer (`0x1FFF0088`). Therefore the line `pt=&buffer[1];` makes `pt` point to `buffer[1]`.

When the `*pt` occurs on the left-hand-side of an assignment statement data is stored into memory at the address. Recall the `*pt` means "the 32-bit signed integer at `0x1FFF0088`". You can optionally add the parentheses `()` to clarify that `*` and `pt` are one object. In this case the parentheses are not needed. Later when we perform address arithmetic, the parentheses will be important. Therefore the line `(*pt) = 0x1234;` sets `buffer[1]` to `0x1234`.

When the `*pt` occurs on the right-hand-side of an assignment statement, data is retrieved from memory at the address. Again, you can optionally add the parentheses `()` to clarify that `*` and `pt` are one object. Therefore the line `data = (*pt);` sets data to `0x1234` (more precisely, it copies the 32-bit information from `buffer[1]` into `data`).

We can get a better understanding of pointers by observing the assembly generated by our compiler. The following K64 assembly was generated by MX when the above pointer example (Figure 2.31) was compiled. Notice that the K64 uses a RISC CPU and it can't directly operate on memory addresses – it can only load register contents from memory (`ldr`) and store register contents into memory (`str`). The notation `[r3]` is the assembly language equivalent of a dereference and means use the contents of `r3` as an address of the actual operand.

```
main:
    ldr   r3,pt
    ldr   r2,buffer+4
    str   r2,[r3]
    ldr   r3,[r3]
    movw  r2,#4660
    str   r2,[r3]
    ldr   r3,pt
    ldr   r3,[r3]
    ldr   r2,[r3]
    ldr   r3,data
    str   r2,[r3]
    bx    lr
```

Listing 2.61 – Examples of pointer references created by MX

2.7.4 Memory Addressing

With a 32-bit CPU, addressing is "flat" and occurs with 32-bit addresses

The size of a pointer depends on the architecture of the CPU and the implementation of the C compiler. The K64 employs an absolute memory addressing scheme in which an effective address is composed simply of a single 32-bit unsigned value.

Types of memory in the K64

Most embedded systems employ a segmented memory architecture. From a physical standpoint we might have a mixture of regular RAM, battery-backed-up RAM, regular EEPROM, Flash EPROM, regular PROM, one-time-programmable PROM and ROM. RAM is the only memory structure that allows the program both read and write access. The other types are usually loaded with object code from our `.elf` file and our program is allowed only to read the data. Table 2.40 shows the various types of memory available in the K64 microcontroller. The RAM contains temporary information that is lost when the power is shut off. This means that all variables allocated in RAM must be explicitly initialized at run time by the software. If the embedded system includes a separate battery for the RAM, then information is not lost when the main power is removed. EEPROM is a technology that allows individual small sectors (typically 4 KiB) to be erased and bytes individually written. Most microcontrollers now have non-volatile Flash ROM as the main program memory, which has bulk erasure (typically 4 KiB) and individual write capability at the byte level. The one-time-programmable (OTP) ROM is a simple non-volatile storage technology used in large volume products that can be programmed only once by the semiconductor manufacturer.

Memory	When power is removed	Ability to Read/Write	Program cycles (typical)
RAM	volatile	Random and fast access	infinite
Battery-backed RAM	non-volatile	Random and fast access	infinite
EEPROM	non-volatile	Easily reprogrammed	25 million
Flash	non-volatile	Easily reprogrammed	50 000
OTPROM	non-volatile	Can be programmed once at the factory	N/A

Table 2.40 – Various types of memory available for the K64

From a logical standpoint we implement segmentation when we group together in memory information that has similar properties or usage. Typical software segments include global variables (`.data` section), the heap, local variables, fixed constants (`.rodata` section), and machine instructions (`.text` section). Global variables are permanently allocated and usually accessible by more than one program. We must use global variables for information that must be permanently available, or for information that is to be shared by more than one module. We will see the first-in-first-out (FIFO) queue is a global data structure that is shared by more than one module. MX allows the use of a heap to dynamically allocate and release memory. This information can be shared or not shared depending on which modules have pointers to the data. The heap is efficient in situations where storage is needed for only a limited amount of time. Local variables are usually allocated on the stack at the beginning of the function, used within the function, and deallocated at the end of the function. Local variables are not shared with other modules. Fixed constants do not change and include information such as numbers, strings, sounds and pictures. Just like the heap, the fixed constants can be shared or not shared depending on which modules have pointers to the data.

The type of memory dictates its usage

In an embedded application, we usually put global variables, the heap, and local variables in RAM because these types of information can change during execution. When software is to be executed on a regular computer, the machine instructions are usually read from a mass storage device (like a disk) and loaded into memory. Because the embedded system usually has no mass storage device, the machine instructions and fixed constants must be stored in non-volatile memory. If there is both EEPROM and Flash on our microcontroller, we put some fixed constants in EEPROM and some in Flash. If it is information that we may wish to change in the future, we could put it in EEPROM. Examples include language-specific strings, calibration constants, finite state machines, and system ID numbers. This allows us to make minor modifications to the system by reprogramming the EEPROM without throwing the chip away. For a project with a large volume it will be cost effective to place the machine instructions in OTPROM.

2.7.5 Pointer Arithmetic

A major difference between addresses and ordinary variables or constants has to do with the interpretation of addresses. Since an address points to an object of some particular type, adding one (for instance) to an address should direct it to the next object, not necessarily the next byte. If the address points to integers, then it should end up pointing to the next integer. But, since integers occupy four bytes, adding one to an integer address must actually increase the address by four. Likewise, if the address points to short integers, then adding one to an address should end up pointing to the next short integer by increasing the address by two. A similar consideration applies to subtraction. In other words, values added to or subtracted from an address must be scaled according to the size of the objects being addressed. This is done automatically by the compiler, and saves the programmer a lot of thought and makes programs less complex since the scaling need not be coded explicitly. The scaling factor for integers is four; the scaling factor for short integers is two; the scaling factor for characters is one. Therefore, character addresses do not receive special handling. It should be obvious that when we define structures of other sizes, the appropriate factors would have to be used.

Pointer arithmetic takes into account the size of the data being pointed to

A related consideration arises when we imagine the meaning of the difference of two addresses. Such a result is interpreted as the number of objects between the two addresses. If the objects are integers, the result must be divided by four in order to yield a value which is consistent with this meaning. See the next section for more information on address arithmetic.

When an address is operated on, the result is always another address of the same type. Thus, if `ptr` is a signed 32-bit integer pointer, then `ptr+1` also points to a signed 32-bit integer.

Type is preserved in pointer arithmetic

Precedence determines the order of evaluation. One of the most common mistakes results when the programmer neglects the fact the `*` used as a unary pointer reference has precedence over all binary operators. This means the expression `*ptr + 1` is the same as `(*ptr) + 1` and not `*(ptr + 1)`. Remember (2.2): "When confused about precedence (and aren't we all) add parentheses to clarify the expression."

2.7.6 Pointer Comparisons

Pointers are always unsigned

One major difference between pointers and other variables is that pointers are always considered to be unsigned. This should be obvious since memory addresses are not signed. This property of pointers (actually all addresses) ensures that only unsigned operations will be performed on them. It further means that the other operand in a binary operation will also be regarded as unsigned (whether or not it actually is). In the following example, `pt1` and `pt2[5]` return the current values of the addresses. For instance, if the array `pt2[]` contains addresses, then it would make sense to write:

```
short *pt1;      // define 16-bit integer pointer
short *pt2[10];  // define ten 16-bit integer pointers
short done(void)
{
  // returns true if pt1 is higher than pt2[5]
  if (pt1 > pt2[5])
    return (1);
  return (0);
}
```

Listing 2.62 – Example showing a pointer comparison

which performs an unsigned comparison since `pt1` and `pt2` are pointers. Thus, if `pt2[5]` contains `0x1FFFF000` and `pt1` contains `0x1FFF1000`, the expression will yield false, since `0x1FFFF000` is a higher unsigned value than `0x1FFF1000`.

The address of zero is reserved for `NULL` – a pointer that doesn't yet point to anything

It makes no sense to compare a pointer to anything but another address or zero. C guarantees that valid addresses can never be zero, so that particular value is useful in representing the absence of an address in a pointer.

Furthermore, to avoid portability problems, only addresses within a single array should be compared for relative value (e.g., which pointer is larger). To do otherwise would necessarily involve assumptions about how the compiler organizes memory. Comparisons for equality, however, need not observe this restriction, since they make no assumption about the relative positions of objects. For example if `pt1` points into one data array and `pt2` points into a different array, then comparing `pt1` to `pt2` would be meaningless. Which pointer is larger would depend on where in memory the two arrays were assigned.

2.7.7 A FIFO Queue Example

To illustrate the use of pointers we will design a two-pointer FIFO. The first-in first-out circular queue (FIFO) is also useful for data flow problems. It is a very common data structure used for I/O interfacing. The order preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFOs studied in this section will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be shared by more than one program. The advantage of using a FIFO structure for a data flow problem is that we can decouple the source and sink processes. Without the FIFO we would have to produce 1 piece of data, then process it, produce another piece of data, then process it. With the FIFO, the source process can continue to produce data without having to wait for the sink to finish processing the previous data. This decoupling can significantly improve system performance.

`GetPt` points to the data that will be removed by the next call to **`FIFO_Get`**`()`, and `PutPt` points to the empty space where the data will be stored by the next call to **`FIFO_Put`**`()`. If the FIFO is full when **`FIFO_Put`**`()` is called then the subroutine should return a full error. Similarly, if the FIFO is empty when **`FIFO_Get`**`()` is called, then the subroutine should return an empty error. The `PutPt` and `GetPt` pointers must be wrapped back up to the top when they reach the bottom.

Four **`FIFO_Put()`** operations…

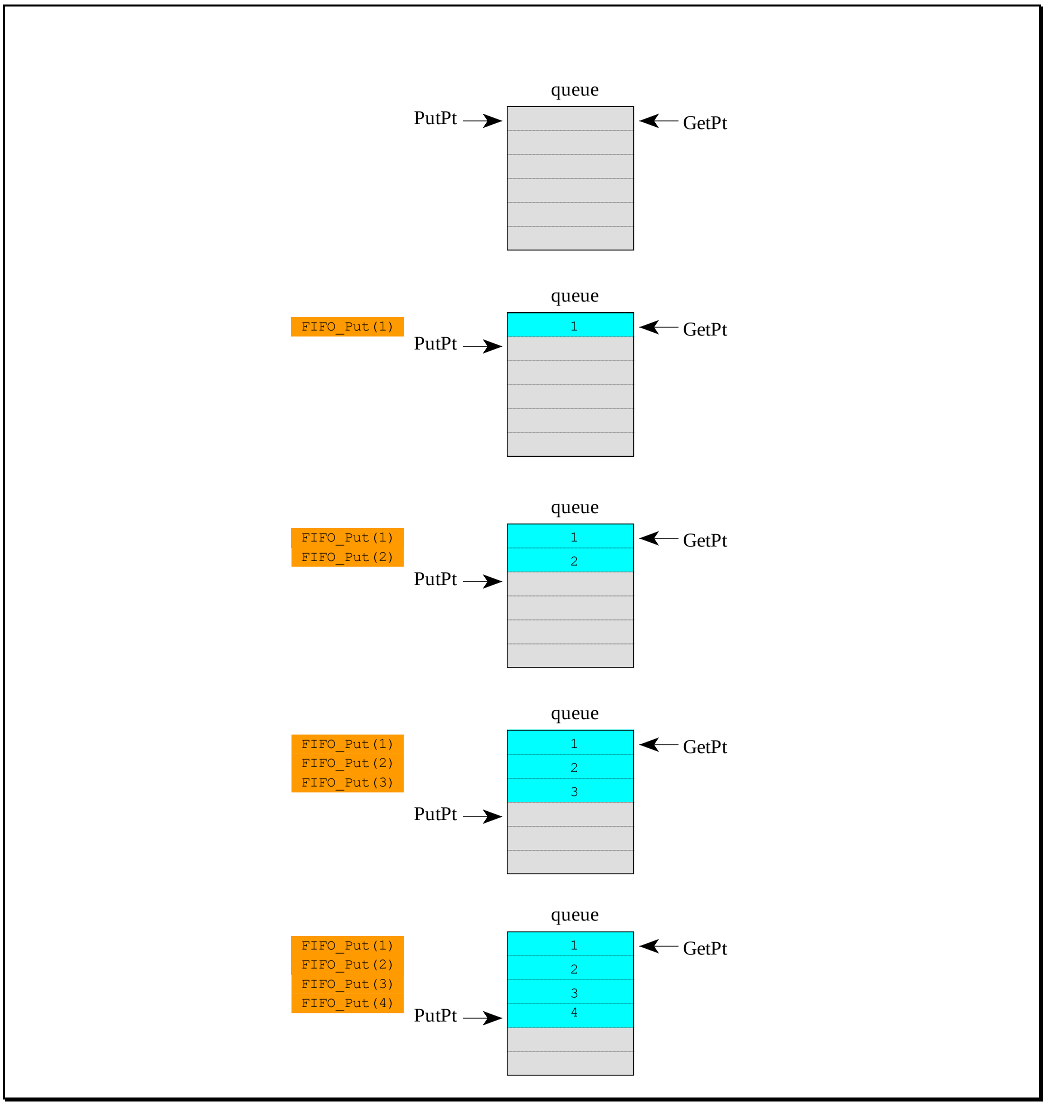


Figure 2.34 – FIFO example showing the wrapping of pointers – step 1

Two **`FIFO_Get()`** operations…

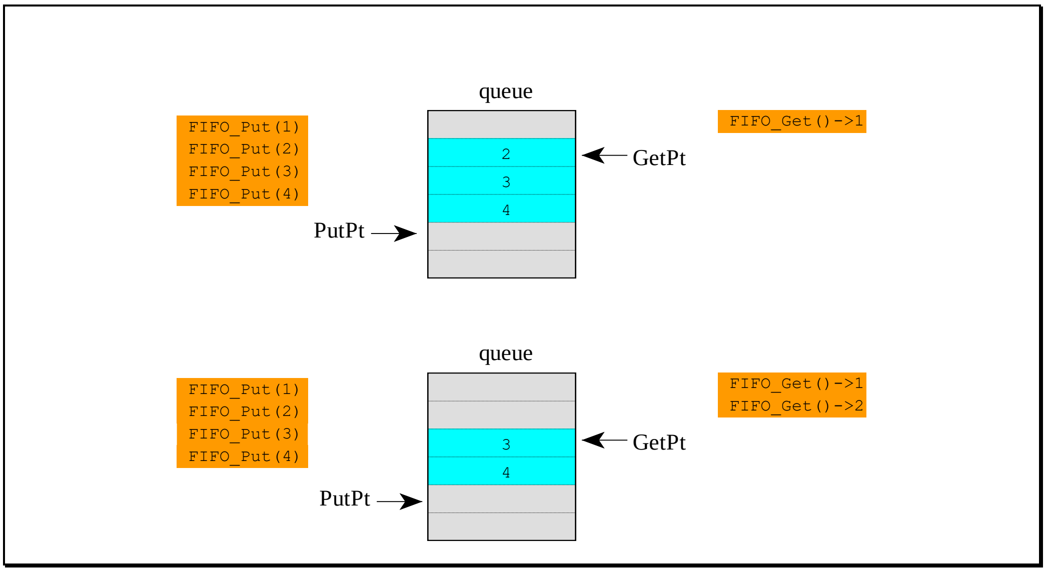


Figure 2.35 – FIFO example showing the wrapping of pointers – step 2

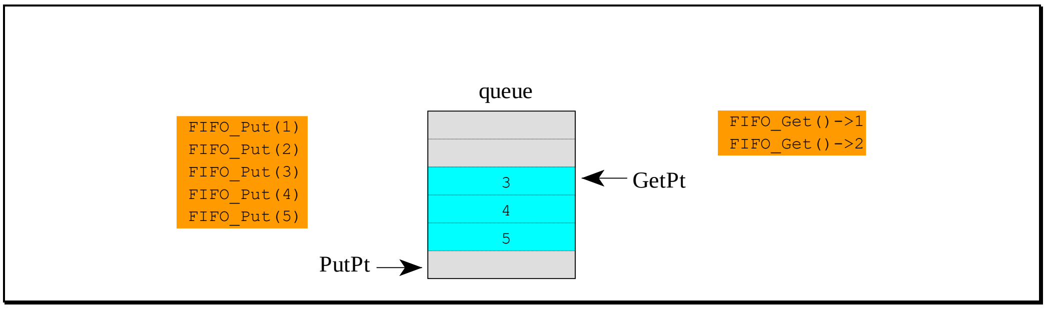


Figure 2.36 – FIFO example showing the wrapping of pointers – step 3

A **FIFO_Put()** followed by a **FIFO_Get()**

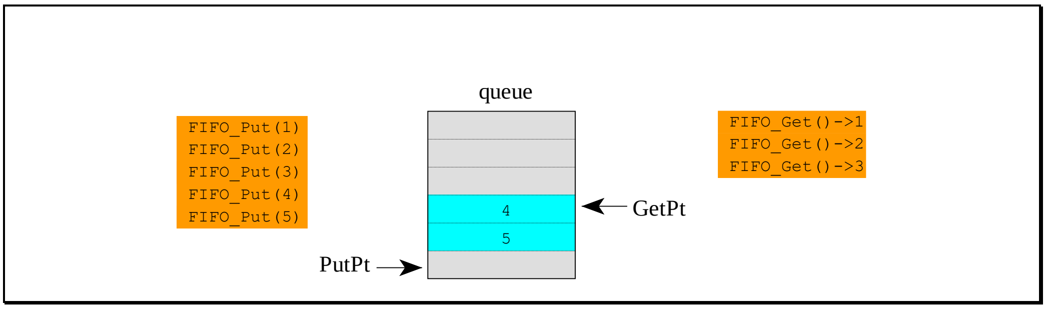


Figure 2.37 – FIFO example showing the wrapping of pointers – step 4

Two **FIFO_Put()** operations…

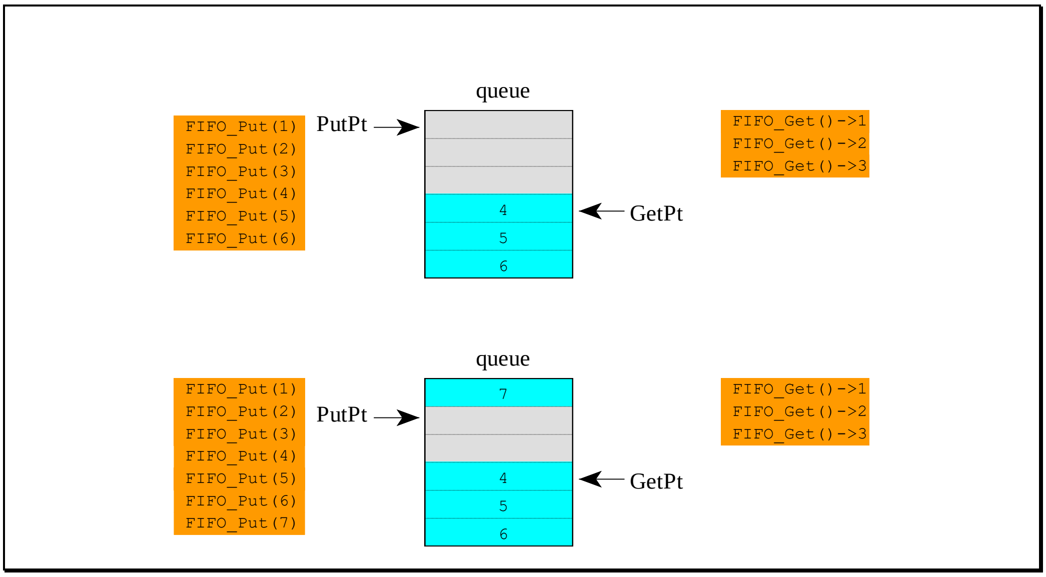


Figure 2.38 – FIFO example showing the wrapping of pointers – step 5

Two **FIFO_Get()** operations…

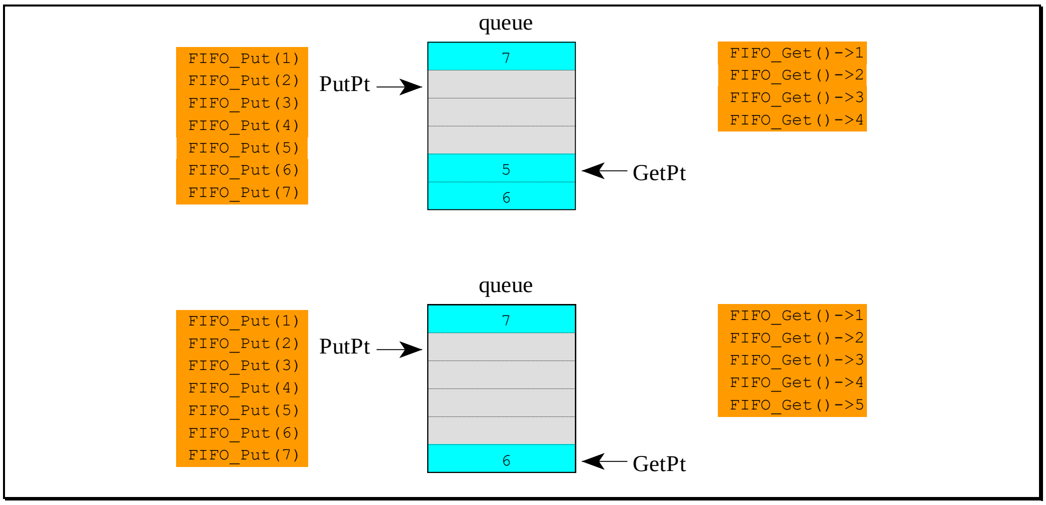


Figure 2.39 – FIFO example showing the wrapping of pointers – step 6

Two **FIFO_Put()** operations…

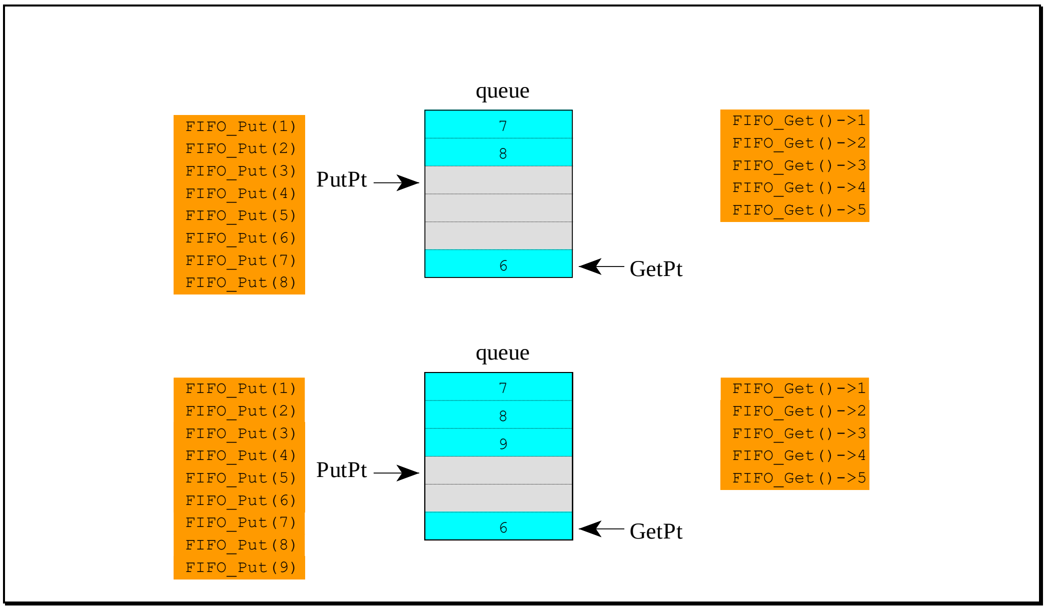
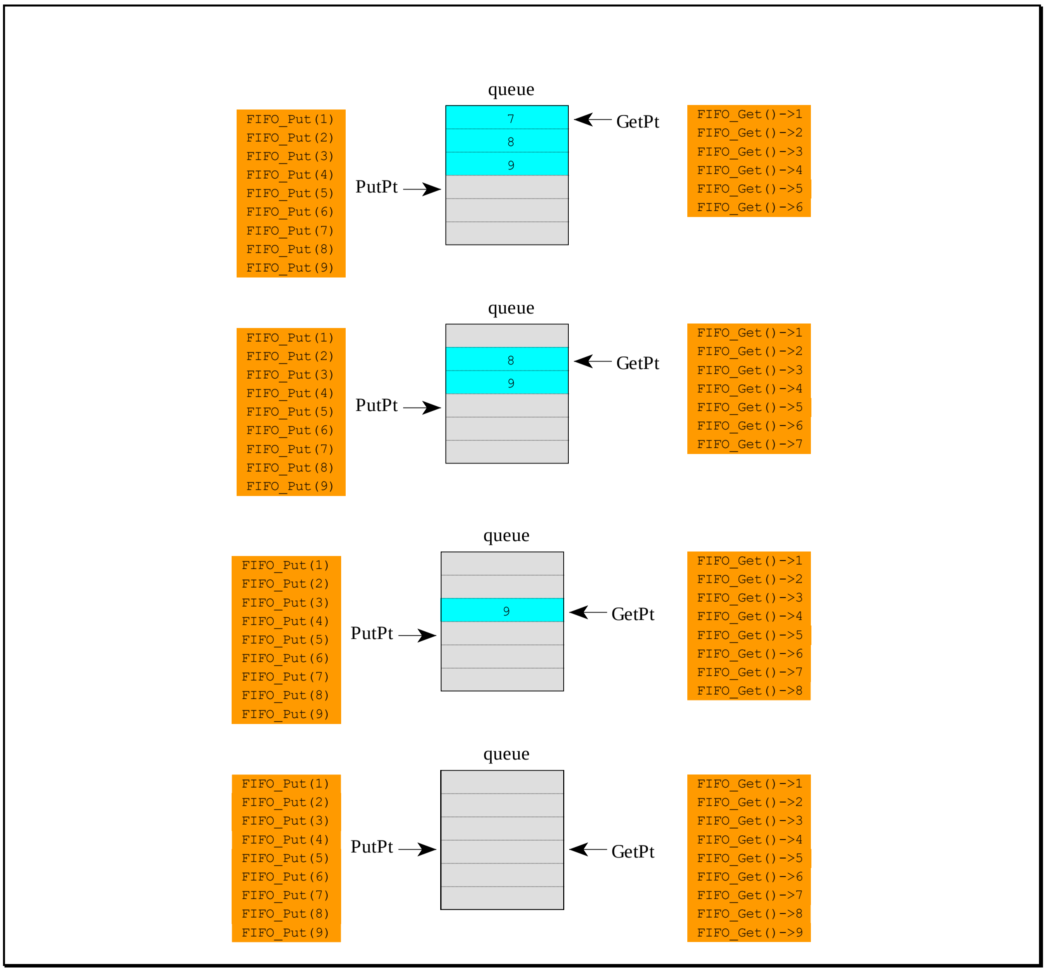


Figure 2.40 – FIFO example showing the wrapping of pointers – step 7



Finally, four **FIFO_Get()** operations that empty the queue

Figure 2.41 – FIFO example showing the wrapping of pointers – step 8

There are two mechanisms to determine whether the FIFO is empty or full. A simple method is to implement a counter containing the number of bytes currently stored in the FIFO. **`FIFO_Get()`** would decrement the counter and **`FIFO_Put()`** would increment the counter. The second method is to prevent the FIFO from being completely full. For example, if the FIFO had 100 bytes allocated, then the **`FIFO_Put()`** subroutine would allow a maximum of 99 bytes to be stored. If there were already 99 bytes in the FIFO and another PUT were called, then the FIFO would not be modified and a full error would be returned. In this way if `PutPt` equals `GetPt` at the beginning of **`FIFO_Get()`**, then the FIFO is empty. Similarly, if `PutPt + 1` equals `GetPt` at the beginning of **`FIFO_Put()`**, then the FIFO is full. Be careful to wrap the `PutPt + 1` before comparing it to `GetPt`. This second method does not require the length to be stored or calculated.

```c
// Pointer implementation of the FIFO
#define FIFO_SIZE 10 // Max number of 8-bit data in the FIFO

char* PutPt;    // Pointer of where to put next
char* GetPt;    // Pointer of where to get next
// FIFO is empty if PutPt == GetPt
// FIFO is full if PutPt + 1 == GetPt

char FIFO[FIFO_SIZE]; // The statically allocated FIFO data

void FIFO_Init(void)
{
  // Make atomic, entering critical section
  EnterCritical();
  PutPt = GetPt = FIFO; // Empty when PutPt == GetPt
  ExitCritical();  // End critical section
}

int FIFO_Put(const char data)
{
  char* pt;          // Temporary put pointer

  // Make atomic, entering critical section
  EnterCritical();
  pt = PutPt + 1;    // Make new potential PutPt
  if (pt == &FIFO[FIFO_SIZE])
    pt = FIFO;       // Wrap pointer if necessary
  if (pt == GetPt)   // If FIFO is full, fail
  {
    ExitCritical();  // End critical section
    return 0;
  }   // Failed, FIFO was full
  else
  {
    *PutPt = data;      // Put data into FIFO
    PutPt = pt;      // Update PutPt
    ExitCritical();  // End critical section
    return 1;       // Successful
  }
}

int FIFO_Get(char* const dataPt)
{
  if (PutPt == GetPt) // Empty if PutPt == GetPt
    return 0;

  // Make atomic, entering critical section
  EnterCritcial();
  *dataPt = *GetPt++;
  if (GetPt == &FIFO[FIFO_SIZE])
    GetPt = FIFO;
  ExitCrtical();    // End critical section
  return 1;
}
```

Listing 2.63 – FIFO queue implemented with pointers

The `EnterCritcial()` macro is defined to save the state of the global interrupt enable bit and disable interrupts. This prevents another thread from interfering with the FIFO operation. The `ExitCrtical()` macro restores the state of the global interrupt enable bit.

Since these routines have read / modify / write accesses to global variables the three functions (**`FIFO_Init`**`()`, **`FIFO_Put`**`()`, **`FIFO_Get`**`()`) are themselves not re-entrant. Consequently interrupts are temporarily disabled, to prevent one thread from re-entering these FIFO functions. One advantage of this pointer implementation is that if you have a single thread that calls **`FIFO_Get`**`()` (e.g., the main program) and a single thread that calls **`FIFO_Put`**`()` (e.g., the serial port receive interrupt handler), then this **`FIFO_Put`**`()` function can interrupt this **`FIFO_Get`**`()` function without loss of data. So in this particular situation, interrupts would not have to be disabled. It would also operate properly if there were a single interrupt thread calling **`FIFO_Get`**`()` (e.g., the serial port transmit interrupt handler) and a single thread calling **`FIFO_Put`**`()` (e.g., the main program.) On the other hand, if the situation is more general, and multiple threads could call **`FIFO_Put`**`()` or multiple threads could call **`FIFO_Get`**`()`, then the interrupts would have to be temporarily disabled as shown.

You have to be careful when multiple threads are using the same resource

2.7.8 I/O Port Access

Even though the mechanism to access I/O ports technically does not fit the definition of pointer, it is included in this section because it involves addresses. The format used by the MX compiler fits the following model. The following listing shows three 32-bit K64 I/O ports. The line `FTM0_C5SC = 0x80;` generates a 32-bit I/O write operation to the port at address `0x4003800C`. The `FTM0_CNT` on the right hand side of the assignment statement generates a 32-bit I/O read operation from the port at address `0x40038004`. The `FTM0_C5V` on the left hand side of the assignment statement generates a 32-bit I/O write operation from the port at address `0x40038030`. The `FTM0_C5SC` inside the `while` loop generates repeated 32-bit I/O read operations until bit 7 is set.

```
#define FTM0_CNT  *(uint32_t volatile *)(0x40038004)
#define FTM0_C5SC *(uint32_t volatile *)(0x40038028)
#define FTM0_C5V  *(uint32_t volatile *)(0x40038030)

void wait(uint32_t delay)
{
  FTM0_C5SC &= ~0x80; // clear C5F
  FTM0_C5V = FTM0_CNT + delay; // CNT at end of wait
  while ((FTM0_C5SC & 0x80) == 0); // wait for C5F
}
```

Listing 2.64 – Sample MX program that accesses I/O ports

It was mentioned earlier that the `volatile` modifier will prevent the compiler from optimizing I/O statements, i.e., these examples would not work if the compiler read `FTM0_C5SC` once, then used the same data over and over inside the `while` loop.

To understand this syntax we break it into parts. Starting on the right is the absolute address of the I/O port. For example the K64 `FTM0_CNT` register is at location `0x40038004`. The parentheses are necessary because the definition might be used in an arithmetic calculation. For example the following two lines are quite different:

```
TheTime = *(unsigned char volatile *)(0x1023) + 100;
TheTime = *(unsigned char volatile *)0x1023 + 100;
```

In the second (incorrect) case the addition `0x01023 + 100` is performed on the address, not the data. The next part of the definition is a type casting. C allows you to change the type of an expression. For example `(unsigned char volatile *)` specifies that `0x1023` is an address that points at an 8-bit `unsigned char`. The `*` at the beginning of the definition causes the data to be fetched from the I/O port if the expression exists on the right-hand side of an assignment statement. The `*` also causes the data to be stored at the I/O port if the expression is on the left-hand side of the assignment statement. In this last way, I/O port accesses are indeed similar to pointers.

For example the previous example could have been implemented as:

```
uint32_t volatile *pFTM0_CNT;
uint32_t volatile *pFTM0_C5SC;
uint32_t volatile *pFTM0_C5V;

void wait(uint32_t delay)
{
  pFTM0_CNT = (uint32_t volatile *)(0x40038004);
  pFTM0_C5SC = (uint32_t volatile *)(0x40038028);
  pFTM0_C5V = (uint32_t volatile *)(0x40038030)
  (*pFTM0_C5SC) &= ~0x80;
  (*FTM0_C5V) = (*pFTM0_CNT) + delay;
  while (((*pFTM0_C5SC) & 0x80) == 0);
}
```

Listing 2.65 – C program that accesses I/O ports using pointers

This function first sets the three I/O pointers then accesses the I/O ports indirectly through the pointers.

You need to be careful when using pointer variables to I/O ports on the K64. If a global pointer variable to an I/O port is uninitialised, the C startup code will set it to zero. In C, the `NULL` pointer is defined as address `0`. In the K64, the initial stack pointer (held in Flash memory) has address `0`. Therefore, if you accidentally try and write to a dereferenced `NULL` pointer you will generate a HardFault exception (the program will “crash”).

2.8 Arrays and Strings

An array is a collection of like variables that share a single name. The individual elements of an array are referenced by appending a subscript, in square brackets `[]`, behind the name. The subscript itself can be any legitimate C expression that yields an integer value, even a general expression. Although arrays represent one of the simplest data structures, they have wide-spread usage in embedded systems.

Strings are similar to arrays with just a few differences. Usually, the array size is fixed, while strings can have a variable number of elements. Arrays can contain any data type (`char`, `short`, `int`, even other arrays) while strings are usually ASCII characters terminated with a `NULL` (`0`) character. In general we allow random access to individual array elements. On the other hand, we usually process strings sequentially character by character from start to end. Since these differences are a matter of semantics rather than specific limitations imposed by the syntax of the C programming language, the descriptions in this section apply equally to data arrays and character strings. String literals were discussed earlier; in this section we will define data structures to hold our strings. In addition, C has a rich set of predefined functions to manipulate strings.

2.8.1 Array Subscripts

When an array element is referenced, the subscript expression designates the desired element by its position in the data. The first element occupies position zero, the second position one, and so on. It follows that the last element is subscripted by `[N-1]` where `N` is the number of elements in the array. The statement:

```
data[9] = 0;
```

for instance, sets the tenth element of `data` to zero. The array subscript can be any expression that results in a 32-bit integer.

The following `for` -loop clears 100 elements of the array `data` to zero:

```
for (j=0; j < 100; j++)
  data[j] = 0;
```

Multidimensional Arrays

C supports arrays of multiple dimensions, which are stored in row-major order. In row-major order, consecutive elements of the rows of the array are contiguous in memory. Technically, C multidimensional arrays are just one-dimensional arrays whose elements are arrays. The syntax for declaring multidimensional arrays is:

```
int array2D[ROWS][COLUMNS];
```

where `ROWS` and `COLUMNS` are constants. This defines a two-dimensional array. Reading the subscripts from left to right, `array2D` is an array of length `ROWS`, each element of which is an array of `COLUMNS` integers.

As programmers we may assign any logical meaning to the first and second subscripts. For example we could consider the first subscript as the row and the second as the column. Then, the statement:

```
ThePosition = position[3][5];
```

copies the information from the 4th row and 6th column into the variable `ThePosition`.

If the array has three dimensions, then three subscripts are specified when referencing. Again we may assign any logical meaning to the various subscripts. For example we could consider the first subscript as the *x* coordinate, the second subscript as the *y* coordinate and the third subscript as the *z* coordinate. Then, the statement:

```
humidity[2][3][4] = 100;
```

sets the humidity at point (2, 3, 4) to 100.

Array subscripts are treated as signed 32-bit integers. It is the programmer's responsibility to see that only positive values are produced, since a negative subscript would refer to some point in memory preceding the array. One must be particularly careful about assuming what exists either in front of or behind our arrays in memory. C provides no facility for automatic bounds checking for array usage.

2.8.2 Array Declarations

Just like any variable, arrays must be declared before they can be accessed. The number of elements in an array is determined by its declaration. Appending a constant expression in square brackets to a name in a declaration identifies the name as the name of an array with the number of elements indicated. Multi-dimensional arrays require multiple sets of brackets. The examples in Listing 2.66 are valid declarations:

```
// define data, allocate space for 5 16-bit integers
short data[5];
// define string, allocate space for 20 8-bit characters
char string[20];
// define time, width, allocate space for 32-bit integers
int time, width[6];
// define xx, allocate space for 50 16-bit integers
short xx[10][5];
// define pts, allocate space for 125 16-bit integers
short pts[5][5][5];
// declare buffer as an external character array
extern char buffer[];
```

Listing 2.66 – Example showing array declarations

Notice in the third example that ordinary variables may be declared together with arrays in the same statement. In fact array declarations obey the syntax rules of ordinary declarations, as described in previous sections, except that certain names are designated as arrays by the presence of a dimension expression.

Notice the size of the external array, `buffer[]`, is not given. This leads to an important point about how C deals with array subscripts. The array dimensions are only used to determine how much memory to reserve. **It is the programmer's responsibility to stay within the proper bounds.** In particular, you must not let the subscript become negative or above `N-1`, where `N` is the size of the array.

Another situation in which an array's size need not be specified is when the array elements are given initial values. In this case, the compiler will determine the size of such an array from the number of initial values.

2.8.3 Array References

In C we may refer to an array in several ways. Most obviously, we can write subscripted references to array elements, as we have already seen. C interprets an unsubscripted array name as the address of the array. In the following example, the first two lines set `x` to equal the value of the first element of the array. The third and fourth lines both set `pt` equal to the address of the array. Recall that the address operator `&` yields the address of an object. This operator may also be used with array elements. Thus, the expression `&data[3]` yields the address of the fourth element. Notice too that `&data[0]` and `data+0` and `data` are all equivalent. It should be clear by analogy that `&data[3]` and `data+3` are also equivalent.

```
short x, *pt, data[5]; // a variable, a pointer, and an array

void Set(void)
{
  x = data[0];   // set x equal to the first element of data
  x = *data;     // set x equal to the first element of data
  pt = data;     // set pt to the address of data
  pt = &data[0]; // set pt to the address of data
  x = data[3];     // set x equal to the fourth element of data
  x = *(data + 3); // set x equal to the fourth element of data
  pt = data + 3; // set pt to the address of the fourth element
  pt = &data[3]; // set pt to the address of the fourth element
}
```

Listing 2.67 – Example showing array references

2.8.4 Pointers and Array Names

The previous examples suggest that pointers and array names might be used interchangeably, and, in many cases, they may. C will let us subscript pointers and also use array names as addresses. In the following example, the pointer `pt` contains the address of an array of integers. Notice the expression `pt[2]` is equivalent to `*(pt+2)`:

```
short *pt, data[5]; // a pointer, and an array

void Set(void)
{
  pt = data;        // set pt to the address of data
  data[2] = 5;      // set the third element of data to 5
  pt[2] = 5;        // set the third element of data to 5
  *(pt + 2) = 5;    // set the third element of data to 5
}
```

Listing 2.68 – Example showing pointers to access array elements

It is important to realize that although C accepts unsubscripted array names as addresses, they are not the same as pointers. In the following example, we cannot place the unsubscripted array name on the left-hand-side of an assignment statement:

```
short buffer[5], data[5]; // two arrays

void Set(void)
{
  data = buffer;      // illegal assignment
}
```

Listing 2.69 – Example showing an illegal array assignment

Since the unsubscripted array name is its address, the statement `data = buffer;` is an attempt to change its address. What sense would that make? The array, like any object, has a fixed home in memory; therefore, its address cannot be changed. We say that array is not an *lvalue*; i.e. it cannot be used on the left side of an assignment operator (nor may it be operated on by increment or decrement operators). It simply cannot be changed. Not only does this assignment make no sense, it is physically impossible because an array address is not a variable. There is no place reserved in memory for an array's address to reside, only the elements.

2.8.5 Negative Subscripts

Since a pointer may point to any element of an array, not just the first one, it follows that negative subscripts applied to pointers might well yield array references that are in bounds. This sort of thing might be useful in situations where there is a relationship between successive elements in an array and it becomes necessary to reference an element preceding the one being pointed to. In the following example, `data` is an array containing time-dependent (or space-dependent) information. If `pt` points to an element in the array, `pt[-1]` is the previous element and `pt[1]` is the following one. The function calculates the second derivative using a simple discrete derivative.

```c
short *pt, data[100]; // a pointer and an array

void CalcSecondDerivative(void)
{
  short d2Vdt2;

  for (pt = data + 1; pt < data + 99; pt++)
  {
    d2Vdt2 = (pt[-1] - 2 * pt[0] + pt[1]);
    ...
  }
}
```

Listing 2.70 – Example showing negative array subscripting

2.8.6 Address Arithmetic

As we have seen, addresses (pointers, array names, and values produced by the address operator) may be used freely in expressions. This one fact is responsible for much of the power of C.

As with pointers, all addresses are treated as unsigned quantities. Therefore, only unsigned operations are performed on them. Of all the arithmetic operations that could be performed on addresses, only two make sense: displacing an address by a positive or negative amount, and taking the difference between two addresses. All others, though permissible, yield meaningless results.

Displacing an address can be done either by means of subscripts or by use of the plus and minus operators, as we saw earlier. These operations should be used only when the original address and the displaced address refer to positions in the same array or data structure. Any other situation would assume a knowledge of how memory is organized and would, therefore, be ill-advised for portability reasons.

As we saw in the previous section on pointers, taking the difference of two addresses is a special case in which the compiler interprets the result as the number of objects lying between the addresses.

2.8.7 String functions in `string.h`

MX implements many useful string manipulation functions. Recall that strings are 8-bit arrays with a null-termination. The prototypes for these functions can be found in the `string.h` file. You simply include this file whenever you wish to use any of these routines. The rest of this section explains the functions one by one.

```
typedef unsigned int size_t;
void*  memchr(const void*, int, size_t);
int    memcmp(const void*, const void*, size_t);
void*  memcpy(void*, const void*, size_t);
void*  memmove(void*, const void*, size_t);
void*  memset(void*, int, size_t);
char*  strcat(char*, const char*);
char*  strchr(const char*, int);
int    strcmp(const char*, const char*);
int    strcoll(const char*, const char*);
char*  strcpy(char*, const char*);
size_t strcspn(const char*, const char*);
size_t strlen(const char*);
char*  strncat(char*, const char*, size_t);
int    strncmp(const char*, const char*, size_t);
char*  strncpy(char*, const char*, size_t);
char*  strpbrk(const char*, const char*);
char*  strrchr(const char*, int);
size_t strspn(const char*, const char*);
char*  strstr(const char*, const char*);
```

Listing 2.71 – Prototypes for string functions

The first five functions are general-purpose memory handling routines.

Scan Memory for a Character

```
void* memchr(const void* block, int c, size_t size);
```

This function finds the first occurrence of the byte `c` (converted to an `unsigned char`) in the initial `size` bytes of the object beginning at `block`. The return value is a pointer to the located byte, or a `NULL` pointer if no match was found.

Compare Two Blocks of Memory

```
int memcmp(const void* a1, const void* a2, size_t size);
```

The function `memcmp` compares the `size` bytes of memory beginning at `a1` against the `size` bytes of memory beginning at `a2`. The value returned has the same sign as the difference between the first differing pair of bytes, `a1[n]-a2[n]` (interpreted as `unsigned char` objects, then promoted to `int`). If the contents of the two blocks are equal, `memcmp` returns `0`.

Copy a Block of Memory

```
void* memcpy(void* dst, const void* src, size_t size);
```

The `memcpy` function copies `size` bytes from the object beginning at `src` into the object beginning at `dst`. The behaviour of this function is undefined if the two arrays `src` and `dst` overlap. The value returned by `memcpy` is the value of `dst`.

Move a Block of Memory

```
void* memmove(void* dst, const void* src, size_t size);
```

`memmove` copies the `size` bytes at `src` into the `size` bytes at `dst`, even if those two blocks of space overlap. In the case of overlap, `memmove` is careful to copy the original values of the bytes in the block at `src`, including those bytes which also belong to the block at `dst`. The value returned by `memmove` is the value of `dst`.

Fill a Block of Memory

```
void* memset(void* block, int c, size_t size);
```

This function copies the value of `c` (converted to an `unsigned char`) into each of the first `size` bytes of the object beginning at `block`. It returns the value of `block`.

The remaining functions are string-handling routines.

Concatenate Strings

```
char* strcat(char* dst, const char* src);
```

Assuming the two pointers are directed at two null-terminated strings, `strcat` will append a copy of the string pointed to by pointer `src`, placing it at the end of the string pointed to by pointer `dst`. The pointer `dst` is returned. It is the programmer's responsibility to ensure the destination buffer is large enough. This function has undefined results if the strings overlap.

Locate the First Occurrence of a Character in a String

```
char* strchr(const char* string, int c);
```

Assuming the pointer is directed at a null-terminated string, starting in memory at address `string`, `strchr` will search for the first occurrence of the character `c` (converted to a `char`). It will search until a match is found or stop at the end of the string. If successful, a pointer to the located character is returned, otherwise a `NULL` pointer is returned.

Compare Two Strings (using Locale)

```
int strcmp(const char* s1, const char* s2);
int strcoll(const char* s1, const char* s2);
```

Assuming the two pointers are directed at two null-terminated strings, `strcmp` will return a negative value if the string pointed to by `s1` is lexicographically less than the string pointed to by `s2`. The return value will be zero if they match, and positive if the string pointed to by `s1` is lexicographically greater than the string pointed to by `s2`. A consequence of the ordering used by `strcmp` is that if `s1` is an initial substring of `s2`, then `s1` is considered to be “less than” `s2`.

In general C allows the comparison rule used in `strcoll` (string collate) to depend on the current locale, but in MX `strcoll` is the same as `strcmp`.

Copy a String

```
char* strcpy(char* dst, const char* src);
```

We assume `src` points to a null-terminated string and `dst` points to a memory buffer large enough to hold the string. **`strcpy`** will copy the string (including the null) pointed to by `src`, into the buffer pointed to by pointer `dst`. The pointer `dst` is returned. It is the programmer's responsibility to ensure the destination buffer is large enough.

Get Span until Character in String

```
size_t strcspn(const char* string, const char* stopset);
```

The string function **`strcspn`** (string complement span) will compute the length of the maximal initial substring within the string pointed to by `string` that has no characters in common with the string pointed to by `stopset`. For example the following call returns the value `5`:

```
n = strcspn("label: movw r2,#4660 ;comment", " ;:*\n\t\l");
```

A common application of this routine is parsing for tokens. The first parameter is a line of text and the second parameter is a list of delimiters (e.g., space, semicolon, colon, star, return, tab and linefeed). The function returns the length of the first token (i.e., the size of `label`).

Get String Length

```
size_t strlen(const char* string);
```

The string function **`strlen`** returns the length of the string pointed to by pointer `string`. The length is the number of characters in the string not counting the null-termination.

Append Characters from a String

```
char* strncat(char* dst, const char* src, size_t size);
```

This function is similar to `strcat`. Assuming the two pointers are directed at two null-terminated strings, `strncat` will append a copy of the string pointed to by pointer `src`, placing it the end of the string pointed to by pointer `dst`. The parameter `size` limits the number of characters, not including the null, that will be copied. The pointer `dst` is returned. It is the programmer's responsibility to ensure the destination buffer is large enough. The behaviour of `strncat` is undefined if the strings overlap.

Compare Characters of Two Strings

```
int strncmp(const char* s1, const char* s2, size_t size);
```

This function is similar to `strcmp`. Assuming the two pointers are directed at two null-terminated strings, `strncmp` will return a negative value if the string pointed to by `s1` is lexicographically less than the string pointed to by `s2`. The return value will be zero if they match, and positive if the string pointed to by `s1` is lexicographically greater than the string pointed to by `s2`. The parameter `size` limits the number of characters, not including the null, that will be compared. For example, the following function call will return a zero because the first 5 characters are the same:

```
n = strncmp("MK64FN1M0VMJ12", "MK64FX512VMJ12", 5);
```

Copy Characters from a String

```
char* strncpy(char* dst, const char* src, size_t size);
```

We assume `src` points to a null-terminated string and `dst` points to a memory buffer large enough to hold the string. `strncpy` will copy the string (including the null) pointed to by `src`, into the buffer pointed to by pointer `dst`. The pointer `dst` is returned. The parameter `size` limits the number of characters, not including the null, that will be copied. If the size of the string pointed to by `src` is equal to or larger than `size`, then the null will not be copied into the buffer pointed to by `dst`. It is the programmer's responsibility to ensure the destination buffer is large enough. The behaviour of `strncpy` is undefined if the strings overlap.

Locate Characters in a String

```
char* strpbrk(const char* string, const char* stopset);
```

This function `strpbrk` (string pointer break) will search the string pointed to by `string` for the first instance of any of the characters in the string pointed to by `stopset`. A pointer to the found character is returned. If the search fails to find any characters of the string pointed to by `stopset` in the string pointed to by `string`, then a null pointer is returned. For example the following call returns a pointer to the colon:

```
pt = strpbrk("label: movw r2,#4660 ;comment", " ;:*\n\t\l");
```

This function, like `strcspn`, can be used for parsing tokens.

Locate the Last Occurrence of a Character in a String

```
char* strrchr(const char* string, int c);
```

The function `strrchr` will search the string pointed to by `string` from the right for the first instance of the character in `c`. A pointer to the found character is returned. If the search fails to find an occurrence of the character `c` (converted to a `char`) in the string pointed to by `string`, then a null pointer is returned. For example the following calls set `pt1` to point to the '`m`' in `movw` and `pt2` to point to the second '`m`' in `;comment`:

```
pt1 = strchr("label: movw r2,#4660 ;comment", 'm');
pt2 = strrchr("label: movw r2,#4660 ;comment", 'm');
```

Notice that `strchr` searches from the left while `strrchr` searches from the right.

Get Span of Character Set in String

```
size_t strspn(const char* string, const char* skipset);
```

The `strspn` (string span) function returns the length of the maximal initial substring of `string` that consists entirely of characters that are members of the set specified by the string `skipset`. The order of the characters in `skipset` is not important.

In the following example the second string contains the valid set of hexadecimal digits.

```
n = strspn("A12F05 + 12BAD * 45", "01234567890ABCDEF");
```

The function call will return `6` because there is a valid 6-digit hexadecimal string at the start of the line.

Locate Substring

```
char* strstr(const char* haystack, const char* needle);
```

The function `strstr` will search the string pointed to by `haystack` from the left for the first instance of the string pointed to by `needle`. A pointer to the found substring within the first string is returned. If the search fails to find a match, then a null pointer is returned. For example, the following call sets `pt` to point to the '`m`' in `movw`:

```
pt = strstr("label: movw r2,#4660 ;comment", "movw");
```

2.8.8 A FIFO Queue Example using Indices

Another method to implement a statically allocated first-in-first-out FIFO is to use indices instead of pointers. The purpose of this example is to illustrate the use of arrays and indices. Just like the previous FIFO, this is used for order-preserving temporary storage. The function **`FIFO_Put`** will enter one 8-bit byte into the queue, and **`FIFO_Get`** will remove one byte. If you call **`FIFO_Put`** while the FIFO is full (`Size` is equal to `FIFO_SIZE`), the routine will return a zero. Otherwise, **`FIFO_Put`** will save the data in the queue and return a one. The index `PutI` specifies where to put the next 8-bit data. The routine **`FIFO_Get`** actually returns two parameters. The queue status is the regular function return parameter, while the data removed from the queue is returned by reference, i.e., the calling routine passes in a pointer, and **`FIFO_Get`** stores the removed data at that address. If you call **`FIFO_Get`** while the FIFO is empty (`Size` is equal to zero), the routine will return a zero. Otherwise, **`FIFO_Get`** will return the oldest data from the queue and return a one. The index `GetI` specifies where to get the next 8-bit data.

The following FIFO implementation uses two indices and a counter.

```
// Index, counter implementation of the FIFO
#define FIFO_SIZE 10   // Number of 8 bit data in the FIFO

unsigned char PutI;    // Index of where to put next
unsigned char GetI;    // Index of where to get next
unsigned char Size;    // Number currently in the FIFO
               // FIFO is empty if Size == 0
               // FIFO is full  if Size == FIFO_SIZE
char FIFO[FIFO_SIZE];  // The statically allocated data

void FIFO_Init(void)
{
  PutI = GetI = Size = 0;   // Empty when Size==0
}

int FIFO_Put(const char data)
{
  if (Size == FIFO_SIZE)
    return 0;               // Failed, FIFO was full

  Size++;
  FIFO[PutI++] = data;      // Put data into FIFO
  if (PutI == FIFO_SIZE)
    PutI = 0;               // Wrap
  return 1;                 // Successful
}

int FIFO_Get(char* const dataPt)
{
  if (Size == 0)
    return 0;               // Empty if Size == 0

  *dataPt = FIFO[GetI++];   // Get data out of FIFO
  Size--;
  if (GetI == FIFO_SIZE)
    GetI = 0;               // Wrap
  return 1;                 // Successful
}
```

Listing 2.72 – FIFO implemented with two indices and a counter

2.9 Structures

A structure is a collection of variables that share a single name. In an array, each element has the same format. With structures we specify the types and names of each of the elements or members of the structure. The individual members of a structure are referenced by their subname. Therefore, to access data stored in a structure, we must give both the name of the collection and the name of the element. Structures are one of the most powerful features of the C language. In the same way that functions allow us to extend the C language to include new operations, structures provide a mechanism for extending the data types. With structures we can add new data types derived from an aggregate of existing types.

2.9.1 Structure Declarations

Like other elements of C programming, the structure must be declared before it can be used. The declaration specifies the tagname of the structure and the names and types of the individual members. The following example has three members: one 32-bit integer and three pointers to 32-bit unsigned integers:

```
struct theport
{
  // 0 for I/O, 1 for in only, -1 for out only
  int mode;
  // pointer to its output address
  uint32_t volatile* outAddress;
  // pointer to its input address
  uint32_t volatile* inAddress;
  // pointer to its data direction register
  uint32_t volatile* ddr;
};
```

The above declaration does not create any variables or allocate any space. Therefore to use a structure we must define a global or local variable of this type. The tagname (`theport`) along with the keyword `struct` can be used to define variables of this new data type:

```
struct theport PortA, PortB, PortC;
```

The previous line defines the four variables and allocates 16 bytes for each variable. If you knew you needed just three copies of structures of this type, you could have defined them as:

```
struct theport
{
  int mode;
  uint32_t volatile* outAddress;
  uint32_t volatile* inAddress;
  uint32_t volatile* ddr;
} PortA, PortB, PortC;
```

Definitions like the above are hard to extend, so to improve code reuse we can use `typedef` to actually create a new data type (called `port` in the example below) that behaves syntactically like `char`, `int`, `short` etc.

```
struct theport
{
  int mode;     // 0 for I/O, 1 for in only, -1 for out only
  uint32_t volatile* outAddress; // out address
  uint32_t volatile* inAddress;  // in address
  uint32_t volatile* ddr;        // data direction register
};

typedef struct theport port;

port PortA, PortB, PortC;
```

Once we have used `typedef` to create `port`, we don't need access to the name `theport` anymore. Consequently, some programmers use the following short-cut:

```
typedef struct
{
  int mode;     // 0 for I/O, 1 for in only, -1 for out only
  uint32_t volatile* outAddress; // out address
  uint32_t volatile* inAddress;  // in address
  uint32_t volatile* ddr;        // data direction register
} port;

port PortA, PortB, PortC;
```

2.9.2 Accessing Members of a Structure

We need to specify both the structure name (name of the variable) and the member name when accessing information stored in a structure. The following examples show accesses to individual members:

```
PortA.mode = -1;        // Specify Port A as output
PortA.outAddress = (uint32_t volatile *)(0x400FF000);
PortA.inAddress = (uint32_t volatile *)(0x400FF010);
PortA.ddr = (uint32_t volatile *)(0x400FF014);
(*PortA.ddr) = 0xFFFFFFFF;

PortB.mode = 0;         // Port B is input and output
PortA.outAddress = (uint32_t volatile *)(0x400FF040);
PortA.inAddress = (uint32_t volatile *)(0x400FF050);
PortB.ddr = (uint32_t volatile *)(0x400FF054);

// Copy from PortB to PortA
(*PortA.outAddress) = (*PortB.inAddress);
```

The syntax can get a little complicated when a member of a structure is another structure as illustrated in the next example:

```
typedef struct
{
  int x1, y1;     // starting point
  int x2, y2;     // starting point
  char color;     // color
} line;

typedef struct
{
  line L1, L2;    // two lines
  char direction;
} path;

path p;           // global

void Setup(void)
{
  line myLine;
  path q;
  p.L1.x1 = 5;     // black line from 5,6 to 10,12
  p.L1.y1 = 6;
  p.L1.x2 = 10;
  p.L1.y2 = 12;
  p.L1.color = 255;
  p.L2={5, 6, 10, 12, 255};  // black line from 5,6 to 10,12
  p.direction = -1;
  myLine = p.L1;
  q = {{0, 0, 5, 6, 128}, {5, 6, -10, 6, 128}, 1};
  q = p;
}
```

Listing 2.73 – Examples of accessing structures

The local variable declaration `line myLine;` will allocate 17 bytes on the stack while `path q;` will allocate 35 bytes on the stack. In actuality most C compilers in an attempt to maintain addresses on word boundaries will actually allocate 20 and 44 bytes respectively. In particular, the K64 executes faster out of external memory if 32-bit accesses occur on word-aligned addresses. For example, a 32-bit data access to an external odd address requires two bus cycles, while a 32-bit data access to an external word-aligned address requires only one bus cycle. There is no particular odd-address speed penalty for K64 internal addresses (internal RAM or Flash). Notice that the expression `p.L1.x1` is of the type **`int`**, the term `p.L1` has the type `line`, while just `p` has the type `path`. The expression `q = p;` will copy the entire 35 bytes that constitute the structure from `p` to `q`.

2.9.3 Initialization of a Structure

Just like any variable, we can specify the initial value of a structure at the time of its definition:

```
path thePath = {{0, 0, 5, 6, 128}, {5, 6, -10, 6, 128}, 1};
line theLine = {0, 0, 5, 6, 128};
port PortE =
{
  0,
  (uint32_t volatile *)(0x400FF100),
  (uint32_t volatile *)(0x400FF110),
  (uint32_t volatile *)(0x400FF114)
};
```

If we leave part of the initialization blank it is filled with zeros.

```
path thePath = {{0, 0, 5, 6, 128}, };
line theLine = {5, 6, 10, 12, };
port PortE = {1, (uint32_t volatile *)(0x400FF100), };
```

To place a structure in Flash memory, we define it as a global constant. In the following example the structure `fsm[3]` will be allocated and initialized in Flash memory. The linked structure of a finite state machine is a good example of a Flash-based structure.

```
typedef const struct State
{
  unsigned char out;            // Output to Port A bits 0-7
  unsigned short wait;          // Time (bus cycles) to wait
  unsigned char andMask[4];
  unsigned char equMask[4];
  const struct State* next[4]; // Next states
} TState;

typedef TState* PState;

#define Stop &FSM[0]
#define Turn &FSM[1]
#define Bend &FSM[2]

TState FSM[3] =
{
  {
     0x34,   2000,   // stop 1 ms
    {0xFF,   0xF0,   0x27,   0x00},
    {0x51,   0xA0,   0x07,   0x00},
    {Turn,   Stop,   Turn,   Bend}
  },
  {
     0xB3,   5000,   // turn 2.5 ms
    {0x80,   0xF0,   0x00,   0x00},
    {0x00,   0x90,   0x00,   0x00},
    {Bend,   Stop,   Turn,   Turn}
  },
  {
     0x75,   4000,   // bend 2 ms
    {0xFF,   0x0F,   0x01,   0x00},
    {0x12,   0x05,   0x00,   0x00},
    {Stop,   Stop,   Turn,   Stop}
  }
};
```

Listing 2.74 – Example of initializing a structure in Flash

2.9.4 Using pointers to access structures

Just like other variables we can use pointers to access information stored in a structure. The syntax is illustrated in the following examples:

```
void Setup(void)
{
  path* ppt;
  ppt = &p;          // pointer to an existing global variable
  ppt->L1.x1 = 5;   // black line from 5,6 to 10,12
  ppt->L1.y1 = 6;
  ppt->L1.x2 = 10;
  ppt->L1.y2 = 12;
  ppt->L1.color = 255;
  ppt->L2 = {5, 6, 10, 12, 255};
  ppt->direction = -1;
  (*ppt).direction = -1;
}
```

Listing 2.75 – Examples of accessing a structure using a pointer

Notice that the syntax `ppt->direction` is equivalent to `(*ppt).direction`. The parentheses in this access are required, because along with `()` and `[]`, the operators `.` and `->` have the highest precedence and associate from left to right. Therefore `*ppt.direction` would be a syntax error because `ppt.direction` cannot be evaluated.

As an another example of pointer access, consider the finite state machine controller for the `fsm[3]` structure shown previously. The state machine is illustrated below, along with the program.

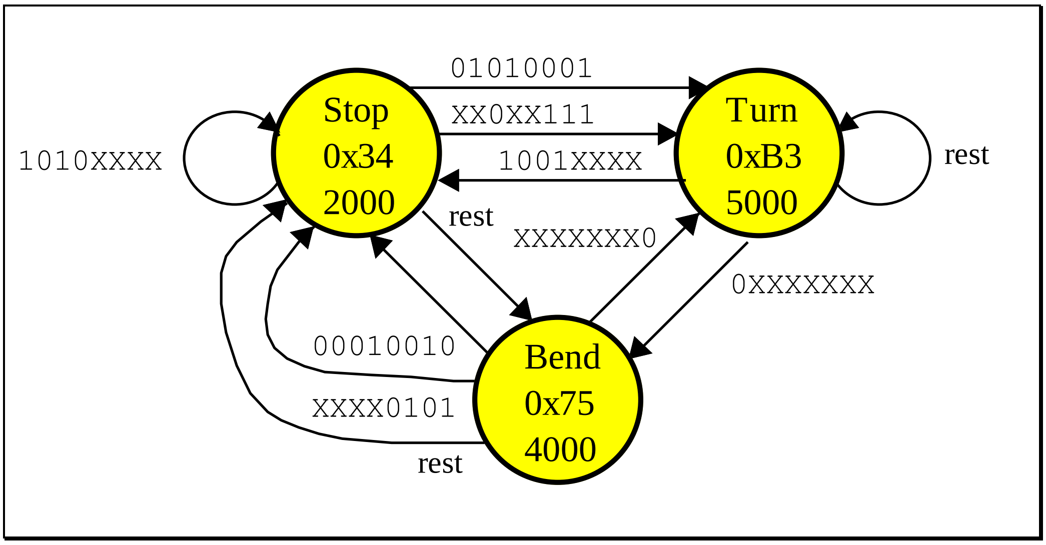


Figure 2.42 – Finite state machine

```
void Control(void)
{
  PState pt;
  unsigned char input;
  unsigned short startTime;

  FTM0_MODE |= 0x01;          // Enable timer
  FTM0_SC |= 0x01;            // timer/2 (500ns)
  GPIOA_PDDR = 0x000000FF;    // PortA bits 7-0 are outputs
  GPIOB_PDDR = 0x00000000;    // PortB bits 7-0 are inputs
  pt = stop;                  // Initial State

  while(1)
  {
    // 1) output
    GPIOA_PDOR = pt->out;
    // Time (500 ns each) to wait
    startTime = FTM0_CNT;
    // 2) wait
    while ((FTM0_CNT - startTime) <= pt->wait);
    // 3) input
    input = GPIOB_PDIR;
    for (int i = 0; i < 4; i++)
      if ((input & pt->andMask[i]) == pt->equMask[i])
      {
        // 4) next depends on input
        pt = pt->next[i];
        i = 4;
      }
  }
}
```

Listing 2.76 – Finite state machine controller for K64

2.9.5 Passing Structures to Functions

Like any other data type, we can pass structures as parameters to functions. Because most structures occupy a large number of bytes, it makes more sense to pass the structure by reference rather than by value. In the following "call by value" example, the entire 16-byte structure is copied on the stack when the function is called:

```
unsigned char Input(port thePort)
{
  return (*thePort.inAddress);
}
```

When we use "call by reference", a pointer to the structure is passed when the function is called.

```
typedef const struct
{
  int mode;      // 0 for I/O, 1 for in only, -1 for out only
  uint32_t volatile* outAddress; // out address
  uint32_t volatile* inAddress;  // in address
  uint32_t volatile* ddr;        // data direction register
} port;

port PortC =
{
  0,
  (uint32_t volatile *)(0x400FF080),
  (uint32_t volatile *)(0x400FF090),
  (uint32_t volatile *)(0x400FF094)
};

int MakeOutput(port* ppt)
{
  if (ppt->mode == 1)
    return 0; // input only
  if (ppt->mode == -1)
    return 1; // OK, output only
  (*ppt->ddr) = 0xFFFFFFFF; // make output
  return 1;
}

int MakeInput(port* ppt)
{
  if (ppt->mode == -1)
    return 0; // output only
  if (ppt->mode == 1)
    return 1;  // OK, input only
  (*ppt->ddr) = 0x00000000; // make input
  return 1;
}
```

```c
unsigned int Input(port* ppt)
{
  return (*ppt->inAddress);
}

void Output(port* ppt, unsigned int data)
{
  (*ppt->outAddress) = data;
}

void main(void)
{
  unsigned int myData;

  MakeInput(&PortC);
  MakeOutput(&PortC);
  Output(&PortC, 0);
  myData = Input(&PortC);
}
```

Listing 2.77 – Port access organized with a data structure

2.9.6 Linear Linked Lists

One of the applications of structures involves linking elements together with pointers. A linear linked list is a simple one-dimensional data structure where the nodes are chained together one after another. Each node contains data and a link to the next node. The first node is pointed to by the `HeadPt` and the last node has a null-pointer in the next field. A node could be defined as:

```c
typedef struct node
{
  unsigned short data;  // 16 bit information
  struct node* next;    // pointer to the next node
} TNode;

TNode* HeadPt;
```

Listing 2.78 – Linear linked list node structure

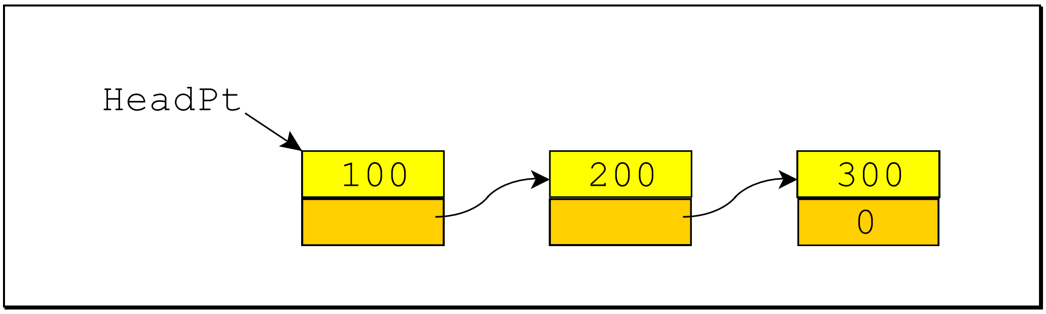


Figure 2.43 – Linear linked list with 3 nodes

In order to store more data in the structure, we will first create a new node then link it into the list. The routine `StoreData` will return a true value if successful.

```
#include <stdlib.h>

int StoreData(unsigned short info)
{
  TNode* pt;
  pt = malloc(sizeof(TNode));  // create a new entry
  if (pt)
  {
    pt->data = info;           // store data
    pt->next = HeadPt;         // link into existing
    HeadPt = pt;
    return 1;
  }
  return 0;                   // out of memory
}
```

Listing 2.79 – Code to add a node at the beginning of a linear linked list

In order to search the list we start at the `HeadPt`, and stop when the pointer becomes `NULL`. The routine `Search` will return a pointer to the node if found, and it will return a null-pointer if the data is not found.

```
TNode* Search(unsigned short info)
{
  TNode* pt;

  pt = HeadPt;
  while (pt)
  {
    if (pt->data == info)
      return pt;
    pt = pt->next;    // link to next
  }
  return pt;       // not found
}
```

Listing 2.80 – Code to find a node in a linear linked list

To count the number of elements, we again start at the `HeadPt`, and stop when the pointer becomes `NULL`. The routine **`Count`** will return the number of elements in the list.

```
unsigned short Count(void)
{
  TNode* pt;
  unsigned short count;

  count = 0;
  pt = HeadPt;
  while (pt)
  {
    count++;
    pt = pt->next;   // link to next
  }
  return count;
}
```

Listing 2.81 – Code to count the number of nodes in a linear linked list

If we wanted to maintain a sorted list, then we can insert new data at the proper place, in between data elements smaller and larger than the one we are inserting. In the following figure we are inserting the element 250 in between elements 200 and 300.

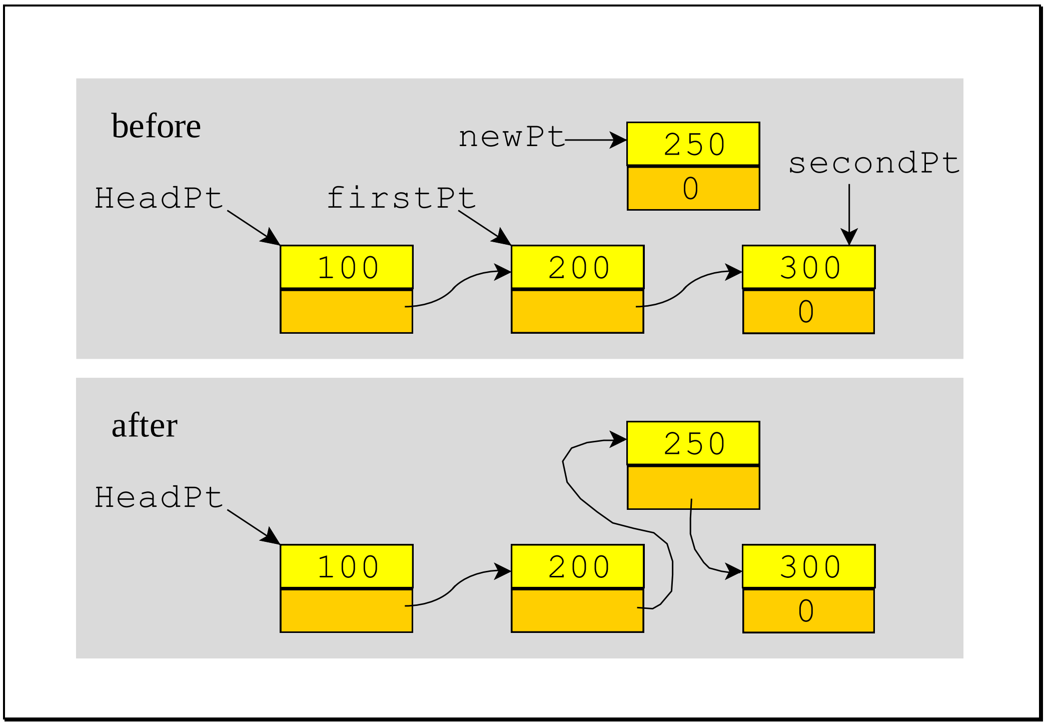


Figure 2.44 – Inserting a node in sorted order

There are 4 cases to consider. In case 1, the list is initially empty, and this new element is the first and only one. In case 2, the new element is inserted at the front of the list because it has the smallest data value. Case 3 is the general case depicted in the previous figure. In this situation, the new element is placed in between `firstPt` and `secondPt`. In case 4, the new element is placed at the end of the list because it has the largest data value.

```
int InsertData(unsigned short info)
{
  TNode *firstPt, *secondPt, *newPt;

  newPt = malloc(sizeof(TNode));  // create a new entry
  if (newPt)
  {
    newPt->data = info;       // store data

    // case 1
    if (HeadPt == 0)
    {
      newPt->next = HeadPt;   // only element
      HeadPt = newPt;
      return 1;
    }
    // case 2
    if (info <= HeadPt->data)
    {
      newPt->next = HeadPt;   // first element in list
      HeadPt = newPt;
      return 1;
    }
    // case 3
    firstPt = HeadPt;         // search from beginning
    secondPt = HeadPt->next;
    while (secondPt)
    {
      if (info <= secondPt->data)
      {
        newPt->next = secondPt;   // insert element here
        firstPt->next = newPt;
        return 1;
      }
      firstPt = secondPt;     // search next
      secondPt = secondPt->next;
    }
    // case 4
    newPt->next = secondPt;   // insert at end
    firstPt->next = newPt;
    return 1;
  }
  return 0;                  // out of memory
}
```

Listing 2.82 – Code to insert a node in a sorted linear linked list

The following function will search and remove a node from the linked list. Case 1 is the situation in which an attempt is made to remove an element from an empty list. The return value of zero signifies the attempt failed. In case 2, the first element is removed. In this situation the `HeadPt` must be updated to now point to the second element. It is possible the second element does not exist, because the list originally had only one element. This is okay because in this situation `HeadPt` will be set to `NULL` signifying the list is now empty. Case 3 is the general situation in which the element at `secondPt` is removed. The element before, `firstPt`, is now linked to the element after. Case 4 is the situation where the element that was requested to be removed did not exist. In this case, the return value of zero signifies the request failed.

```
int Remove(unsigned short info)
{
  TNode *firstPt, *secondPt;

  // case 1
  if (HeadPt == 0)
    return 0;   // empty list

  // case 2
  firstPt = HeadPt;
  secondPt = HeadPt->next;
  if (info == HeadPt->data)
  {
    HeadPt = secondPt; // remove first element in list
    free(firstPt);     // return unneeded memory to heap
    return 1;
  }

  // case 3
  while (secondPt)
  {
    if (secondPt->data == info)
    {
      firstPt->next = secondPt->next; // remove this one
      free(secondPt);   // return unneeded memory to heap
      return 1;
    }
    firstPt = secondPt;   // search next
    secondPt = secondPt->next;
  }

  // case 4
  return 0;    // not found
}
```

Listing 2.83 – Code to remove a node from a sorted linear linked list

2.9.7 Example of a Huffman Code

When information is stored or transmitted there is a fixed cost for each bit. Data compression and decompression provide a means to reduce this cost without loss of information. If the sending computer compresses a message before transmission and the receiving computer decompresses it at the destination, the effective bandwidth is increased. In particular, this example introduces a way to process bit streams using Huffman encoding and decoding. A typical application is illustrated by the following flow diagram.

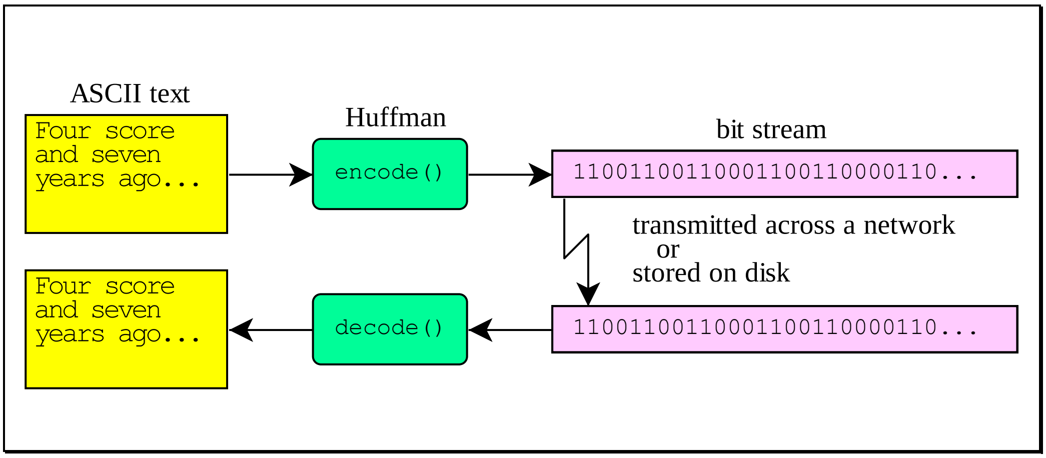


Figure 2.45 – Data flow diagram showing a typical application of Huffman encoding and decoding

The Huffman code is similar to the Morse code in that they both use short patterns for letters that occur more frequently. In regular ASCII, all characters are encoded with the same number of bits (8). Conversely, with the Huffman code, we assign codes where the number of bits to encode each letter varies. In this way, we can use short codes for letters like "e t a o i n" (that have a higher probability of occurrence) and long codes for seldom used consonants like "j x q z" (that have a lower probability of occurrence).

To illustrate the encode-decode operations, consider the following Huffman code for the letters M, I, P and S. S is encoded as "0", I as "10", P as "110" and M as "111". We can store a Huffman code as a binary tree.

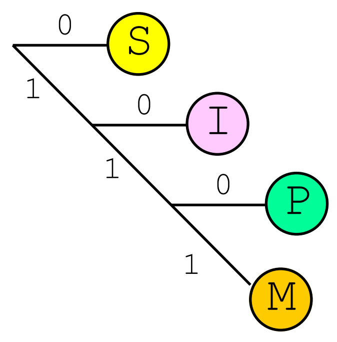


Figure 2.46 – Huffman code for the letters S I P M

If "MISSISSIPPI" were to be stored in ASCII, it would require 10 bytes or 80 bits. With this simple Huffman code, the same string can be stored in 21 bits.

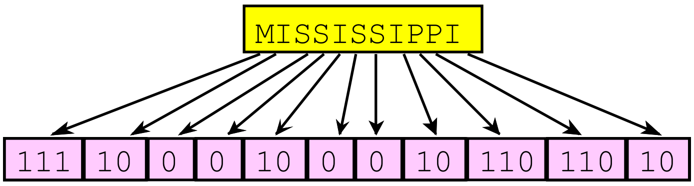


Figure 2.47 – Huffman encoding for MISSISSIPPI

Of course, this Huffman code can only handle 4 letters, while the ASCII code has 128 possibilities, so it is not fair to claim we have an 80 to 21 bit saving. Nevertheless, for information that has a wide range of individual probabilities of occurrence, a Huffman code will be efficient.

2.190

In the following implementation the functions `BitPut()` and `BitGet()` are called to save and recover binary data. The implementations of these two functions are not shown.

```c
typedef const struct Node
{
  char letter0;     // ASCII code if binary 0
  char letter1;     // ASCII code if binary 1
  // letter1 is NULL(0) if link is pointer to another node
  const struct Node* link; // binary tree pointer
} TNode;

typedef TNode* PNode;

// Huffman tree
TNode twentysixth= {'Q','Z',0};
TNode twentyfifth= {'X',0,&twentysixth};
TNode twentyfourth={'J',0,&twentyfifth};
TNode twentythird= {'K',0,&twentyfourth};
TNode twentysecond={'V',0,&twentythird};
TNode twentyfirst= {'B',0,&twentysecond};
TNode twentyth=    {'P',0,&twentyfirst};
TNode ninteenth=   {'Y',0,&twentyth};
TNode eighteenth=  {'G',0,&ninteenth};
TNode seventeenth= {'F',0,&eighteenth};
TNode sixteenth=   {'W',0,&seventeenth};
TNode fifteenth=   {'M',0,&sixteenth};
TNode fourteenth=  {'C',0,&fifteenth};
TNode thirteenth=  {'U',0,&fourteenth};
TNode twelfth=     {'L',0,&thirteenth};
TNode eleventh=    {'D',0,&twelfth};
TNode tenth=       {'R',0,&eleventh};
TNode ninth=       {'H',0,&tenth};
TNode eighth=      {'S',0,&ninth};
TNode seventh=     {' ',0,&eighth};
TNode sixth=       {'N',0,&seventh};
TNode fifth=       {'I',0,&sixth};
TNode fourth=      {'O',0,&fifth};
TNode third=       {'A',0,&fourth};
TNode second=      {'T',0,&third};
TNode root=        {'E',0,&second};
```

```c
// ********encode***************
// convert ASCII string to Huffman bit sequence
// input is a null-terminated ASCII string
// returns bit count if OK
// returns 0          if BitFIFO full
// returns 0xFFFF     if illegal character

short encode(char* sPt)
{
  short notFound;
  char data;
  short bitCount = 0;     // number of bits created
  PNode hpt;              // pointer into Huffman tree

  while (data = (*sPt))
  {
    sPt++;                // next character
    hpt = &root;          // start search at root
    notFound = 1;         // changes to 0 when found
    while (notFound)
    {
      if ((hpt->letter0) == data)
      {
        if (!BitPut(0))
          return 0;       // data structure full
        bitCount++;
        notFound = 0;
      }
      else
      {
        if (!BitPut(1))
          return 0;       // data structure full
        bitCount++;
        if ((hpt->letter1) == data)
          notFound = 0;
        else
        {       // doesn't match either letter0 or letter1
          hpt = hpt->link;
          if (hpt == 0)
            return 0xFFFF; // illegal, end of tree?
        }
      }
    }
  }
  return bitCount;
}
```

```c
// *******decode**************
// convert Huffman bit sequence to ASCII
// output is a null-terminated ASCII string
// will remove from the BitFIFO until it is empty
// returns character count

short decode(char* sPt)
{
  short charCount = 0; // number of ASCII characters created
  short notFound;
  unsigned short data;
  PNode hpt;           // pointer into Huffman tree

  hpt = &root;         // start search at root
  while (BitGet(&data))
  {
    if (data == 0)
    {
      (*sPt) = hpt->letter0;
       sPt++;
       charCount++;
       hpt = &root;    // start over and search at root
    }
    else  //data is 1
      if (hpt->link == 0)
      {
        (*sPt) = hpt->letter1;
        sPt++;
        charCount++;
        hpt = &root;   // start over and search at root
      }
      else       // doesn't match either letter0 or letter1
        hpt = hpt->link;
  }
  (*sPt) = 0;  // null terminated
  return charCount;
}
```

Listing 2.84 – A Huffman code implementation

2.10 Functions

We have been using functions throughout this document, but have put off formal presentation until now because of their immense importance. The key to effective software development is the appropriate division of a complex problem into modules. A module is a software task that takes inputs and operates in a well-defined way to create outputs. In C, functions are our way to create modules. A small module may be a single function. A medium-sized module may consist of a group of functions together with global data structures, collected in a single file. A large module may include multiple medium-sized modules. A hierarchical software system combines these software modules in either a top-down or bottom-up fashion. We can consider the following criteria when we decompose a software system into modules:

1) We wish to make the overall software system easy to understand;
2) We wish to minimize the coupling or interactions between modules;
3) We wish to group together I/O port accesses to similar devices;
4) We wish to minimize the size (maximize the number) of modules;
5) Modules should be able to be tested independently;
6) We should be able to replace / upgrade one module without affecting the others;
7) We would like to reuse modules in other situations.

It is essential to divide a large software task into smaller, well-defined and easy to debug modules.

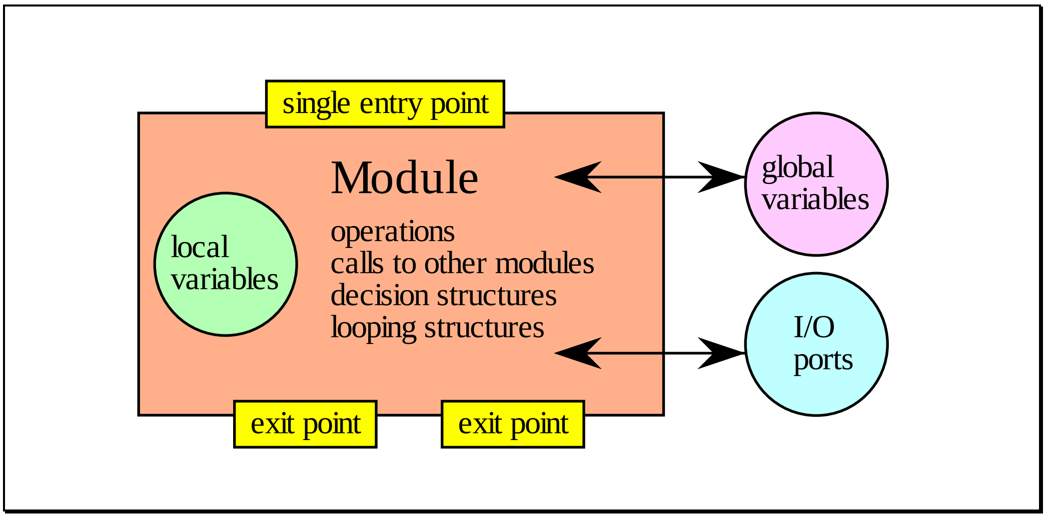


Figure 2.48 – A module has inputs and outputs

As a programmer we must take special care when dealing with global variables and I/O ports. In order to reduce the complexity of the software we will limit access to global variables and I/O ports.

The term *function* in C is based on the concept of mathematical functions. In particular, a mathematical function is a well-defined operation that translates a set of input values into a set of output values. In C, a function translates a set of input values into a single output value. We will develop ways for our C functions to return multiple output values and for a parameter to be both an input and an output parameter. As a simple example consider the function that converts temperature in degrees F into temperature in degrees C:

```
short FtoC(short tempF)
{
  short tempC;

  tempC = (5 * (tempF - 32)) / 9;   // conversion
  return tempC;
}
```

When the function's name is written in an expression, together with the values it needs, it represents the result that it produces. In other words, an operand in an expression may be written as a function name together with a set of values upon which the function operates. The resulting value, as determined by the function, replaces the function reference in the expression. For example, in the expression:

```
// T+2 degrees Fahrenheit plus 4 degrees Centigrade
FtoC(T + 2) + 4;
```

the term `FtoC(T + 2)` names the function `FtoC` and supplies the variable `T` and the constant `2` from which `FtoC` derives a value, which is then added to `4`. The expression effectively becomes:

```
((5 * ((T + 2) - 32)) / 9) + 4;
```

Although `FtoC(T + 2) + 4` returns the same result as `((5 * ((T + 2) - 32)) / 9) + 4`, they are not identical. As will we see later, the function call requires the parameter `(T + 2)` to be passed on the stack and a subroutine call will be executed.

2.10.1 Function Declarations

Similar to the approach with variables, C differentiates between a function declaration and a function definition. A declaration specifies the syntax (name and input / output parameters), whereas a function definition specifies the actual program to be executed when the function is called. Many C programmers refer to a function declaration as a prototype. Since the C compiler is essentially a one-pass process (not including the preprocessor), a function must be declared (or defined) before it can be called. A function declaration begins with the type (format) of the return parameter. If there is no return parameter, then the type can be either specified as void or left blank. Next comes the function name, followed by the parameter list. In a function declaration we do not have to specify names for the input parameters, just their types. If there are no input parameters, then the type can be either specified as void or left blank. The following examples illustrate that the function declaration specifies the name of the function and the types of the function parameters.

```
//  declaration                     input            output
void Init(void);                 // none             none
char InChar(void);               // none             8-bit
void OutChar(char);              // 8-bit            none
short InSDec(void);              // none             16-bit
void OutSDec(short);             // 16-bit           none
char Max(char, char);            // two 8-bit        8-bit
int EMax(int, int);              // two 32-bit       32-bit
void OutString(char*);           // pointer to 8-bit none
char* alloc(int);                // 32-bit           pointer to 8-
bit
int Exec(void(*fnctPt)(void));  // function pointer 32-bit
```

Normally we place function declarations in the header file. We should add comments that explain what the function does.

```
void InitUART(void);  // Initialize 38400 bits/sec
char InChar(void);    // Reads in a character
void OutChar(char);   // Output a character
char UpCase(char);    // Converts lower case character to upper case
void InString(char*, unsigned int); // Reads in a string of max length
```

To illustrate some options when declaring functions, alternative declarations of these same five functions are given below:

```
InitUART();
char InChar();
void OutChar(char letter);
char UpCase(char letter);
InString(char* pt, unsigned int maxSize);
```

Sometimes we wish to call a function that will be defined in another module. If we define a function as external, software in this file can call the function (because the compiler knows everything about the function except where it is), and the linker will resolve the unknown address later when the object codes are linked.

```
extern void InitUART(void);
extern char InChar(void);
extern void OutChar(char);
extern char UpCase(char);
extern void InString(char*, unsigned int);
```

One of the powerful features of C is to define pointers to functions. A simple example follows:

```
// pointer to a function with input and output
int (*fp)(int);

int fun1(int input)
{
  return (input + 1);    // this adds 1
}

int fun2(int input)
{
  return (input + 2);    // this adds 2
}

void Setup(void)
{
  int data;

  fp = &fun1;       // fp points to fun1
  data = (*fp)(5); // data=fun1(5);
  fp = &fun2;       // fp points to fun2
  data = (*fp)(5); // data=fun2(5);
}
```

Listing 2.85 – Example of a function pointer

The declaration of `fp` looks a bit complicated because it has two sets of parentheses and an asterisk. In fact, it declares `fp` to be a pointer to any function that takes one integer argument and returns an integer. In other words, the line `int (*fp)(int);` doesn't define the function. As in other declarations, the asterisk identifies the following name as a pointer. Therefore, this declaration reads "`fp` is a pointer to a function with a 32-bit signed input parameter that returns a 32-bit signed output parameter." Using the term object loosely, the asterisk may be read in its usual way as "object at." Thus we could also read this declaration as "the object at `fp` is a function with an `int` input that returns an `int`."

So why the first set of parentheses? By now you have noticed that in C declarations follow the same syntax as references to the declared objects. Since the asterisk and parentheses (after the name) are expression operators, an evaluation precedence is associated with them. In C, parentheses following a name are associated with the name before the preceding asterisk is applied to the result. Therefore,

```
int *fp(int);
```

would be taken as

```
int *(fp(int));
```

saying that `fp` is a function returning a pointer to an integer, which is not at all like the declaration in Listing 2.85.

2.10.2 Function Definitions

The second way to declare a function is to fully describe it; that is, to define it. Obviously every function must be defined somewhere. So if we organize our source code in a bottom-up fashion, we would place the lowest level functions first, followed by the function that calls these low level functions. It is possible to define large projects in C without ever using a standard declaration (function prototype). On the other hand, most programmers like the top-down approach illustrated in the following example. This example includes three modules: the LCD interface, the COP functions, and some Timer routines. Notice the function names are chosen to reflect the module in which they are defined. If you are a C++ programmer, consider the similarities between this C function call **`LCD_Clear`**`()` and a C++ LCD class and a call to a member function **`LCD.Clear`**`()`. The `*.h` files contain function declarations and the `*.c` files contain the implementations.

```
#include "LCD.h"
#include "COP.h"
#include "Timer.h"

void main(void)
{
  char letter;
  short n = 0;

  COP_Init();
```

```
  LCD_Init();
  Timer_Init()
  LCD_String("This is a LCD");
  Timer_MsWait(1000);
  LCD_Clear();
  letter = 'a' - 1;
  while(1)
  {
    if (letter == 'z')
      letter = 'a';
    else
      letter++;
    LCD_PutChar(letter);
    Timer_MsWait(250);
    if (++n == 16)
    {
      n = 0;
      LCD_Clear();
    }
  }
}
```

Listing 2.86 – Modular approach to software development

C function definitions have the following form:

```
return_type Name(parameter list)
{
  Compound Statement
}
```

Just like the function declaration, we begin the definition with the `return_type`, which is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, we can use **`void`** or leave it blank. `Name` is the name of the function. The `parameter list` is a list of zero or more names for the arguments that will be received by the function when it is called. The parameter list is also known as the *formal* parameters of the function. When a function is invoked, you pass a value to each parameter. This value is referred to as an *actual* parameter or argument. Both the type and name of each input formal parameter is required. MX passes the input parameters from left to right on the stack. If the last parameter has a simple type, it is not pushed but passed in a register. Function results are returned in registers, except if the function returns a result larger than 32 bits. Functions returning a result larger than 32 bits are called with an additional parameter. This parameter is the address where the result should get copied.

Since there is no way in C to declare strings, we cannot declare formal arguments as strings, but we can declare them as character pointers or arrays. In fact, C does not recognize strings, but arrays of characters. The string notation is merely a shorthand way of writing a constant array of characters.

Furthermore, since an unsubscripted array name yields the array's address and since arguments are passed by value, an array argument is effectively a pointer to the array. It follows that the formal argument declarations `arg[]` and `*arg` are really equivalent. The compiler takes both as pointer declarations. Array dimensions in argument declarations are ignored by the compiler since the function has no control over the size of arrays whose addresses are passed to it. It must either assume an array's size, receive its size as another argument, or obtain it elsewhere.

The last, and most important, part of the function definition above is *`Compound Statement`*. This is where the action occurs. Since compound statements may contain local declarations, simple statements, and other compound statements, it follows that functions may implement algorithms of any complexity and may be written in a structured style. Nesting of compound statements is permitted without limit.

As an example of a function definition consider a function named `add3` which takes three input arguments:

```c
int add3(int z1, int z2, int z3)
{
  int y;
  y = z1 + z2 + z3;
  return y;
}
```

Listing 2.87 – Example function with 3 inputs and one output

2.10.3 Function Calls

A function is called by writing its name followed by a parenthesized list of argument expressions. The general form is:

```
Name(parameter list)
```

where `Name` is the name of the function to be called. The ***`parameter list`*** specifies the particular input parameters used in this call. Each input parameter is in fact an expression. It may be as simple as a variable name or a constant, or it may be arbitrarily complex, including perhaps other function calls. Whatever the case, the resulting value is pushed onto the stack where it is passed to the called function.

C programs evaluate arguments in any order, but push them onto the stack in the order left to right. MX allocates the stack space for the parameters at the start of the code that will make the function call. Then the values are stored into the pre-allocated stack position before it calls the function. The input parameters are removed from the stack at the end of the function. The return parameter is generally located in a register.

When the called function receives control, it refers to the first actual argument using the name of the first formal argument. The second formal argument refers to the second actual argument, and so on. In other words, actual and formal arguments are matched by position in their respective lists. Extreme care must be taken to ensure that these lists have the same number and type of arguments.

Function calls can appear in expressions. Since expressions are legal statements, and since expressions may consist of only a function call, it follows that a function call may be written as a complete statement. Thus the statement:

```
add3(--counter, time + 5, 3);
```

is legal. It calls **`add3`**`()`, passing it three arguments: `--counter`, `time + 5`, and `3`. Since this call is not part of a larger expression, the value that **`add3`**`()` returns will be ignored.

As a better example, consider:

```
y = add3(--counter, time + 5, 3);
```

which is also an expression. It calls `add3()` with the same arguments as before but this time it assigns the returned value to `y`. It is a mistake to use an assignment statement like the above with a function that does not return an output parameter.

The ability to pass one function a pointer to another function is a very powerful feature of the C language. It enables a function to call any of several other functions with the caller determining which subordinate function is to be called.

```
int fun1(int input)
{
  return (input + 1);     // this adds 1
}

int fun2(int input)
{
  return (input + 2);     // this adds 2
}

int execute(int (*fp)(int))
{
  int data;

  data = (*fp)(5);
  return data;
}

void main(void)
{
  int result;

  result = execute(&fun1); // result = fun1(5);
  result = execute(&fun2); // result = fun2(5);
}
```

Listing 2.88 – Example of passing a function pointer

Notice that `fp` is declared to be a function pointer. Also, notice that the designated function is called by writing an expression of the same form as the declaration.

2.10.4 Argument Passing

Let us take a closer look at the matter of argument passing. With respect to the method by which arguments are passed, two types of subroutine calls are used in programming languages – *call by reference* and *call by value*.

The *call by reference* method passes arguments in such a way that references to the formal arguments become, in effect, references to the actual arguments. In other words, references (pointers) to the actual arguments are passed, instead of copies of the actual arguments themselves. In this scheme, assignment statements have implied side effects on the actual arguments; that is, variables passed to a function are affected by changes to the formal arguments. Sometimes side effects are beneficial, and sometimes they are not. Since C supports only one formal output parameter, we can implement additional output parameters using call by reference. In this way the function can return parameters back using the reference. The function **`FIFO_Get`**, shown below, returns two parameters. The return parameter is an integer specifying whether or not the request was successful, and the actual data removed from the queue is returned via the call by reference. The calling program **`InChar`** passes the address of its local variable data. The assignment statement `*datapt = FIFO[GetI++];` within **`FIFO_Get`** will store the return parameter into a local variable of **`InChar`**. Normally **`FIFO_Get`** does not have the scope to access local variables of **`InChar`**, but in this case **`InChar`** explicitly granted that right by passing a pointer to **`FIFO_Get`**.

```
int FIFO_Get(char *datapt)
{
  if (Size == 0)
    return 0;              // Empty if Size == 0
  *datapt = FIFO[GetI++];  // Get data out of FIFO
  Size--;
  if (GetI == FIFO_SIZE)
    GetI = 0;              // Wrap
  return -1;               // Successful
}

char InChar(void)
{
  char data;
  while (!FIFO_Get(&data));
  return data;
}
```

Listing 2.89 – Multiple output parameters using call by reference

When we use the *call by value* scheme, the values, not references, are passed to functions. With call by value, copies are made of the parameters. Within a called function, references to formal arguments see copied values on the stack, instead of the original objects from which they were taken. At the time when the computer is executing within `FIFO_Put()` of the example below, there will be three separate and distinct copies of the 0x41 data (`main`, `OutChar` and `FIFO_Put`).

```
int FIFO_Put(char data)
{
  if (Size == FIFO_SIZE)
    return 0;                // Failed, FIFO was full

  Size++;
  FIFO[PutI++] = data;       // Put data into FIFO
  if (PutI == FIFO_SIZE)
    PutI = 0;                // Wrap
  return -1;                 // Successful
}

void OutChar(char data)
{
  while (!FIFO_Put(data));
  UART2_C2 = 0xAC;
}

void main(void)
{
  char data = 0x41;

  OutChar(data);
}
```

Listing 2.90 – Call by value passes a copy of the data

The most important point to remember about passing arguments by value in C is that there is no connection between an actual argument and its source. Changes to the arguments made within a function have no effect whatsoever on the objects that might have supplied their values. They can be changed at will and their sources will not be affected in any way. This removes a burden of concern for a programmer since they may use arguments as local variables without side effects. It also avoids the need to define temporary variables just to prevent side effects.

It is precisely because C uses call by value that we can pass expressions, not just variables, as arguments. The value of an expression can be copied, but it cannot be referenced since it has no existence in global memory. Therefore, call by value adds important generality to the language.

Although the C language uses the call by value technique, it is still possible to write functions that have side effects; but it must be done deliberately. This is possible because of C's ability to handle expressions that yield addresses. Since any expression is a valid argument, addresses can be passed to functions.

Since expressions may include assignment, increment, and decrement operators, it is possible for argument expressions to affect the values of arguments lying to their left or right (recall that C evaluates argument expressions in any order). Consider, for example:

```
func(y = x + 1, 2 * y);
```

If the arguments are evaluated left to right, then the first argument has the value `x+1` and the second argument has the value `2*(x+1)`, but if the arguments are evaluated right to left, then the first argument has the value `x+1` and the second argument has the value `2*y` (whatever that may be). The order of evaluation of arguments is an example of *unspecified behaviour* in the C language. This is only an issue when the arguments consist of expressions that modify and use the same object. The safe way to write the function call is:

```
y = x + 1;
func(y, 2 * y);
```

It is the programmer's responsibility to ensure that the parameters passed match the formal arguments in the function's definition. Some mistakes will be caught as syntax errors by the compiler, but this mistake is a common and troublesome problem for all C programmers.

Occasionally, the need arises to write functions that work with a variable number of arguments. An example is **`printf`**`()` in the ANSI C library. To write a function with a variable number of arguments, you need to consult a reference on advanced C programming.

2.10.5 Private versus Public Functions

For every function definition, MX generates an assembler directive declaring the function's name to be *public*. This means that every C function is a potential entry point and so can be accessed externally. One way to create private / public functions is to control which functions have declarations. Consider again the main program in Listing 2.86 shown earlier. Let's look inside the `Timer.h` and `Timer.c` files. To implement private and public functions we place the function declarations of the *public* functions in the `Timer.h` file.

```
void Timer_Init(void);
void Timer_MsWait(unsigned int time);
```

Listing 2.91 – `Timer.h` header file has public functions

The implementations of all functions are written in the `Timer.c` file. The function **`TimerWait`** is private and can only be called by software inside the `Timer.c` file. We can apply this same approach to private and public global variables. Notice that in this case the global variable, `TimerClock`, is private and cannot be accessed by software outside the `Timer.c` file.

```
static unsigned short TimerClock; // private global

// public function
void Timer_Init(void)
{
  FTM0_MODE |= 0x01;  // Enable timer
  FTM0_SC |= 0x01;    // timer/2 (500ns)
  TimerClock = 2000;  // 2000 counts per ms
}

// private function
static void TimerWait(unsigned short time)
{
  FTM0_C5V = FTM0_CNT + TimerClock;  // 1.00ms wait
  FTM0_CnSC(5) &= 0x80;              // clear C5F
  while ((FTM0_CnSC(5) & 0x80) == 0);
}

// public function
void Timer_MsWait(unsigned short time)
{
  for (; time > 0; time--)
    TimerWait(TimerClock); // 1.00ms wait
}
```

Listing 2.92 – `Timer.c` implementation file defines all functions

2.10.6 Finite State Machine using Function Pointers

Now that we have seen how to declare, initialize and access function pointers, we can create very flexible finite state machines. In the finite state machine presented in Listing 2.74 and Listing 2.76, the output was a simple number that is written to the output port. In the next example, we will implement the exact same FSM, but in a way that supports much more flexibility in the operations that each state performs. In fact, we will define a general C function to be executed at each state. In this implementation the functions perform the same output as the previous FSM.

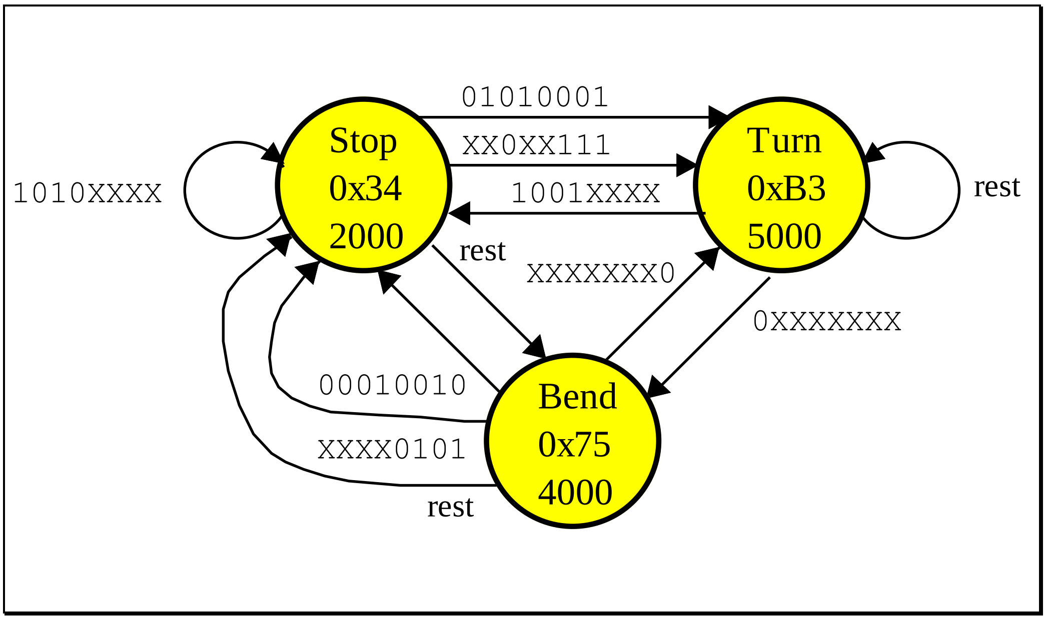


Figure 2.49 – Finite state machine

Compare the following implementation to Listing 2.74, and see that the `unsigned char out;` constant is replaced with a `void (*cmdPt)(void);` function pointer. The three general functions `DoStop()`, `DoTurn()` and `DoBend()` are also added.

```c
typedef const struct State
{
  void (*cmdPt)(void);          // function to execute
  unsigned short wait;          // Time (bus cycles) to wait
  unsigned char andMask[4];
  unsigned char equMask[4];
  const struct State *next[4]; // Next states
} TState;

typedef TState* PState;

#define Stop &FSM[0]
#define Turn &FSM[1]
#define Bend &FSM[2]

void DoStop(void)
{
  GPIOA_PDOR = 0x34;
}

void DoTurn(void)
{
  GPIOA_PDOR = 0xB3;
}

void DoBend(void)
{
  GPIOA_PDOR = 0x75;
}

TState FSM[3] =
{
  {
    &DoStop, 2000,    // stop 1 ms
    {0xFF,   0xF0,    0x27,    0x00},
    {0x51,   0xA0,    0x07,    0x00},
    {Turn,   Stop,    Turn,    Bend}
  },
  {
    &DoTurn, 5000,    // turn 2.5 ms
    {0x80,   0xF0,    0x00,    0x00},
    {0x00,   0x90,    0x00,    0x00},
    {Bend,   Stop,    Turn,    Turn}
  },
  {
    &DoBend, 4000,    // bend 2 ms
    {0xFF,   0x0F,    0x01,    0x00},
    {0x12,   0x05,    0x00,    0x00},
    {Stop,   Stop,    Turn,    Stop}
  }
};
```

Listing 2.93 – Linked finite state machine structure stored in Flash

Compare the following implementation to Listing 2.76, and see that the `GPIOA_PDOR = pt->out;` assignment is replaced with a `(*pt->cmdPt)();` function call. In this way, the appropriate function **`DoStop`**`()`, **`DoTurn`**`()` or **`DoBend`**`()` will be called.

```
void Control(void)
{
  PState pt;
  unsigned char input;
  unsigned short startTime;

  FTM0_MODE |= 0x01;        // Enable timer
  FTM0_SC |= 0x01;          // timer/2 (500ns)
  GPIOA_PDDR = 0x000000FF; // PortA bits 7-0 are outputs
  GPIOB_PDDR = 0x00000000; // PortB bits 7-0 are inputs
  pt = stop;                // Initial State

  while(1)
  {
    // 1) execute function
    (*pt->cmdPt)();
    // Time (500 ns each) to wait
    startTime = FTM0_CNT;
    // 2) wait
    while ((FTM0_CNT - startTime) <= pt->wait);
    // 3) input
    input = GPIOB_PDIR;
    for (int i = 0; i < 4; i++)
      if ((input & pt->andMask[i]) == pt->equMask[i])
      {
        // 4) next depends on input
        pt = pt->next[i];
        i = 4;
      }
  }
}
```

Listing 2.94 – Finite state machine controller for K64

2.10.7 Linked List Interpreter using Function Pointers

In the next example, function pointers are stored in a linked list. An interpreter accepts ASCII input from a keyboard and scans the list for a match. In this implementation, each node in the linked list has a function to be executed when the operator types the corresponding letter. The linked list `LL` has three nodes. Each node has a letter, a function and a link to the next node.

```
// Linked List Interpreter
typedef const struct Node
{
  unsigned char letter;
  void (*fnctPt)(void);
  const struct Node *next;
} TNode;

typedef TNode* PNode;

void CommandA(void)
{
  OutString("\nExecuting Command a");
}

void CommandB(void)
{
  OutString("\nExecuting Command b");
}

void CommandC(void)
{
  OutString("\nExecuting Command c");
}

TNode LL[3] =
{
  {'a', &CommandA, &LL[1]},
  {'b', &CommandB, &LL[2]},
  {'c', &CommandC, NULL}
};
```

```c
void main(void)
{
  PNode pt;
  char string[40];

  UART_Init(); // Enable UART
  OutString("\nEnter a single letter followed by <enter>");
  while (1)
  {
    OutString("\n>");
    InString(string, 1);  // first character is interpreted
    pt = &LL[0];          // first node to check
    while (pt)
    {
      if (string[0] == pt->letter)
      {
        pt->fnctPt(); // execute function
        break;        // leave while loop
      }
      else
      {
        pt = pt->next;
        if (pt == 0)
          OutString(" Error");
      }
    }
  }
}
```

Listing 2.95 – Linked list implementation of an interpreter

Compare the syntax of the function call, `(*pt->cmdPt)();`, in Listing 2.94, with the syntax in this example, `pt->fnctPt();`. In the MX compiler, these two expressions both generate code that executes the function.

2.11 Preprocessor Directives

C compilers incorporate a preprocessing phase that alters the source code in various ways before passing it on for compiling. Four capabilities are provided by this facility in C. They are:

- macro processing
- conditional compiling
- inclusion of text from other files
- implementation-dependent features

The preprocessor is controlled by directives which are not part of the C language. Each directive begins with a # character and is written on a line by itself. Only the preprocessor sees these directive lines since it deletes them from the code stream after processing them.

Depending on the compiler, the preprocessor may be a separate program or it may be integrated into the compiler itself. MX has an integrated preprocessor that operates at the front end of its single pass algorithm.

2.11.1 Macro Processing

We use macros for three reasons:

1) To save time we can define a macro for long sequences that we will need to repeat many times.

2) To clarify the meaning of the software we can define a macro giving a symbolic name to a hard-to-understand sequence. The I/O port `#define` macros are good examples of this reason.

3) To make the software easy to change, we can define a macro such that changing the macro definition automatically updates the entire software.

Macros define names which stand for arbitrary strings of text:

```
#define Name CharacterString
```

After such a definition, the preprocessor replaces each occurrence of `Name` (except in string constants and character constants) in the source text with `CharacterString`. As C implements this facility, the term macro is misleading, since parameterized substitutions are not supported. That is, `CharacterString` does not change from one substitution to another according to parameters provided with `Name` in the source text; it is simply a literal replacement of one set of characters with another.

C accepts macro definitions only at the global level.

The `Name` part of a macro definition must conform to the standard C naming conventions as described earlier. `CharacterString` begins with the first printable character following `Name` and continues through to the last printable character of the line or until a comment is reached.

If `CharacterString` is missing, occurrences of `Name` are simply squeezed out of the text. Name matching is based on the whole name (up to 8 characters); part of a name will not match. Thus the directive:

```
#define size 10
```

will change:

```
short data[size];
```

into:

```
short data[10];
```

but it will have no effect on:

```
short data[size1];
```

Replacement is also performed on subsequent `#define` directives, so that new symbols may be defined in terms of preceding ones.

2.214

The most common use of `#define` directives is to give meaningful names to constants; i.e. to define so-called *manifest constants*. The use of manifest constants in programs helps to ensure that code is portable by isolating the definition of these elements in a single header file, where they need to be changed only once.

However, we may replace a name with anything at all: a commonly occurring expression or sequence of statements for instance. To disable interrupts during a critical section we could implement:

```
#define ENTER_CRTITICAL() __asm("CPSID f");
#define EXIT_CRITICAL()   __asm("CPSIE f");

void function(void)
{
  ...
  ENTER_CRITICAL; // make atomic, entering critical section
  // we have exclusive access to global variables
  ...
  EXIT_CRITICAL; // exit critical section
}
```

Listing 2.96 – Example of `#define`

There is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code (but if it does not, you may get error messages from the C compiler when you use the macro).

2.11.2 Conditional Compiling

The preprocessing feature lets us designate parts of a program which may or may not be compiled depending on whether or not certain symbols have been defined. In this way it is possible to write into a program optional features which are chosen for inclusion or exclusion by simply adding or removing `#define` directives at the beginning of the program.

When the preprocessor encounters

```
#ifdef Name
```

it looks to see if the designated name has been defined. If not, it throws away the following source lines until it finds a matching

```
#else
```

or

```
#endif
```

directive. The `#endif` directive delimits the section of text controlled by `#ifdef`, and the `#else` directive permits us to split conditional text into true and false parts. The first part (`#ifdef...#else`) is compiled only if the designated name is defined, and the second (`#else...#endif`) only if it is not defined.

The converse of `#ifdef` is the

```
#ifndef Name
```

directive. This directive also takes matching `#else` and `#ifndef` directives. In this case, however, if the designated name is not defined, then the first (`#ifndef...#else`) or only (`#ifndef...#endif`) section of text is compiled; otherwise, the second (`#else...#endif`), if present, is compiled.

Nesting of these directives is allowed; and there is no limit on the depth of nesting. It is possible, for instance, to write something like

```
#ifdef ABC
... // ABC
#ifndef DEF
... // ABC and not DEF
#else
... // ABC and DEF
#endif
... // ABC
#else
... // not ABC
#ifdef HIJ
... // not ABC but HIJ
#endif
... // not ABC
#endif
```

Listing 2.97 – Examples on conditional compilation

where the ellipses represent conditionally compiled code, and the comments indicate the conditions under which the various sections of code are compiled.

A good application of conditional compilation is inserting debugging code. In this example the only purpose of writing to `PORTC` is to assist in performance debugging. Once the system is debugged, we can remove all the debugging code, simply by deleting the `#define Debug` line.

```
#define Debug

int Sub(int j)
{
  int i;

#ifdef Debug
  GPIOC_PSOR = 0x01; // PC0 set when Sub is entered
#endif
  i = j + 1;
#ifdef Debug
  GPIOC_PCOR = 0x01; // PC0 cleared when Sub is exited
#endif
  return i;
}
```

```c
void ProgA(void)
{
  int i;

#ifdef Debug
  GPIOC_PSOR = 0x02; // PC1 set when ProgA is entered
#endif
  i = Sub(5);
  while (1)
  {
    i = Sub(i);
  }
}

void ProgB(void)
{
  int i;

  i = 6;
  ...
#ifdef Debug
  GPIOC_PCOR = 0x02; // PC1 cleared when ProgB is exited
#endif
}
```

Listing 2.98 – Conditional compilation can help in debugging code

2.11.3 Including Other Source Files

The preprocessor also recognizes directives to include source code from other files. The two directives

```c
#include <Filename>
#include "Filename"
```

cause a designated file to be read as input to the compiler. The difference between these two directives is where the compiler looks for the file. The `<Filename>` version will search for the file in the standard include directory, while the `"Filename"` version will search for the file in the same directory as the original source file. The preprocessor replaces these directives with the contents of the designated files. When the files are exhausted, normal processing resumes.

`Filename` follows the normal PC file specification format, including drive, path, filename, and extension.

2.11.4 Implementation-Dependent Features

The `#pragma` directive is used to instruct the compiler to use pragmatic or implementation-dependent features. For example, in the GNU Compiler Collection (GCC) for ARM® processors, you can change the maximum alignment of members of structures and unions using the `#pragma pack` directive:

```
#pragma pack(push)
#pragma pack(1)

typedef union
{
  uint8_t bytes[5];
  struct
  {
    uint8_t byte1;
    uint8_t byte2;
    uint8_t byte3;
    uint8_t byte4;
    uint8_t byte5;
  } packetStruct;
} TPacket;
```

The GCC compiler will then ensure that `byte1`, `byte2`, etc. are contiguous in memory, rather than aligned on 32-bit boundaries. In this way, we can use the union to access the same 5 bytes of memory via the array or by a unique name.

Although GCC supports several types of pragmas (primarily in order to compile code originally written for other compilers), it does not recommend the use of pragmas for functions – instead function attributes are introduced by the `__attribute__` keyword on a declaration, followed by an attribute specification inside double parentheses.

For example, the function attribute `interrupt` is used to indicate that the specified function is an interrupt service routine. To declare an ISR for a UART, you would use:

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

Listing 2.99 – Interrupt service routine as specified using GCC

2.12 Assembly Language Programming

One of the main reasons for using the C language is to achieve portability. But there are occasional situations in which it is necessary to sacrifice portability in order to gain full access to the operating system or to the hardware in order to perform some interface requirement, or to maximize performance in time-sensitive code. If these instances are kept to a minimum and are not replicated in many different programs, the negative effect on portability may be acceptable. There are two approaches to writing assembly language with GCC. The first method inserts assembly instructions directly into a C function using the `__asm(string);` feature. Everything within the `string` statement is assumed to be assembly language code and is sent straight to the output of the compiler exactly as it appears in the input. The second approach is to write an entire file in assembly language, which may include global variables and functions. In MX, we include assembly files by adding them to the project. Entire assembly files can also be assembled separately then linked at a later time to the rest of the program. The simple insertion method is discussed in this section.

2.12.1 How to Insert Single Assembly Instructions

To support this capability, GCC provides for assembly language instructions to be written into C programs anywhere a statement is valid. Since the compiler generates assembly language as output, when it encounters assembly language instructions in the input, it simply copies them directly to the output.

A special directive delimits assembly language code. The following example inserts the assembly language instruction `CPSID f` (disable interrupts) into the program at that point.

```
__asm("CPSID f");
```

A better way is to `#define` macros:

```
#define INTR_OFF() __asm("CPSID f")
#define INTR_ON()  __asm("CPSIE f")
```

The following function runs with interrupts disabled.

```
void FIFO_Init(void)
{
  INTR_OFF();       // make atomic, entering critical section
  PutI=GetI=Size=0;// Empty when Size == 0
  INTR_ON();        // end critical section
}
```

Listing 2.100 – Example of an assembly language macro

Of course, to make use of the `__asm` feature, we must let the compiler know about the C variables modified by the instructions, the C expressions read by the instructions, and the registers or other values that are changed by the instructions. We also need to know how the compiler uses the CPU registers, how functions are called, and how the operating system and hardware works. It will certainly cause a programming error if your embedded assembly modifies the stack pointer, `SP`, for example.

In GCC you can access a global or local variable directly using just its name:

```
int Time;

void Add1Time(void)
{
   __asm (\
    "ldr  r3, %[input]\n\t"\
    "adds r2, r3, #1\n\t"\
    "str  r3, %[input]\n\t"\
    ::[input] "m" (Time)  \
    : "r2", "r3");
}
```

Listing 2.101 – Assembly language access to a global variable

For more information on this feature, see:

https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm

2.13 Hardware Abstraction Layers

A Hardware Abstraction Layer (HAL) is a set of routines in software that provide programs with access to hardware resources through programming interfaces. HALs allow programmers to write device-independent applications by providing standard software calls to hardware.

For ARM® Cortex®-based processors, there is a HAL called the Cortex® Microcontroller Software Interface Standard (CMSIS).

To quote from ARM:

> *The CMSIS is a vendor-independent hardware abstraction layer for microcontrollers that are based on Arm® Cortex® processors. It defines generic tool interfaces and enables consistent device support. Its software interfaces simplify software re-use, reduce the learning curve for microcontroller developers, and improve time to market for new devices.*
>
> *The CMSIS provides interfaces to processor and peripherals, real-time operating systems, and middleware components. It includes a delivery mechanism for devices, boards, and software and enables the combination of software components from multiple vendors.*

CMSIS has been created to help the industry in standardization. It enables consistent software layers and device support across a wide range of development tools and microcontrollers. CMSIS is not a huge software layer that introduces overhead and does not define standard peripherals. The silicon industry can therefore support the wide variations of Cortex®-M processor-based devices with this common standard.

This means that if you utilise the CMSIS HAL for various low-level functions, such as setting up and responding to interrupts or using the built-in SysTick, then your software will have a high degree of portability across the range of ARM® Cortex®-based microcontrollers.

There are many individual HALs that support special features of the ARM® Cortex® architecture. For example, there is a HAL that supports the special digital signal processing instructions of the Cortex® M4/M7/M33/M35P – we can use this on the NXP K64 chips.

For more information, see:

https://developer.arm.com/tools-and-software/embedded/cmsis

5 Interrupts

Contents

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 K64 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.

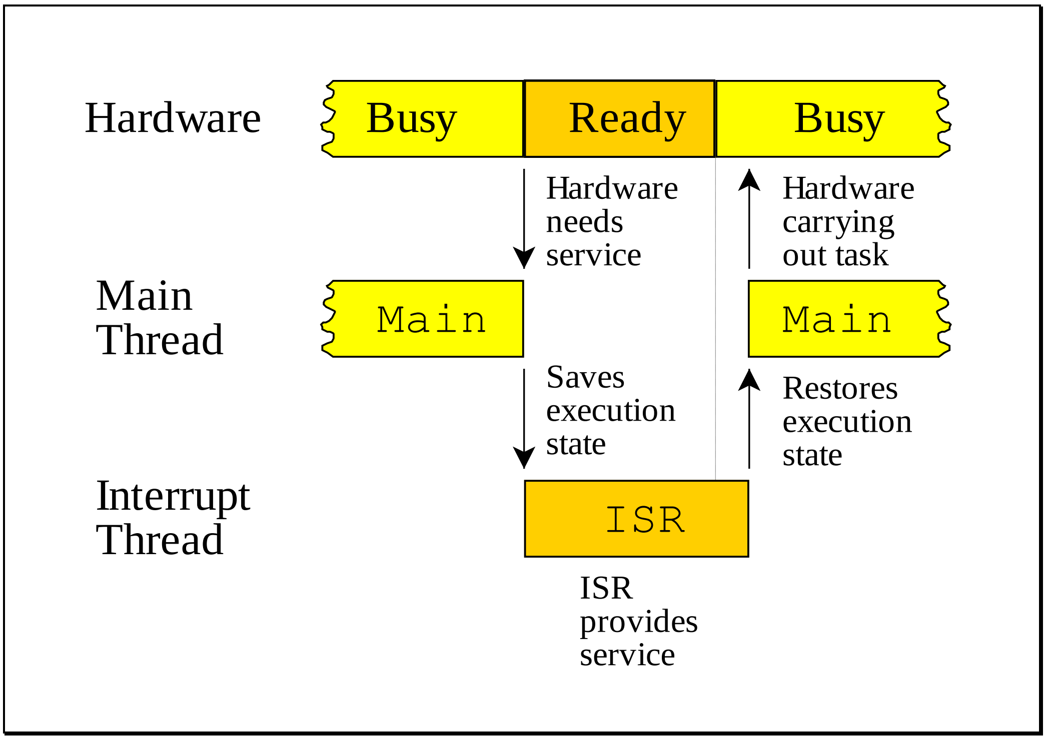
Using interrupts requires that we first understand how a CPU processes an interrupt so that we can configure our software to take advantage of them.

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 K64 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.

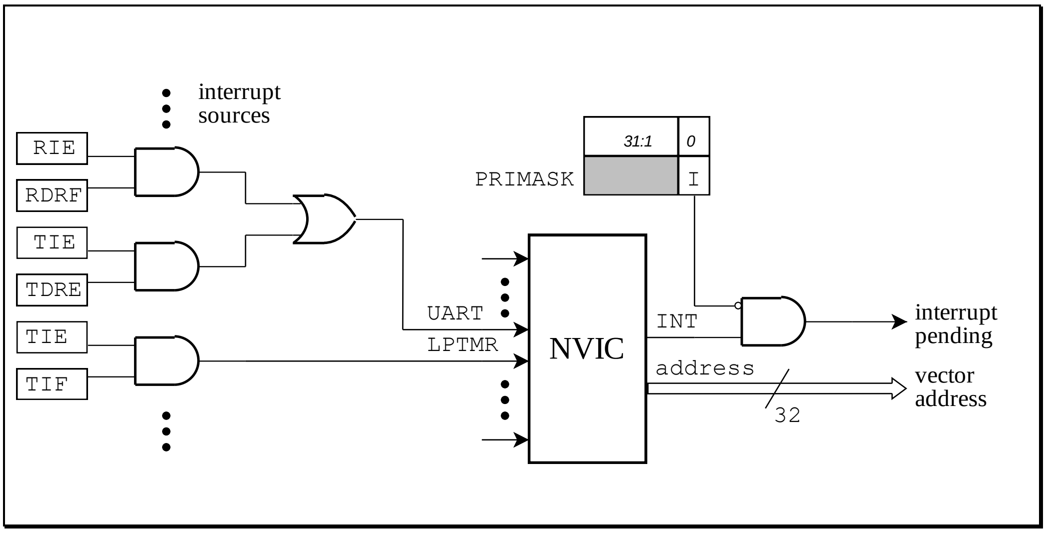


Figure 5.2

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 (UART0->C2 & UART_C2_RIE_MASK)
{
  // Clear RDRF flag by reading the status register
  if (UART0->S1 & UART_S1_RDRF_MASK)
    // Do something with the received byte
    ...
}

// Transmit a character
if (UART0->C2 & UART_C2_TIE_MASK)
{
  // Clear TDRE flag by reading the status register
  if (UART0->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.

Memory Address	Vectors	Exception Number
0x0000_03FC	IRQ #239	255
	⋮	
0x0000_0048	IRQ #2	18
0x0000_0044	IRQ #1	17
0x0000_0040	IRQ #0	16
0x0000_003C	SysTick	15
0x0000_0038	PendSV	14
0x0000_0034	Reserved	13
0x0000_0030	Debug Monitor	12
0x0000_002C	SVC	11
0x0000_0028	Reserved	10
0x0000_0024	Reserved	9
0x0000_0020	Reserved	8
0x0000_001C	Reserved	7
0x0000_0018	Usage Fault	6
0x0000_0014	Bus Fault	5
0x0000_0010	MemManage Fault	4
0x0000_000C	HardFault	3
0x0000_0008	NMI	2
0x0000_0004	Reset	1
0x0000_0000	Initial value of SP	0

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:

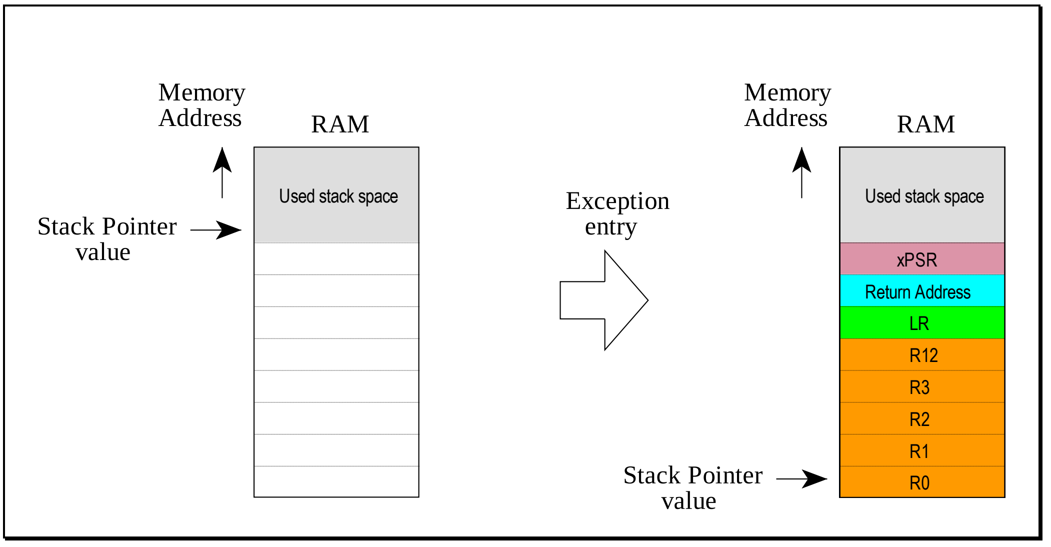


Figure 5.4 – Exception Stack Frame on Entry to Interrupts

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 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.

5.4.2 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 (that were placed there 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.

5.4.1 Specifying an ISR Address in the Vector Table

MCUXpresso declares a vector table in the file `startup_mk64f12.c` in the `startup` folder. The vector table is declared as an array of pointers to functions that take `void` and return `void` (ISRs are never called by a program, so they don't take arguments, and they don't return values). The array lists the exception handlers in numerically ascending order according to ARM's NVIC documentation. The linker will place the array of function pointers at the memory location corresponding to the actual vector table, which starts at 0x00000000. An extract from the vector table as declared in MCUXpresso is shown below:

```
...
UART0_LON_IRQHandler,     // 46 : UART0 LON interrupt
UART0_RX_TX_IRQHandler,   // 47 : UART0 Receive/Transmit interrupt
UART0_ERR_IRQHandler,     // 48 : UART0 Error interrupt
UART1_RX_TX_IRQHandler,   // 49 : UART1 Receive/Transmit interrupt
UART1_ERR_IRQHandler,     // 50 : UART1 Error interrupt
...
```

Listing 5.2 – Vector Table Extract Showing Interrupt Handlers

The vector number for a particular interrupt source is given in the comment, along with its name, which are documented in Table 3-5 of the K64 Sub-Family Reference Manual.

Since MCUXpresso has already declared a vector table, we need a way of overwriting the pre-declared interrupt handler with our own. If you look in the `startup_mk64f12.c` file, you will see the pre-declared interrupt handlers declared as:

```
...
WEAK void UART0_LON_IRQHandler(void);
WEAK void UART0_RX_TX_IRQHandler(void);
WEAK void UART0_ERR_IRQHandler(void);
WEAK void UART1_RX_TX_IRQHandler(void);
WEAK void UART1_ERR_IRQHandler(void);
...
```

`WEAK` is a macro which expands as `__attribute__ ((weak))`. Thus, the pre-declared interrupt handlers can easily be overridden simply by declaring our own "strong" interrupt handler. During linking, our "strong" interrupt handler will override the "weak" one of the same name.

5.5 Enabling and Disabling Interrupts

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

```
static inline uint32_t DisableGlobalIRQ(void)
static inline void EnableGlobalIRQ(uint32_t primask)
```

If you delve into the functions, which in turn call CMSIS provided functions, you will eventually come to some assembly language. The assembly language instruction `cpsie` stands for Change Processor State Interrupt Enable. The `i` parameter refers to the single-bit `PRIMASK` register. Some other implementations of disabling/enabling global interrupts use the `f` parameter which refers to the single-bit "fault mask" register `FAULTMASK`. This register is similar to `PRIMASK`, but it also blocks the HardFault exception.

The above functions are meant to be used in pairs. The return value of **`DisableGlobalIRQ`**`(void)` is the current value of the `PRIMASK` register. This should be used in the call to **`EnableGlobalIRQ`**`(uint32_t primask)`. The reason for this paired nature and the local storage of the `PRIMASK` register is so that paired calls can be nested.

Interrupts are *enabled* by default on reset, so it is necessary to disable them before you embark on peripheral module initialization – so that no inadvertent interrupts occur whilst you set up the hardware. You should then enable interrupts before the main loop of your program:

```
uint32_t priMask;

priMask = DisableGlobalIRQ();
FreedomInit();
EnableGlobalIRQ(priMask);
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 K64 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 K64 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:

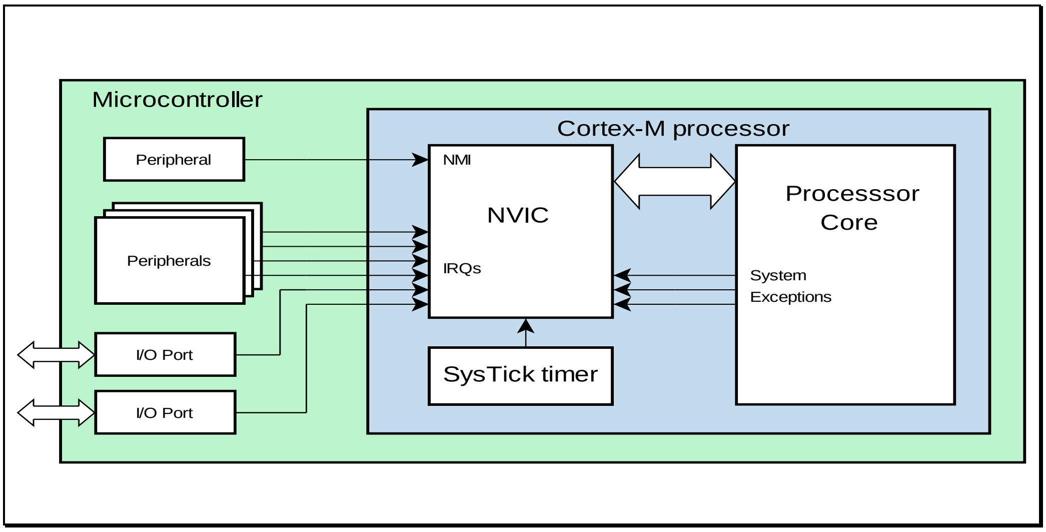


Figure 5.5 – The Nested Vectored Interrupt Controller

The K64 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, `NVICISERx`; to clear the enable bit, you need to write to the NVIC's Clear Enable Register `NVICICERx`. In this way, enabling or disabling an interrupt will not affect other interrupt enable states. The `NVICISERx / NVICICERx` registers are 32-bits wide; each bit represents one interrupt input.

The registers use a write-1-to-enable and write-1-to-clear policy, both registers reading back the current enabled state of the corresponding 32 interrupts. As there are more than 32 external interrupts in the Cortex®-M4 processor, there is more than one `NVICISERx` and `NVICICERx` register.

When an interrupt is disabled, interrupt assertion causes the interrupt to become pending, but the interrupt cannot become active. If an interrupt is active when it is disabled, it remains in the active state until this is cleared by a reset or an exception return. Clearing the enable bit prevents any new activation of the associated interrupt.

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.

The registers use a write-one-to-enable and write-one-to-clear policy, and a read of either register returns the current pending state of the corresponding 32 interrupts. Writing 1 to a bit in the Clear-Pending Register has no effect on the execution status of an active interrupt.

Interrupt Active Registers

Active bit status is provided to enable software to determine whether an interrupt is inactive, active, pending, or active and pending The interrupt-active status can be accessed through the Interrupt Active Bit (`NVICIABRx`) registers. Similarly to the pending registers, there is more than one IAB register.

Interrupt Priority-Level Registers

Each interrupt has an associated interrupt priority-level register (IPR). The 16 priority levels in the K64 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.

CMSIS Supporting Functions

CMSIS implements functions to support NVIC operations. For example, we will often be using the following functions when setting up hardware interrupts:

```
void __NVIC_ClearPendingIRQ(IRQn_Type IRQn);
void __NVIC_EnableIRQ(IRQn_Type IRQn);
```

The parameter passed to these functions is the interrupt request number (defined in the `MK64F12.h` file) corresponding to Table 3-5 of the K64 Sub-Family Reference Manual.

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

```
const gpio_pin_config_t PORTD_GPIO_PIN_CONFIG =
{
  .pinDirection = kGPIO_DigitalInput,
  .outputLogic = 0U
};

const port_pin_config_t PORTD_PORT_PIN_CONFIG =
{
  // Internal pull-up/down resistor is disabled
  .pullSelect = kPORT_PullDisable,
  // Slow slew rate is configured
  .slewRate = kPORT_SlowSlewRate,
  // Passive filter is disabled
  .passiveFilterEnable = kPORT_PassiveFilterDisable,
  // Open drain is disabled
  .openDrainEnable = kPORT_OpenDrainDisable,
  // Low drive strength is configured
  .driveStrength = kPORT_LowDriveStrength,
  // Pin is configured as PTB22
  .mux = kPORT_MuxAsGpio,
  // Pin Control Register fields [15:0] are not locked
  .lockRegister = kPORT_UnlockRegister
};

static uint32_t Count, Ack;
```

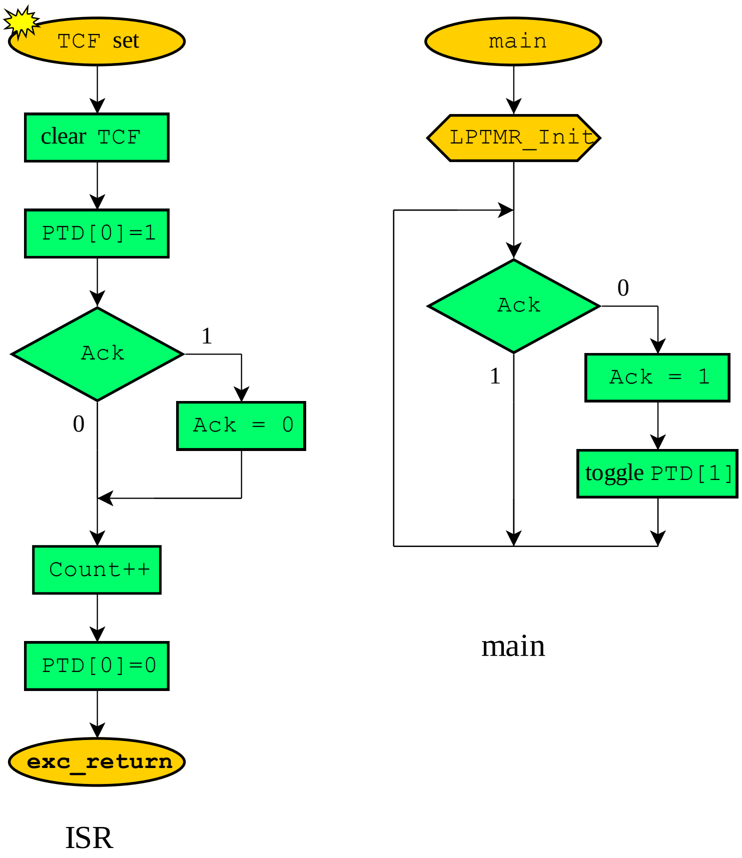
```c
/*! @brief Initializes PORTD bits 1-0.
 *
 */
void PORTD_Init(void)
{
  // Enable clock gate for Port D to enable pin routing
  CLOCK_EnableClock(kCLOCK_PortD);

  // Set up PORTD bits 0-1
  for (uint32_t bitNb = 0; bitNb < 2; bitNb++)
  {
    // Initialize GPIO functionality
    // (set the data direction and output default value)
    GPIO_PinInit(GPIOD, bitNb, &PORTD_GPIO_PIN_CONFIG);
    // Set up pin configuration
    PORT_SetPinConfig(PORTD, bitNb, &PORTD_PORT_PIN_CONFIG);
  }
}

/*! @brief Initializes the low power timer.
 *
 */
void LPTMR_Init(void)
{
  // Enable clock gate to LPTMR module
  CLOCK_EnableClock(kCLOCK_Lptmr0);

  // 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 IRQn_Type in MK64F12.h
  // Clear any pending interrupts on port
  NVIC_ClearPendingIRQ(LPTMR0_IRQn);
  // Enable interrupts on port
  NVIC_EnableIRQ(LPTMR0_IRQn);

  //Turn on LPTMR and start counting
  LPTMR0->CSR |= LPTMR_CSR_TEN_MASK;
}
```

5.20

```
/*! @brief Low power timer interrupt handler.
 *
 */
void LPTMR0_IRQHandler(void)
{
  // Acknowledge interrupt, clear interrupt flag
  LPTMR0->CSR |= LPTMR_CSR_TCF_MASK;
  // Set bit 0
  GPIOD->PSOR = 0x00000001;
  // Software handshake - means LPTMR interrupt happened
  if (Ack == 1)
    Ack = 0;
  // Number of interrupts
  Count++;
  // Clear bit 0
  GPIOD->PCOR = 0x00000001;
}

/*!
 * @brief Main function
 */
int main(void)
{
  BOARD_InitPins();
  BOARD_InitBootClocks();
  // Globally disable interrupts while we set up
  uint32_t priMask = DisableGlobalIRQ();
  PORTD_Init();
  LPTMR_Init();
  // Interrupt counter
  Count = 0;
  // Foreground is ready
  Ack = 1;
  // Globally enable interrupts
  EnableGlobalIRQ(priMask);
  for (;;)
  {
    if (Ack == 0)
    {
      Ack = 1;
      // Toggle bit 1
      GPIOD->PTOR = 0x00000002;
    }
  }
}
```

Note the name of the low power timer interrupts service routine, `void LPTMR0_IRQHandler(void)`, is the same as that already listed in the vector table in `startup_mk64f12.c`. The linker will automatically override the "weak" pre-declared ISR with our one. Thus, our ISR is "injected" into the vector table.

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



Flowchart for simple 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

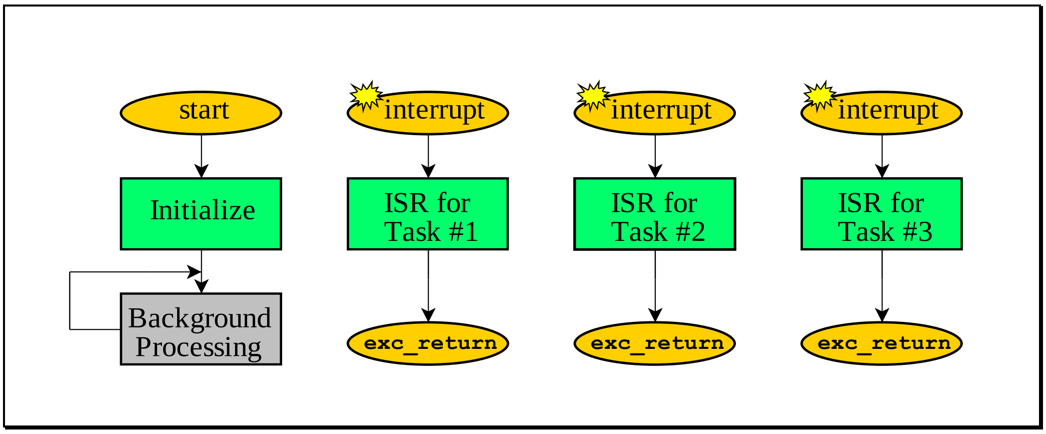


Figure 5.6

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

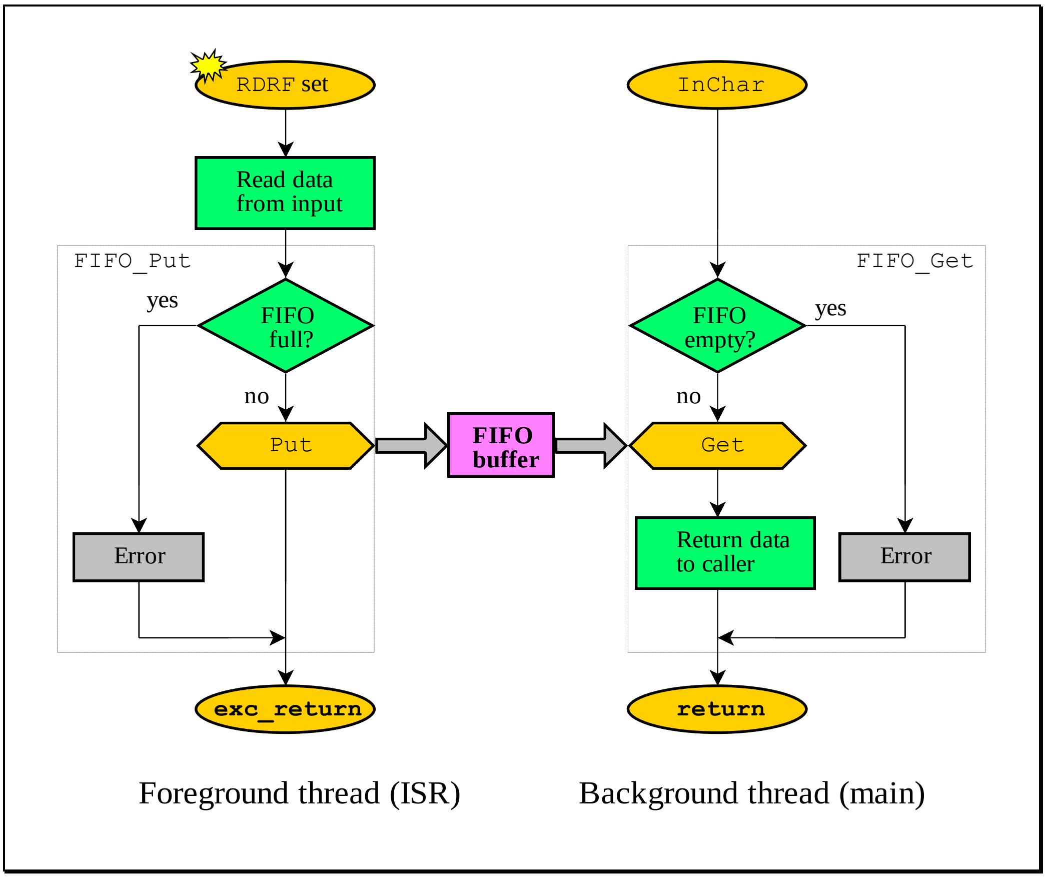


Figure 5.7

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.

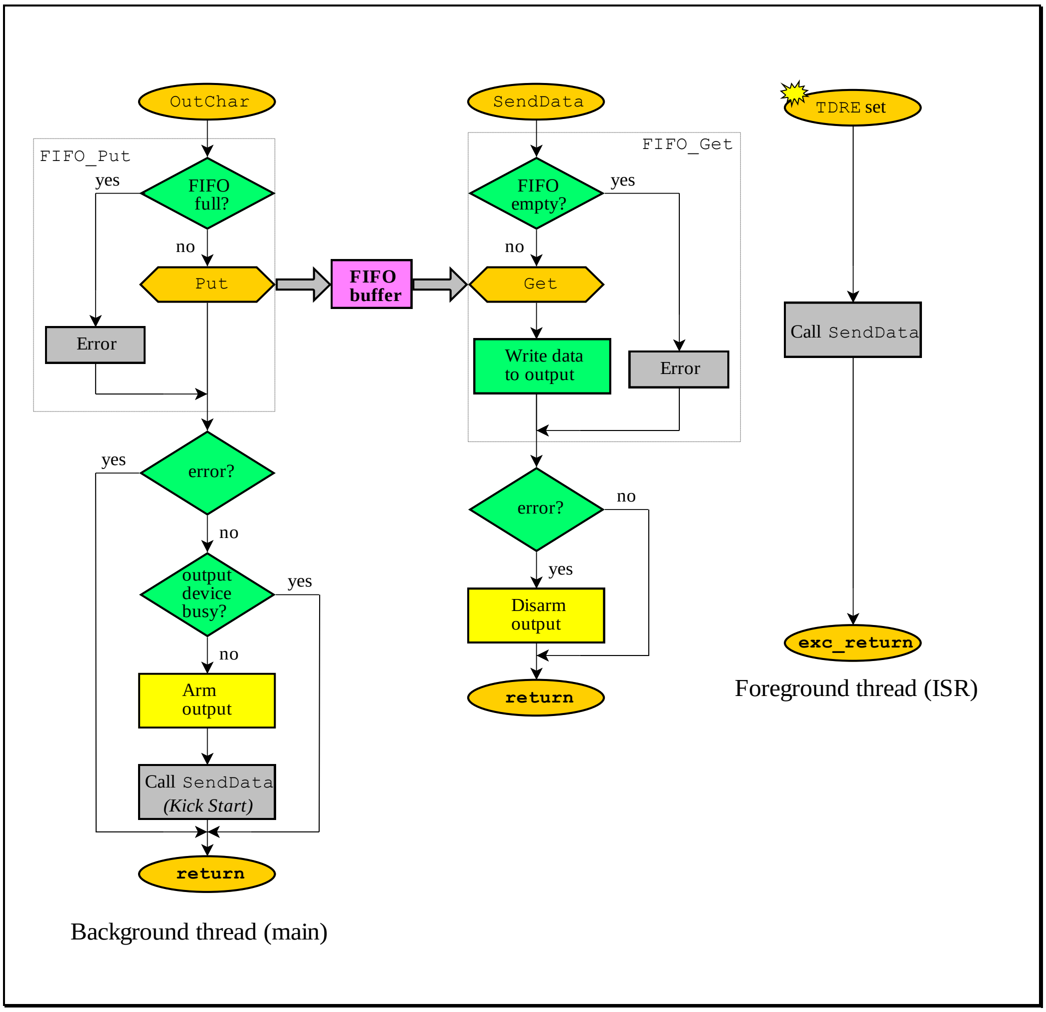
In these situations, the hardware normally provides a mechanism to determine whether or not the output device is busy processing data, such as a flag in a device status register. In these cases, the main work of the ISR should be placed in a separate function (e.g. **`SendData`**) that actually outputs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

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 5.8

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

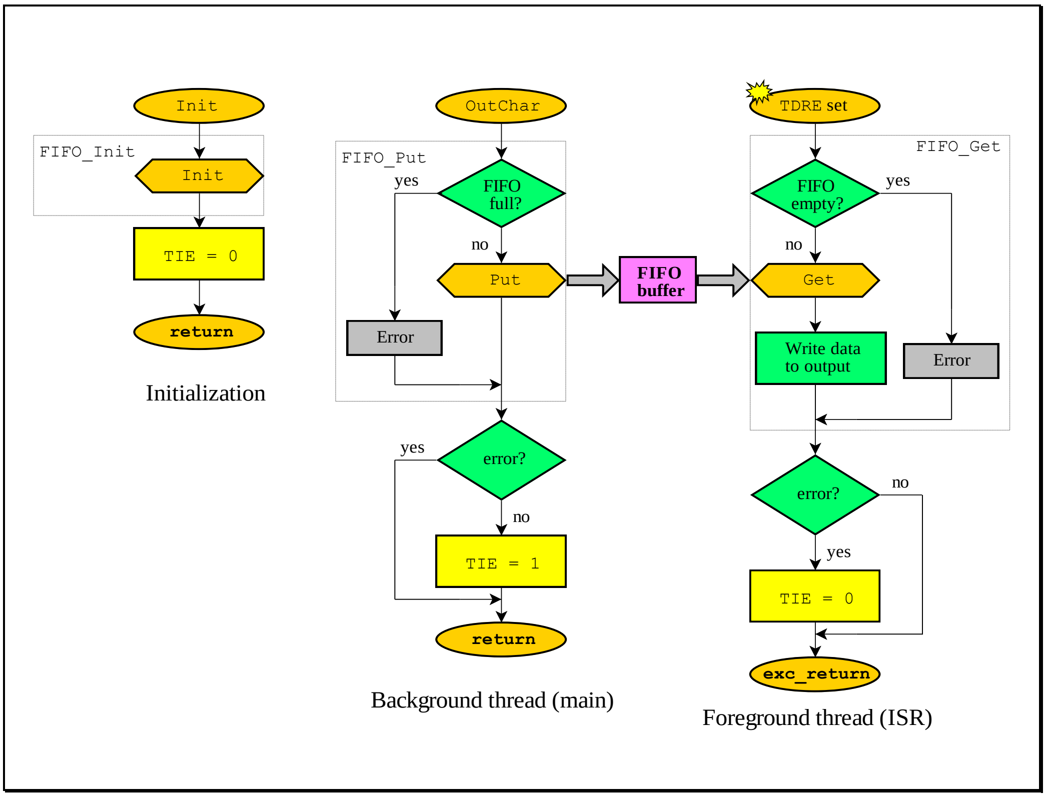


Figure 5.9

5.10 Communicating Between Threads

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.

Communication between threads is accomplished using global variables

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 K64, 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

```
__disable_irq();   // disable interrupts
NbBytes++;         // critical section
__enable_irq();    // 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:

C macros for entering and exiting a critical section

```
// Save status register and disable interrupts
#define EnterCritical() \
 do {\
  uint8_t SR_reg_local;\
   __asm ( \
    "MRS R0, FAULTMASK\n\t" \
    "CPSID f\n\t"            \
    "STRB R0, %[output]"   \
    : [output] "=m" (SR_reg_local)\
    :: "r0");\
   if (++SR_lock == 1u) {\
    SR_reg = SR_reg_local;\
   }\
 } while(0)


// Restore status register
#define ExitCritical() \
 do {\
   if (--SR_lock == 0u) { \
     __asm (                 \
      "LDRB R0, %[input]\n\t"\
      "MSR FAULTMASK, R0;\n\t" \
      ::[input] "m" (SR_reg)  \
      : "r0");                \
   }\
 } 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 as:

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

5.30

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

The advantage of the above macros, rather than using:

```
uint32_t priMask;

priMask = DisableGlobalIRQ();
...
EnableGlobalIRQ(priMask);
```

is that they make the code more readable and portable.

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

ARM®v7-M Architecture Reference Manual, ARM, 2014.

6 Timing Generation and Measurements

Contents

6.2

Introduction

The K64 has several timer modules:

Module	Description
Programmable delay block (PDB)	The PDB provides controllable delays from either an internal or an external trigger, or a programmable interval tick, to the hardware trigger inputs of ADCs and/or generates the interval triggers to DACs, so that the precise timing between ADC conversions and/or DAC updates can be achieved.
FlexTimer modules (FTM)	The FTM is an eight channel timer that supports input capture, output compare, and the generation of PWM signals.
Periodic interrupt timers (PIT)	The PIT module is an array of 4 timers that can be used to raise interrupts and trigger DMA channels.
Low-power timer (LPTMR)	The LPTMR can be configured to operate as a time counter with optional prescaler, or as a pulse counter with optional glitch filter, across all power modes, including the low-leakage modes. It can also continue operating through most system reset events, allowing it to be used as a time of day counter.
Carrier modulator timer (CMT)	The CMT module provides a means to generate the protocol timing and carrier signals for a variety of encoding schemes used in infrared remote controls.
Real-time clock (RTC)	The RTC operates off an independent power supply and 32 kHz crystal oscillator and has a 32-bit seconds counter with a 32-bit alarm.
IEEE 1588 timers	The IEEE 1588 standard provides accurate clock synchronization for distributed control nodes for industrial automation applications.

6.1 FlexTimer Module

The FlexTimer Module (FTM) has the capability of capturing events and time-stamping them, and of generating events at certain times. It also has logic to generate PWM waveforms without the need for software intervention. A simplified block diagram of the FlexTimer module is shown below:

A simplified FTM block diagram

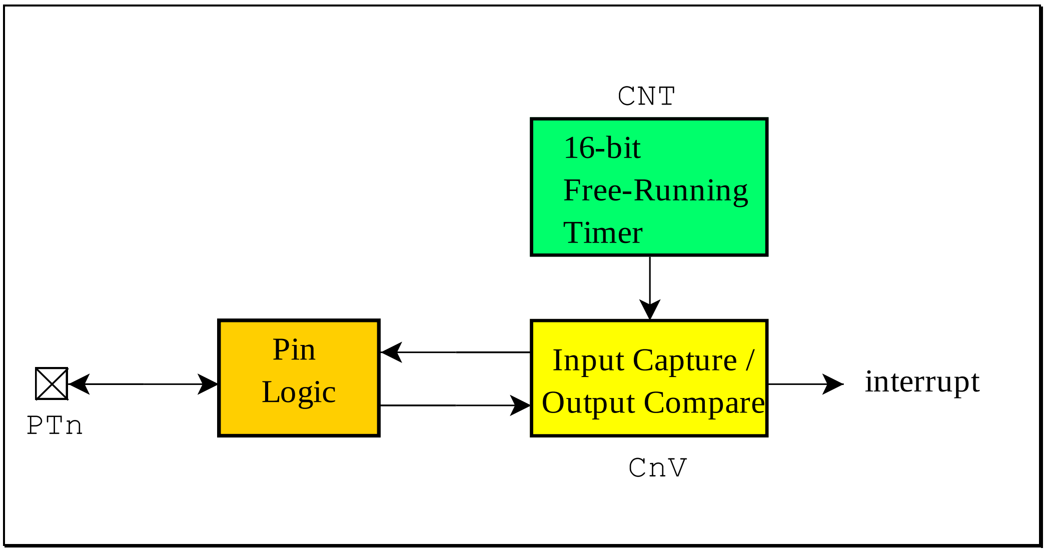


Figure 6.1

There is a 16-bit free-running timer called `CNT`. This is used to time-stamp an input event (an *input capture*) or to trigger an output event (an *output compare*).

The timer has a free-running counter and a value counter for each channel

The "input capture / output compare" block is just a register called `CnV`, where `n` is the channel number, that gets loaded with the current value of `CNT` for an input capture event, and which holds a desired value of `CNT` to trigger an output compare event.

The types of events to capture, or to initiate on a successful compare, are setup through various control registers. For inputs, it is possible to capture rising and falling edges. Outputs can be made to toggle, clear or be set.

There are numerous control registers used to set up the FTM module. Only a few are needed to interact with the FTM once it has been set up for a particular application. A complete description of the FTM can be found in Chapter 43 of NXP's *K64 Sub-Family Reference Manual*.

6.1.1 Output Compare

Output compare can be used to create square waves, generate pulses, implement time delays, and execute periodic interrupts. You can use output compare together with input capture to measure period and frequency over a wide range and with varying resolution.

A channel set up as an *output compare* channel will trigger an output action when the output compare register is equal to the free-running timer. A block diagram of the output compare action is shown below:

A simplified output compare block diagram

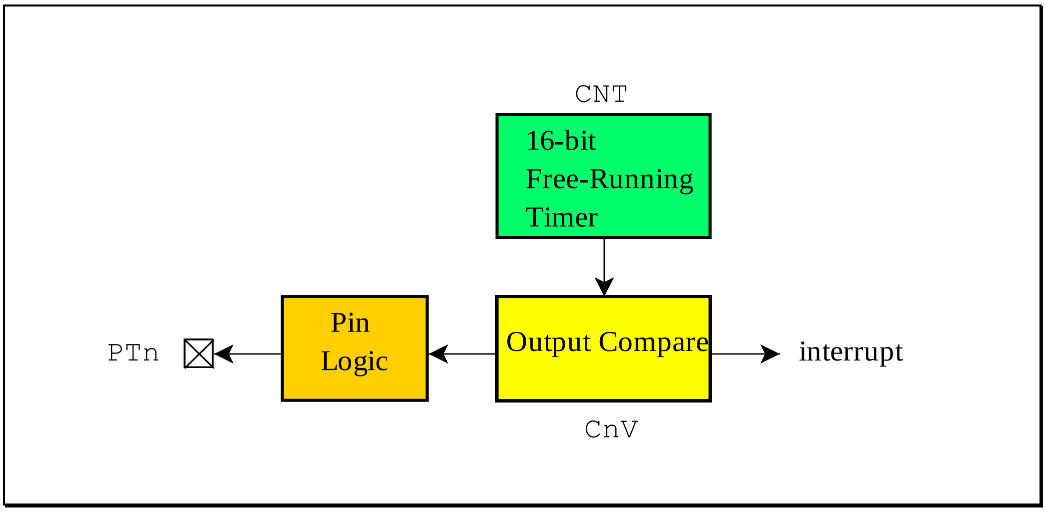


Figure 6.2

A compare result output action can be set up using the Channel Status and Control Register, `CnSC`, for the relevant channel. The options are:

Output compare actions

Action
Timer disconnected from output pin logic
Toggle output on match
Clear output on match
Set output on match

6.1.2 Time Delay Using Output Compare

One simple application of the output compare feature is to create a fixed timer. Timers are useful in situations where you start an operation, wait a certain amount of time, and then stop the operation. Usually the process looks like this:

1. Start an operation (turn on or turn off an output device).
2. Start the timer.
3. When the timer expires, stop the operation (turn off or turn on the output device).

You can also use timers to detect timeout conditions. For example, you turn on a motor and then start a timer. You expect the speed of the motor to increase, and if the speed doesn't exceed a threshold before a timer times out, then you might turn the motor off and notify an operator. In these cases, you start an operation then monitor the process to see if conditions are met before the timer expires:

1. Start an operation (turn on or turn off an output device).
2. Start the timer.
3. Monitor for desired conditions. If conditions are met, stop the timer.
4. If the timer times out, stop the operation and notify the operator.

Let `delay` be the number of cycles you wish to wait. The steps to start a timer are:

1. Read the current 16-bit `CNT`.
2. Set the 16-bit output compare register to `CNT + delay;`
3. Clear the output compare flag.
4. The output compare flag will set and trigger an interrupt after the required delay.

This method will only work for values of `delay` that fall between a minimum value (the time it takes to implement steps 1 to 3) and 65536. It will function properly even if `CNT` rolls over from `0xFFFF` to `0`, since the 16-bit addition is really a modulo `0x10000` addition.

EXAMPLE 6.1 Creating a Timer Using Output Compare

The output compare feature is a convenient mechanism to create a simple timer. We will turn an LED on when the timer on channel 3 of FTM0 times out, after 1 second.

```
// Timer value for 1 second
const uint16_t RATE = 24414;

void FTM0_Init(void)
{
  // Set up FTM0
  ...
  // Initialize NVIC
  ...
}

/*! @brief FTM0 interrupt handler.
 *
 */
void FTM0_IRQHandler(void)
{
  // Clear interrupt flag
  FTM0->CONTROLS[3].CnSC &= ~FTM_CnSC_CHF_MASK;

  // Call user function
  LED_On();
}

/*! @brief Timer output compare 3 initialization.
 *
 */
void TOC3_Init(void)
{
  // Ensure interrupts are disabled
  __disable_irq();

  // Initialise the FTM0 module
  FTM0_Init();

  // Enable Ch 3 as output compare with interrupts enabled
  FTM0->CONTROLS[3].CnSC = (FTM_CnSC_MSA_MASK | FTM_CnSC_CHIE_MASK);

  // Time out for 1 second
  FTM0->CONTROLS[3].CnV = FTM0->CNT + RATE;

  // Enable interrupts
  __enable_irq();
}
```

Note that the FTM0 ISR has taken on the pre-declared name of **`FTM0_IRQHandler`** which will be automatically placed in the vector table by the linker.

6.1.3 Input Capture

A channel can be set up as an *input capture* channel. We can use input capture to measure the period or pulse width of 3.3V CMOS signals. The input capture system can also be used to trigger interrupts on rising or falling transitions of external signals. A simplified block diagram of a channel set up for input capture is shown below:

A simplified input capture block diagram

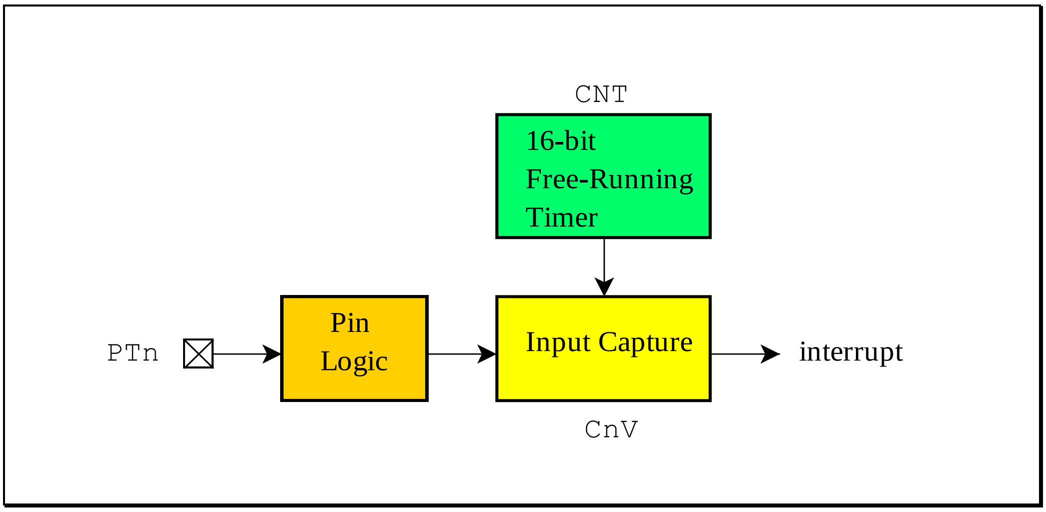


Figure 6.3

The input capture edge detection circuits can be set up using the `CnSC` register. The options are:

Input compare configurations

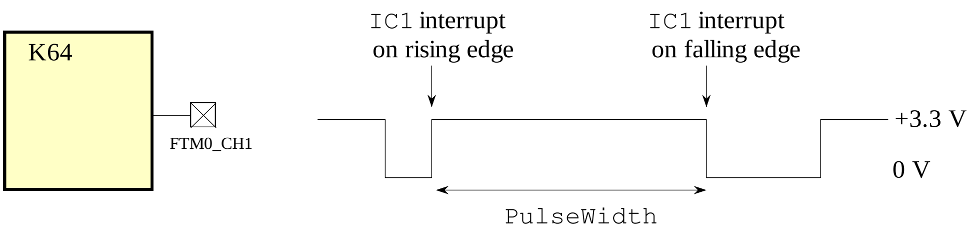
Configuration
Capture disabled
Capture on rising edges only
Capture on falling edges only
Capture on any edge (rising or falling)

Two or three actions result from a capture event:

1. The current 16-bit `CNT` value is copied into the input capture register, `CnV`.
2. The input capture flag is set in `CnSC`.
3. An interrupt is requested when the `CHIE` bit is `1` in `CnSC`.

EXAMPLE 6.2 Pulse-Width Measurement

The basic idea of pulse-width measurement is to cause an input capture event on first the rising edge and then the falling edge of an input signal. The difference between these two times will be the pulse width.



The resolution of the measurement is determined by the rate at which CNT is incremented. We will use Channel 1 of the timer for the implementation.

```
uint16_t PulseWidth;      // Pulse width in CNT units
bool Done;                // True when pulse width is measured

void FTM0_Init(void)
{
  // Set up FTM0
  ...
  // Initialize NVIC
  ...
}

void FTM0_IRQHandler(void)
{
  // Value of CNT at rising edge
  static uint16_t rising;

  // Clear interrupt flag
  FTM0->CONTROLS[1].CnSC &= ~FTM_CnSC_CHF_MASK;

  // See if a rising edge is detected
  if (FTM0_CH1)
  {
    // Record time of rising edge
    rising = FTM0->CONTROLS[1].CnV;
    // Set edge detection to falling edge only
    FTM0->CONTROLS[1].CnSC = (FTM_CnSC_ELSB_MASK | FTM_CnSC_CHIE_MASK);
  }
  else
  {
    // Falling edge detected – calculate the pulse width
    PulseWidth = FTM0->CONTROLS[1].CnV - rising;
    Done = true;
  }
}
```

```c
/*! @brief Timer input capture 1 initialization.
 *
 */
void TIC1_Init(void)
{
  // Ensure interrupts are disabled
  __disable_irq();

  // Initialize the FTM0 module
  FTM0_Init();

  // Enable Ch1 as input capture on rising edge
  FTM0->CONTROLS[1].CnSC = (FTM_CnSC_ELSA_MASK | FTM_CnSC_CHIE_MASK);

  // No measurement yet
  Done = false;

  // Enable interrupts
  __enable_irq();
}
```

Note that the FTM0 ISR has taken on the pre-declared name of **`FTM0_IRQHandler`** which will be automatically placed in the vector table by the linker.

6.2 Periodic Interrupt Timer

A block diagram of one of the periodic interrupt timer (PIT) is shown below:

A simplified periodic interrupt timer block diagram

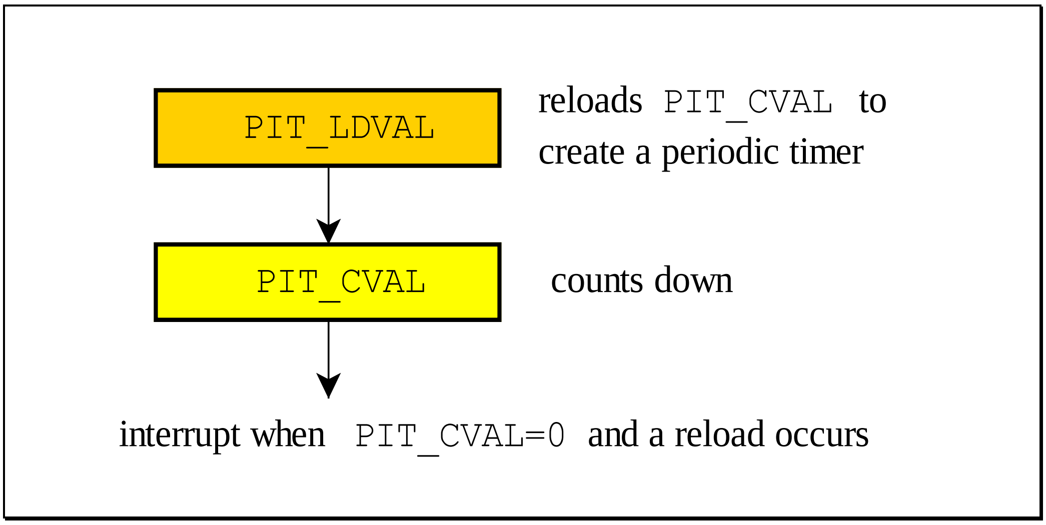


Figure 6.4

A PIT generates triggers at periodic intervals, when enabled. The timer loads the start value as specified in the `LDVAL` register, counts down to `0` and then loads the respective start value again. Each time the timer reaches `0`, it will generate a trigger pulse and set the interrupt flag.

Note that an interrupt will occur (if enabled) when the `CVAL` register reaches zero and the next clock "tick" reloads the start value as specified in the `LDVAL` register. For example, to create a timer with a period of 1000 "ticks" of the module clock, the `LDVAL` register needs to be loaded with 999.

A complete description of the PIT can be found in Chapter 41 of NXP's *K64 Sub-Family Reference Manual*.

6.3 Real Time Clock

The management of time is important in many embedded systems. The real time clock (RTC) unit in the K64 has the ability to operate in a separate power domain – it operates from *Vbat*, which can be connected to a battery. It therefore has the potential to keep track of the time while the main power is off.

On the FRDM-K64 board, the RTC is powered via the USB and not a battery. Therefore, it does not have the ability to keep the time when powered off.

The RTC unit relies on an external 32.768 kHz crystal for its timekeeping. The crystal must have "load" capacitors connected to it to function properly. The K64 has the ability to select internal load capacitors (and therefore minimise external hardware). The selection of the internal load capacitors is accomplished by bits in the RTC Control register, `RTC_CR`. However, for the FRDM-K64 board, the load capacitors are **external**:

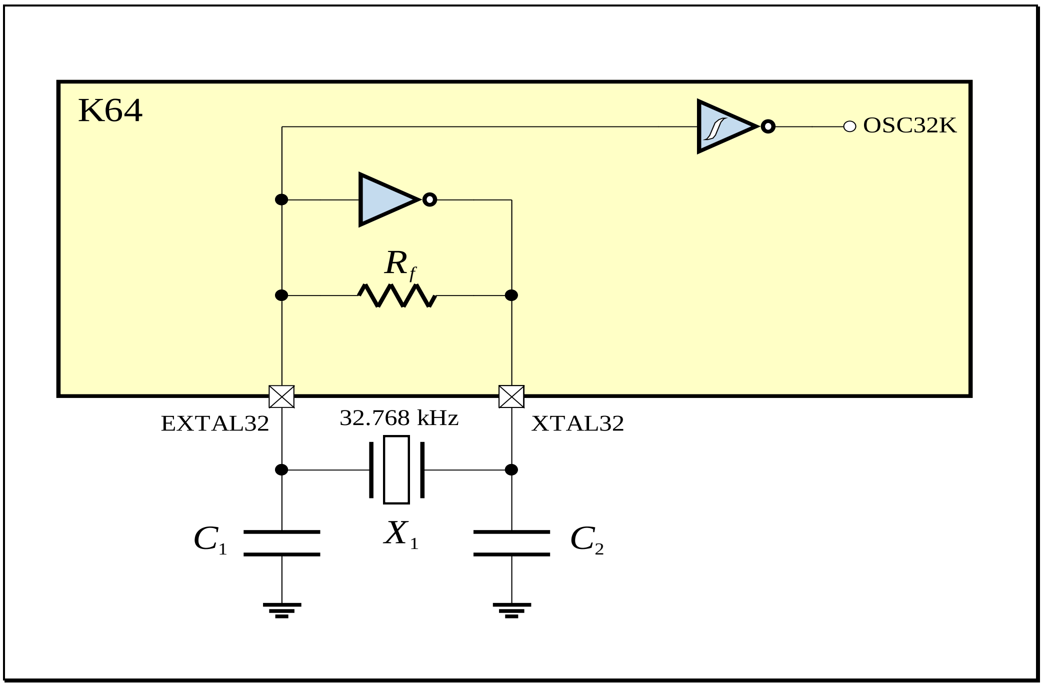


Figure 6.5

The RTC has the ability to generate an interrupt every second.

A complete description of the RTC can be found in Chapter 44 of NXP's *K64 Sub-Family Reference Manual*.

7 Concurrent Software

Contents

7.2

Introduction

A thread is an executing program, with a context

A program is a list of instructions for the computer to execute. A *thread* is an executing program, including the current values of the program counter, registers and variables. A thread has an execution state (such as running, ready, waiting) and a saved thread context when not running. Conceptually, each thread has its own CPU. In reality, of course, the real CPU switches back and forth from thread to thread.

A process is a multiprogramming concept

By contrast, a *process* is a conceptual entity used when dealing with multiple programs running on a general purpose computer (such as a PC with Windows®). A process has its own virtual address space and has protected access to the CPU, other processes, files, and I/O resources.

Simple embedded systems only have threads

In an embedded system, there is no need for multiple programs to be executing, since the embedded system is usually designed for a specific application. Thus, in an embedded system there is no need for the concept of a process and we will deal exclusively with threads.

Simple embedded systems are foreground / background systems

The execution of the main program is called the *background* thread. In most embedded applications, the background thread executes a loop that never ends. This thread can be broken (execution suspended, then restarted) by *foreground* threads (interrupt service routines). These threads are run using a simple algorithm. The ISR of an input device is invoked when new input is available. The ISR of an output device is invoked when the output device is idle and needs more data. Last, the ISR of a periodic task is run at a regular rate. The main program runs in the remaining intervals. Many embedded applications are small in size, and static in nature, so this configuration is usually adequate.

The limitation of a single background thread comes as the size and complexity of the system grows. Projects where the software modules are loosely coupled (independent) more naturally fit a multiple background thread configuration.

A *scheduler* is a piece of software that implements multiple background threads, and forms the basis of a program known as an *operating system* (OS). Synchronization tools that allow threads to interact with each other (such as *semaphores*) are also a key feature of operating systems.

A scheduler lets us implement multiple background threads

Systems that implement a thread scheduler still may employ regular I/O driven interrupts. In this way, the system supports multiple foreground threads and multiple background threads.

7.1 Threads

A *thread* is the execution of a software task that has its own stack and registers. Since each thread has a separate stack, its local variables are private, which means it alone has access.

Each thread has its own registers and stack

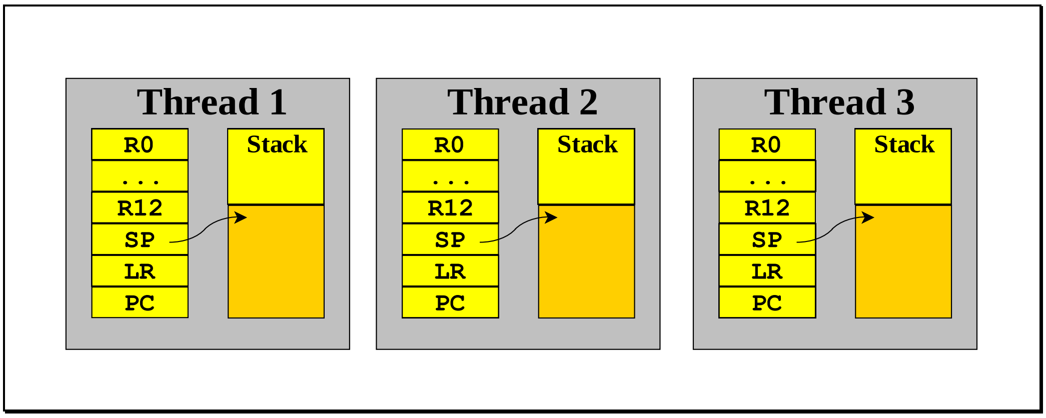


Figure 7.1

Multiple threads cooperate to perform an overall function. Since threads interact for a common goal, they do share resources, such as global memory, and I/O devices.

Threads share global memory and I/O ports

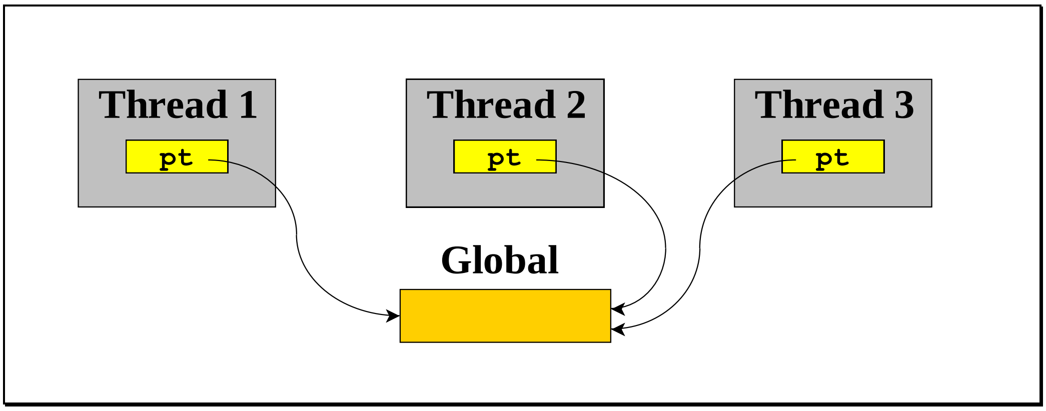


Figure 7.2

In summary, a thread:

- is the execution of a software task
- has its own stack and registers
- has local variables which are private
- cooperates to perform an overall function

A thread can be in one of three states.

Thread states

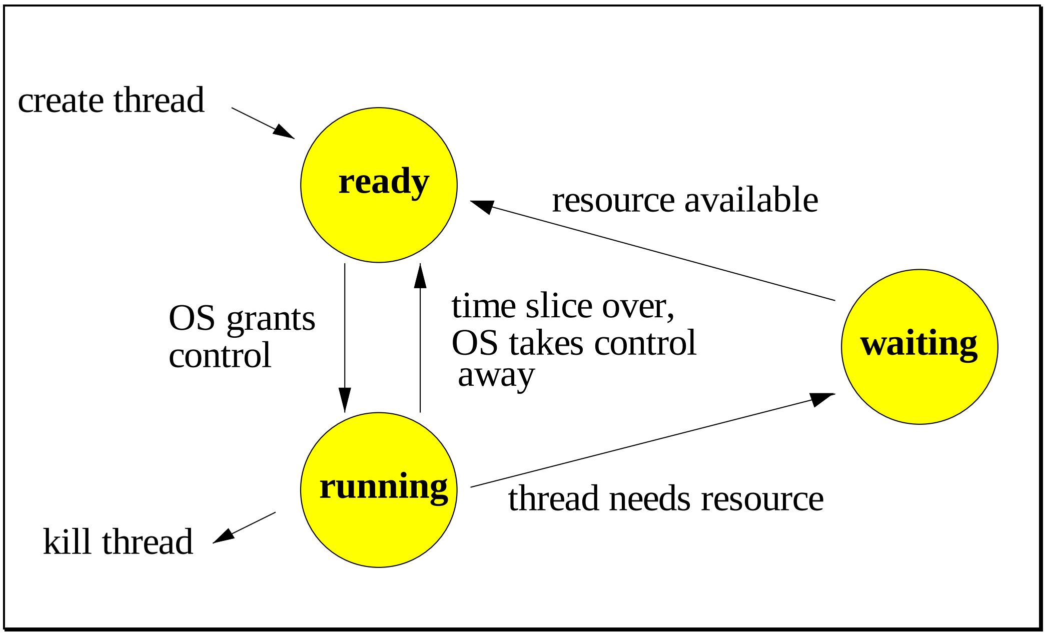


Figure 7.3

A thread is in the *ready* state if it is ready to run but waiting for its turn.

A thread is in the *running* state if it is currently executing. With a single instruction stream (i.e. one core) processor like the K64, at most one thread can be in the run state at a time.

A thread is in the *waiting* state when it is waiting for some external event like I/O (keyboard input available, printer ready, I/O device available). If a thread communicates with other threads, then it can be waiting for an input message or waiting for another thread to be ready to accept its output message. If a thread wishes to output to the serial port, but another thread is currently outputting, it will wait. If a thread needs information from a FIFO (calls `FIFO_Get`), then it will wait if the FIFO is empty (because it cannot retrieve any information). On the other hand, if a thread outputs information to a FIFO (calls `FIFO_Put`), then it will wait if the FIFO is full (because it cannot save its information).

An OS may use a linked list data structure to hold the ready and waiting threads. It may create a separate waiting linked list for each reason why the thread cannot execute. For example, one waiting list for "full" during a call to **`FIFO_Put`**, and one for "empty" during a call to **`FIFO_Get`**. In general, the OS could have one waiting list associated with each cause of waiting.

In the figure below, thread 5 is running, threads 1 and 2 are ready to run, and threads 3 and 4 are waiting because a FIFO is empty.

Threads are placed in linked lists depending on whether they are ready or waiting

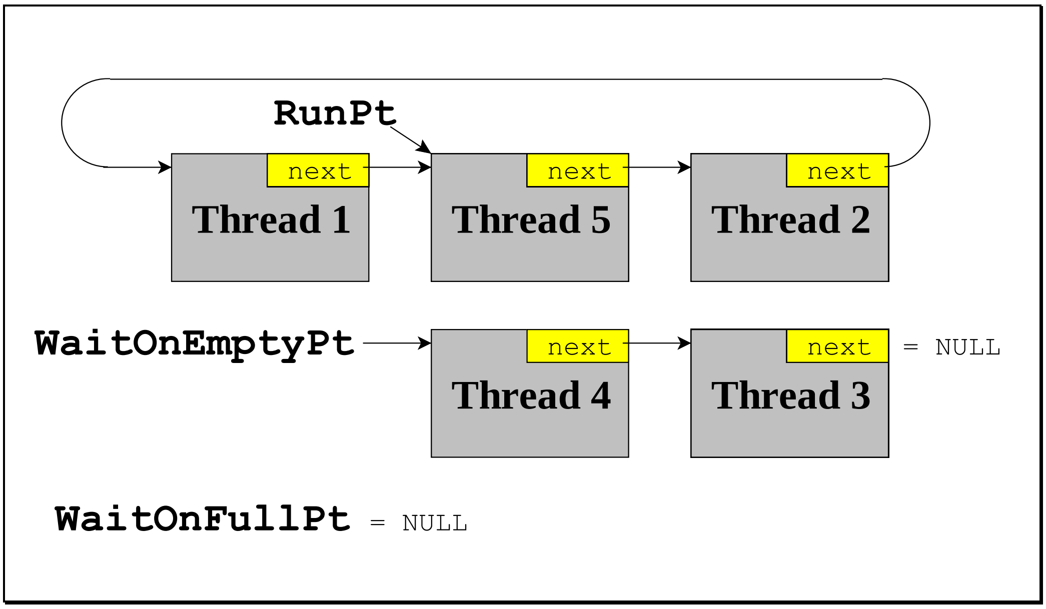


Figure 7.4

7.1.1 Thread Control Blocks (TCBs)

If a thread is ready, it may be granted control of the CPU by the OS at any time. Conversely, while running, the OS may stop the thread executing and make it ready. We therefore need a way for the scheduler to save and restore the state of a thread. A *thread control block* (TCB) is used to store the information about each thread.

The TCB must contain:

Essential thread control block components

1) a pointer to its private stack;
2) a pointer so it can be chained into a linked list;
3) a stack area that includes local variables

While a thread is running, it uses the actual hardware registers, `R0–R15`. In addition to these necessary components, the TCB might also contain:

Optional thread control block components

4) Thread number, type, or name;
5) Age, or how long this thread has been active;
6) Priority;
7) Resources that this thread has been granted.

TCB structure

The structure of a typical TCB is shown below:

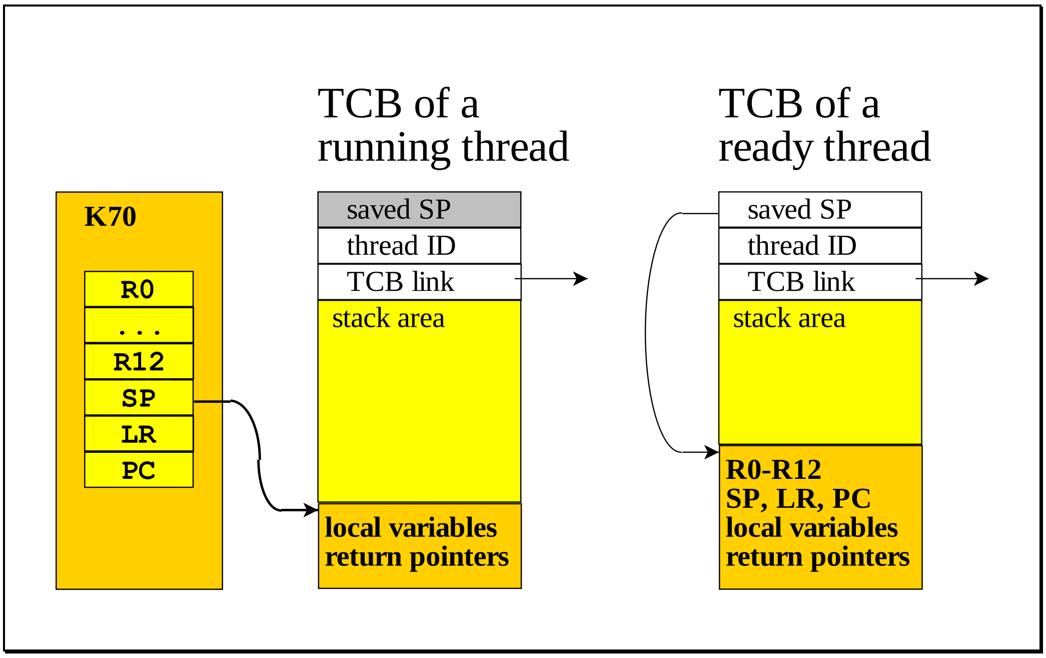


Figure 7.5

The running thread uses the actual registers, while the other threads have their register values saved on the stack.

7.2 Schedulers

A scheduler is responsible for changing the running thread

A *scheduler* is an OS component that has responsibility for switching threads between states. A scheduler has to implement two aspects of this operation. One aspect is to save the currently running thread's state in its TCB and to restore the state of the next thread to run (the process of changing threads, which is also called a context switch). The other aspect is *when* the scheduler actually changes threads, and what it does with waiting threads.

In a preemptive scheduler, the OS interrupts each thread regardless of whether the thread is "in the middle of something important" – the OS is the sole arbiter of when the thread will actually get CPU time.

A round-robin scheduler runs each thread in a fixed order for a certain amount of time

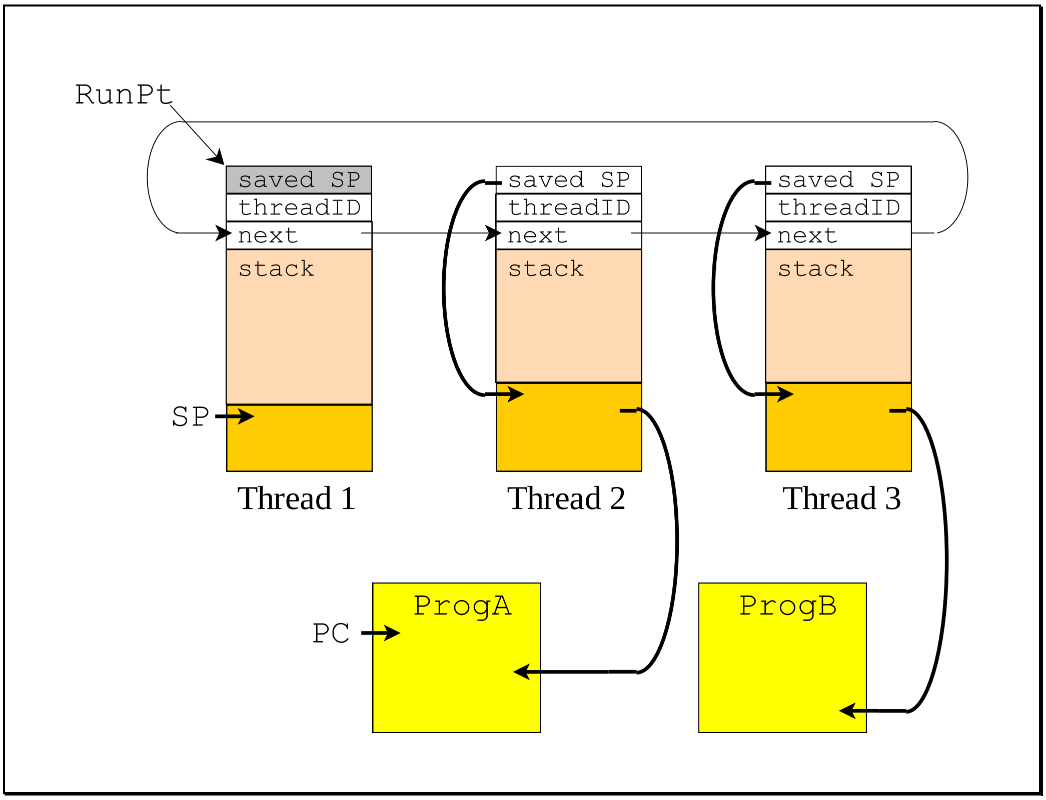
The simplest scheduling system is a round-robin scheduler – a scheduler that runs each "ready" thread for a certain amount of time in a fixed cyclic order. It does this by "hooking" into a periodic timer whose ISR performs the thread changeover function.

EXAMPLE 7.1 Round-Robin Shceduler

Suppose we have three dynamically allocated threads that are executed in a round-robin fashion. Even though there are three threads, there are only two programs to run, **`ProgA`** and **`ProgB`**. Recall that a thread is not simply the software but the execution of the software. We will have two threads executing the same program, **`ProgA`**.

```
void main(void)
{
  OS_AddThread(&ProgA);
  OS_AddThread(&ProgA);
  OS_AddThread(&ProgB);
  OS_Start(TIMESLICE); // doesn't return
}
```

A circular linked list allows the scheduler to run all three threads equally.



Round-robin scheduling

Figure 8.1

This example illustrates the difference between a program (e.g. **`ProgA`** and **`ProgB`**) and a thread (e.g. Thread 1, Thread 2 and Thread 3). Notice that Threads 1 and 2 both execute **`ProgA`**. There are many applications where the same program is being executed multiple times.

7.2.1 Other Scheduling Algorithms

Non-preemptive scheduling

A non-preemptive (cooperative) scheduler trusts each thread to voluntarily release control on a periodic basis. Although easy to implement, because it doesn't require interrupts, it is not appropriate for real-time systems.

Priority scheduling

A priority scheduler assigns each thread a priority number (e.g. 0 is the highest, 15 is the lowest). Two or more threads can have the same priority. A priority 1 thread is run only if no priority 0 threads are ready to run. Similarly, we run a priority 2 thread only if no priority 0 or priority 1 threads are ready. If all threads have the same priority, then the scheduler reverts to a round-robin system. The advantage of priority is that we can reduce the latency (response time) for important tasks by giving those tasks a high priority. The disadvantage is that on a busy system, low-priority threads may never be run. This situation is called *starvation*.

7.3 The SysTick Timer

The Cortex®-M processors have a small integrated timer called the SysTick (System Tick) timer. It is integrated as a part of the NVIC and can generate the SysTick exception (exception type #15). The SysTick timer is a simple decrement 24-bit timer, and runs off the processor's core clock.

An overview of the SysTick timer

In modern operating systems, a periodic interrupt is needed to ensure that the OS *kernel* can be invoked regularly for task management and context switching. This enables a processor to handle different tasks in different time slots. The SysTick timer is similar to the PIT module:

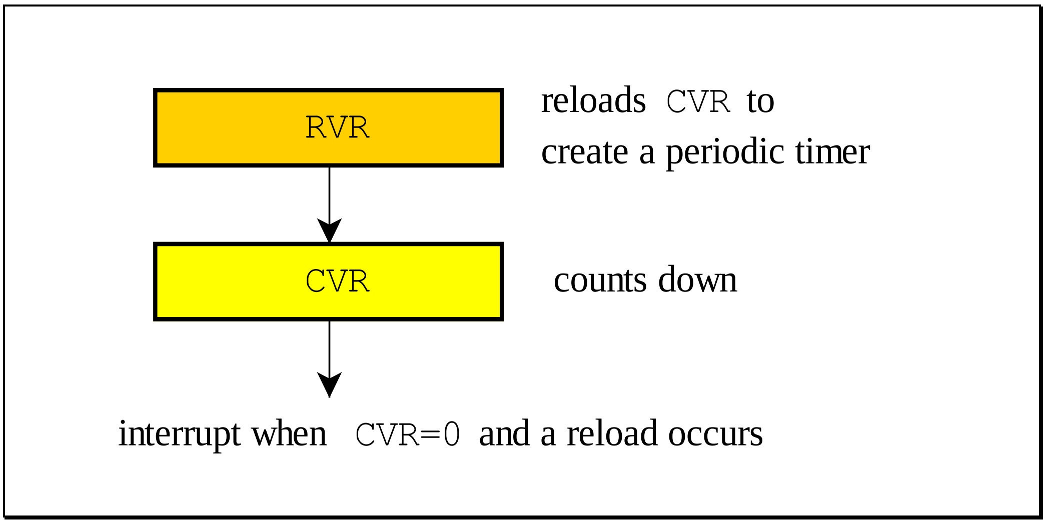


Figure 7.6

There is a current value `CVR` and a reload value `RVR`. When the SysTick counter is enabled, the `CVR` decrements every clock cycle. If it reaches zero, it will then load the value from `RVR` and continue. If the SysTick interrupt is enabled, it will generate an interrupt when it reloads the value.

There is a Control and Status Register (`CSR`) that allows us to control and check the status of the SysTick timer. To generate a periodic interrupt using SysTick, we need to:

- set the `RVR` reload value for the desired period
- set `CSR` bit 0 to a 1 to enable the SysTick timer
- set `CSR` bit 1 to a 1 to enable interrupt generation

7.4 A Simple Operating System

The following example shows how a simple operating system can be created using the SysTick timer, a simple TCB structure, and some low-level assembly language context switching.

EXAMPLE 7.2 Simple Operating System

Suppose we have three statically allocated threads that are each allowed to execute for 125 ms in a round-robin fashion. Even though there are three threads, there are only two programs to run, `ProgA` and `ProgB`. We will have two threads executing the same program, `ProgA`, and one thread executing `ProgB`. The code for these programs is shown below.

```
int Inc(const int j)
{
  int i;
  i = j + 1;
  return i;
}

void ProgA(void)
{
  int i;
  i = 5;
  while (1)
  {
    i = Inc(i);
  }
}

void ProgB(void)
{
  int i;
  i = 6;
  while (1)
  {
    i = Inc(i);
  }
}
```

Notice that both threads call the same subroutine `Inc()`.

The thread control block is defined in `OS.h` as follows:

```
// Thread control block structure
struct TCB
{
  unsigned long *savedSP;        // saved stack pointer
  unsigned long threadId;        // 1, 2, 3, ...
  struct TCB *nextTCB;           // link to next TCB
  unsigned long moreStack[99];   // the free stack
  unsigned long stackedR7;       // the ARM "stack frame" when
  unsigned long stackedR0;       // an exception occurs...
  unsigned long stackedR1;
  unsigned long stackedR2;
  unsigned long stackedR3;
  unsigned long stackedR12;
  unsigned long stackedLR;
  void (*stackedPC)(void);
  unsigned long stackedxPSR;
};

typedef struct TCB TCBType;
typedef TCBType* TCBPtr;
```

The `main()` module statically declares the threads as global variables:

```
TCBType Threads[3] __attribute__ ((aligned(0x08))) =
{
  {
    &Threads[0].stackedR7,     // initial SP
    1,                         // thread ID
    &Threads[1],               // pointer to next TCB
    { 0 },                     // empty stack
    7,                         // R7
    0, 1, 2, 3, 12, 0,         // R0-R3, R12, LR
    ProgA,                     // PC
    0x01000000                 // xPSR
  },
  {
    &Threads[1].stackedR7,     // initial SP
    2,                         // thread ID
    &Threads[2],               // pointer to next TCB
    { 0 },                     // empty stack
    7,                         // R7
    0, 1, 2, 3, 12, 0,         // R0-R3, R12, LR
    ProgA,                     // PC
    0x01000000                 // xPSR
  },
  {
    &Threads[2].stackedR7,     // initial SP
    3,                         // thread ID
    &Threads[0],               // pointer to next TCB
    { 0 },                     // empty stack
    7,                         // R7
    0, 1, 2, 3, 12, 0,         // R0-R3, R12, LR
    ProgB,                     // PC
    0x01000000                 // xPSR
  }
};
```

Even though the threads have not yet been allowed to run, they are created with an initial stack area that "looks like" the thread has been suspended by the K64 exception mechanism (i.e. the stack looks the same as if an interrupt has occurred). When a thread is launched for the first time, it will execute the program specified by the value in the `.stackedPC` location.

The `main()` function initialises the low-level hardware and calls **`OS_Init()`** and **`OS_Start()`** to start multithreading.

```
int main(void)
{
  Board_Init();
  // Initialise OS - sets up SysTick
  OS_Init();
  // Call OS to start multitasking - never returns
  OS_Start(Threads);
}
```

The OS module provides three functions. **`SysTick_Handler()`** handles the context switching for the threads:

```
// Pointer to current thread
static TCBPtr RunPtr;

static uint32_t Count;

void SysTick_Handler(void)
{
  // Save current context
  __asm (\
    "str r7, %[input]\n\t"\
    ".align 4\n\t"\
    ::[input] "m" (*RunPtr)  \
    : );

  // Increment the number of elapsed ticks
  Count++;

  // Get the next thread pointer - round robin scheduler
  RunPtr = RunPtr->nextTCB;

  // Load next context
  __asm (\
    "ldr r0, %[input]\n\t"\
    "msr msp, r0\n\t"\
    "pop {r7}\n\t"\
    "bx lr\n\t"\
    ".align 4\n\t"\
    ::[input] "m" (*RunPtr)  \
    : "r0");
}
```

Note that the SysTick ISR has taken on the pre-declared name of **`SysTick_Handler`** which will be automatically placed in the vector table by the linker.

The assembly language with the comment "Save current context" does exactly that – it saves the current value of the stack pointer (held in `R7`) into the thread control block structure (using `*RunPtr`).

It then increments its own internal counter `Count`, and advances the `RunPtr` through the linked list to get the next thread control block – this is round-robin scheduling.

The assembly language with the comment “Load next context” does exactly that – it loads the stack pointer with the value that was previously saved in the new thread’s control block, pops the contents of `R7` off the stack, and then carries out an “exception return” using the EXC_RETURN value held in the LR register.

Program execution will now continue from wherever the thread was interrupted.

The `OS_Init()` function initialises the SysTick timer for a 125 ms interval:

```
// Set the tick frequency to 8 Hz (1/8 s period)
static const uint32_t TICK_FREQUENCY = 8U;

void OS_Init(void)
{
  // Disable interrupts
  __disable_irq();

  // Set reload value for the timer period
  SysTick_Config(SystemCoreClock / TICK_FREQUENCY);
}
```

`OS_Start()` initialises the `RunPtr` and launches the first thread.

```
void OS_Start(TCBType threads[])
{
  uint32_t initialSP, initialPC;

  // Specify Thread 1 as the first thread to launch
  RunPtr = &threads[0];

  // Get the address of the top of the Thread 1 stack
  initialSP = (uint32_t)RunPtr + sizeof(TCBType);

  // Set MSP to top of Thread 1 stack
  __asm (                   \
    "ldr r0, %[input]\n\t"\
    ::[input] "m" (initialSP)  \
    : "r0");
  __asm ("msr msp, r0\n\t");
  __asm (".align 4\n\t");

  // Enable interrupts
  __enable_irq();

  // Launch Thread 1 by loading the PC directly
  initialPC = (uint32_t)RunPtr->stackedPC;
  __asm (                   \
    "ldr pc, %[input]\n\t"\
    ::[input] "m" (initialPC)  \
    : "r0");
```

7.5 The Semaphore

An operating system, at the very least, provides scheduling and synchronization tools for threads. One of the most important synchronization tools provided by an OS is the semaphore.

A *semaphore* is a non-negative integer, which may *only* be operated on by the primitive operations **`wait`** and **`signal`**. These **`wait`** and **`signal`** operations are *indivisible*. Indivisibility implies that only one thread can access each of these primitives at any one time. That is, only one primitive can modify the value of some semaphore, say `s`. The primitives can't be interrupted or logically cut into any smaller pieces.

Semaphores defined

The primitive operations are defined as follows:

`wait``(s)`	Decrease (indivisibly) the value of `s` by `1`
`signal``(s)`	Increase (indivisibly) the value of `s` by `1`

Note: A semaphore `s` may only be a non-negative integer, so that if the **`wait`**`(s)` *cannot be completed*, then a thread will be put into the waiting state. Conversely a **`signal`**`(s)` may cause a waiting thread to be made active.

Every **`signal`**`(s)` increments `s` but a **`wait`**`(s)` is only completed if `s > 0`.

7.5.1 Mutual Exclusion with Semaphores

Listing 7.1 shows how semaphores can be used to guarantee that a thread will have *uninterrupted* access to its critical code section. That is, there is mutual exclusion of other threads.

Mutual exclusion between threads

```
void p1(void)
{
  while (1)
  {
    ...
    OS_Wait(&Mutex);
    // critical code of p1
    ...
    OS_Signal(&Mutex);
    // remainder of p1
    ...
  }
}

void p2(void)
{
  while (1)
  {
    ...
    OS_Wait(&Mutex);
    // critical code of p2
    ...
    OS_Signal(&Mutex);
    // remainder of p2
    ...
  }
}

int Mutex; // a binary semaphore

void main(void)
{
  // Mutually excluded threads
  Mutex  = 1;
  OS_AddThread(&p1);
  OS_AddThread(&p2);

  OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 7.1 – Mutual exclusion using Semaphores

There are two separate functions **`p1`** and **`p2`**, each with a critical section of code. Each of these critical sections are protected between **`OS_Wait`** and **`OS_Signal`** operations. A semaphore `Mutex` is initialised to `1` at the beginning of the main program. This value guarantees only one thread can be in its critical section at one time.

Since the speed of each thread is indeterminate we have no way of knowing which one will try to execute an `OS_Wait(&Mutex)` first. Let us assume that `p1` tries first. Since the semaphore has a value of `1` at this point, the `OS_Wait` will complete (that is, `OS_Wait` will decrement `Mutex` to `0`) and `p1` will enter it's critical section.

If while `p1` is in the critical section `p2` attempts a `OS_Wait(&Mutex)` then the `OS_Wait` will not complete and `p2` will be waiting (internally to the operating system, `p2` will be added to a queue of threads that are all waiting on the semaphore `Mutex`).

At some later time `p1` will complete its `OS_Signal(&Mutex)` operation, and `p2` may now complete its `OS_Wait(&Mutex)` and enter it's critical section. After each thread completes its `OS_Signal` operation, the value of the semaphore again becomes `1` allowing *either* thread to again gain mutually exclusive access to its critical section.

7.5.2 Simple Mutual Exclusion

Semaphores should be used with care in embedded systems. The use of a semaphore to access a simple shared variable is overkill in most situations. The overhead involved in acquiring and releasing the semaphore consumes valuable CPU time. You can perform the job more efficiently by disabling and enabling interrupts. However there is an indirect cost to disabling interrupts: even higher priority tasks that do not share the specific resource are blocked from using the CPU. Disabling interrupts also increases interrupt latency, which may not be acceptable.

7.5.3 Priority Inversion in Real-Time Systems

Semaphores are subject to a serious problem in real-time systems called *priority inversion*, which only occurs when using an OS based on a priority-based pre-emptive kernel. To solve this, most RTOSs provide for a special type of semaphore called a *mutual-exclusion semaphore*.

7.5.4 Synchronisation using Semaphores

Listing 7.2 shows how two threads **p1** and **p2** can synchronise their operations with each other.

Synchronization between threads

```
void p1(void)
{
  while (1)
  {
    // some amount of code
    ...
    OS_Wait(&Proceed);
    // remainder of p1
    ...
  }
}

void p2(void)
{
  while (1)
  {
    // some other amount of code
    ...
    OS_Signal(&Proceed);
    // remainder of p2
    ...
  }
}

int Proceed; // a binary semaphore

void main(void)
{

  // Synchronized threads
  Proceed = 0;
  OS_AddThread(&p1);
  OS_AddThread(&p2);

  OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 7.2 – Synchronization using Semaphores

In this example **p1** will pause until **p2** executes a **OS_Signal**(&Proceed) before it continues. Of course if **p2** executes the **OS_Signal** before **p1** can execute the **OS_Wait** operation, then **p1** will not be held up.

This is known as *asymmetric* synchronisation – can you re-design this example so that symmetrical synchronisation between the two threads results?

7.5.5 The Producer / Consumer Problem using Semaphores

A classical problem in concurrent programming is the producer / consumer problem. Here two threads communicate through a buffer. The buffer has a finite amount of space and the producer, at its own speed, produces items and deposits them in this buffer of size `SpaceAvailable`.

The consumer, at its own speed, removes items from the buffer. Of course the consumer cannot extract items from an empty buffer nor can the producer deposit items into a full buffer.

There are to be three semaphores:

`SpaceAvailable`	This has an initial value of the size of the empty buffer.
`ItemsAvailable`	This has a value equal to the items in the buffer at initialisation (that is, `0`).
`BufferAccess`	This controls access to the buffer so that only one thread, producer or consumer, can gain access at one time.

Synchronization is achieved through the semaphores `SpaceAvailable` and `ItemsAvailable` whose initial values are the size of the buffer `40` and `0` respectively. Mutual exclusion of threads accessing the buffer simultaneously is effected by the semaphore `BufferAccess` with initial value `1`.

Producer / consumer problem using semaphores

```
void Producer(void)
{
  while (1)
  {
    // produce item
    ...
    OS_Wait(&SpaceAvailable);
    OS_Wait(&BufferAccess);
    // deposit item in buffer
    ...
    OS_Signal(&BufferAccess);
    OS_Signal(&ItemsAvailable);
  }
}

void Consumer(void)
{
  while (1)
  {
    OS_Wait(&ItemsAvailable);
    OS_Wait(&BufferAccess);
    // extract item from buffer
    ...
    OS_Signal(&BufferAccess);
    OS_Signal(&SpaceAvailable);
    // consume item
    ...
  }
}

int SpaceAvailable = BUFFER_SIZE; // size of the buffer
int ItemsAvailable = 0;
int BufferAccess = 1;

void main(void)
{
  // producer and consumer threads
  OS_AddThread(&Producer);
  OS_AddThread(&Consumer);

  OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 7.3 – Producer / Consumer problem using Semaphores

8 Interfacing

Contents

8.2

Introduction

An embedded system is normally designed to interact with the external world. They sometimes need to provide a human-machine interface for simple input / output operations. They also may need to measure analog quantities and output analog quantities. The following sections look at various techniques of interfacing to our microcontroller.

8.1 Input Switches

8.1.1 Interfacing a Switch to the Microcontroller

Several simple switch interfaces are shown below:

Simple switch interfaces

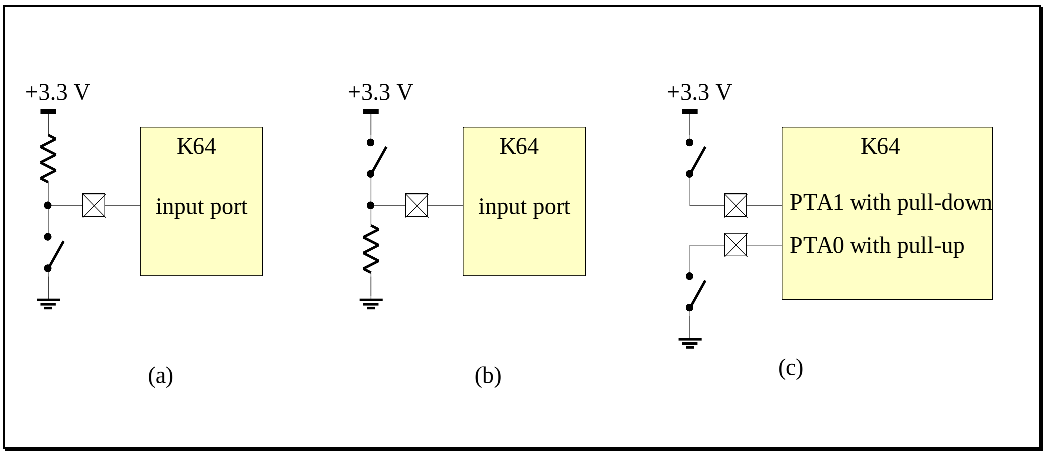


Figure 8.1

CMOS digital logic can use either pull-up or pull-down resistors, and the supply is typically 1.8 V, 2.5 V or 3.3V. In Figure 8.1 (a), a pull-up resistor is used to convert the mechanical signal into an electrical signal. When the switch is open, the input port is pulled to +3.3 V. When the switch is closed, the input port is forced to 0V.

Figure 8.1 (b) shows a pull-down circuit. When the switch in this circuit is open, the input is pulled to 0 V. When the switch is closed, the output is forced to +3.3 V. Notice the logic level of the switch input is reversed in the pull-down interface as compared to the pull-up case.

All ports on the K64 support both internal pull-ups and pull-downs. That is, either of the first two circuits in Figure 8.1 could be implemented on the K64 without the resistor, as shown in Figure 8.1 (c).

The software initialization for using a port sets the pull enable (`PE`) bit in the `PORTx_PCRn` register to enable pull-up or pull-down. For each port pin that is enabled for pull-up or pull-down, the corresponding pull select (`PS`) bit in the `PORTx_PCRn` register determines if it is pull-up (`1`) or pull-down (`0`).

EXAMPLE 8.1 Switch Interfacing with Internal Pull

Suppose we wish to initialize Port A for the circuit shown in Figure 8.1 (c). The software below will initialize Port A with the appropriate pull-up and pull-down.

```
// Port A Bit 0 is connected through a switch to 0 V
//   and uses internal pull-up
// Port A Bit 1 is connected through a switch to +3.3 V
//   and uses internal pull-down

void PortA_Init(void)
{
  // Enable clock gate for Port A to enable pin routing
  SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

  // Set Port A Bit 0 as an input
  GPIOA->PDDR &= ~0x00000001;
  // PORTA_PCR0: ISF=0, MUX=1, PE = 1, PS = 1
  PORTA->PCR0 = PORT_PCR_ISF_MASK | PORT_PCR_MUX(1)
              | PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;

  // Set Port A Bit 1 as an input
  GPIOA->PDDR &= ~0x00000002;
  // PORTA_PCR1: ISF=0, MUX=1, PE = 1, PS = 0
  PORTA->PCR1 = PORT_PCR_ISF_MASK | PORT_PCR_MUX(1)
              | PORT_PCR_PE_MASK;
}
```

8.1.2 Hardware Debouncing Using a Capacitor

Switch bounce causes multiple input changes on an input pin

Most inexpensive switches mechanically “bounce” when touched and when released. Typical bounce times range from 1 ms to 25 ms. Ideally, the switch resistance is zero (actually about 0.1Ω) when closed and infinite when open. This gives rise to the following switch timing:

Switch timing showing bounce on touch and release

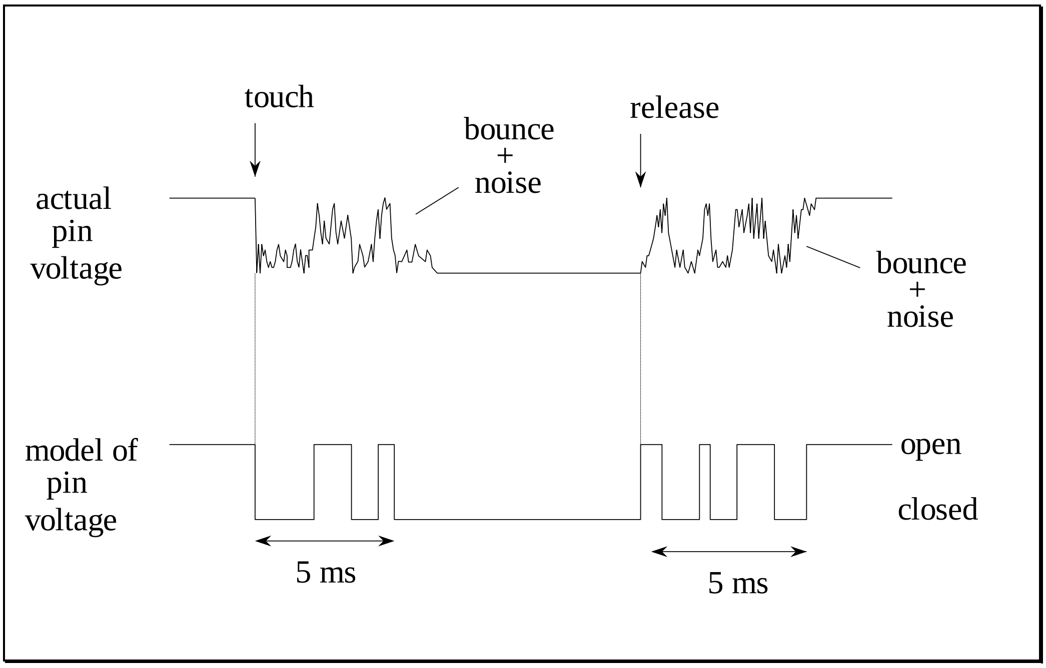


Figure 8.2

Hence, the electrical output “bounces” when using inexpensive switches and circuits having just a pull-up or pull-down resistor. It may or may not be important to debounce the switch. For example, if we are entering data via a keyboard, then we want to record only individual key presses. On the other hand, if the switch position specifies some static condition, and the operator sets the switch before turning on the microcontroller, then debouncing is not necessary.

A hardware method to debounce a switch places a capacitor across the switch to limit the rise time, followed by an inverter with hysteresis. With this circuit there is a significant delay from the release of the switch until the fall of the output.

A hardware circuit that removes switch bounce

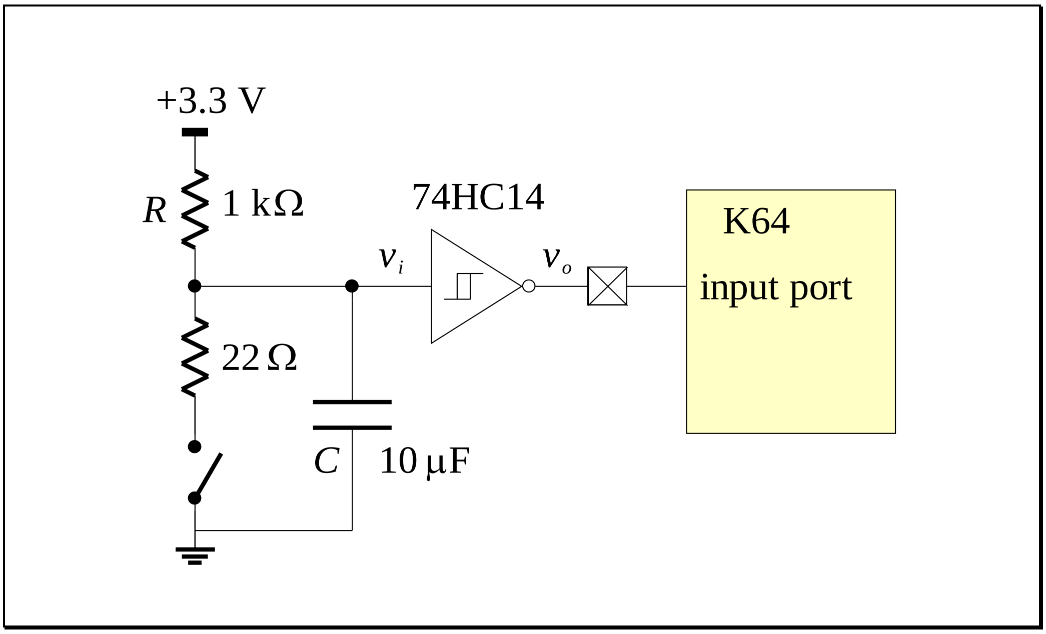


Figure 8.3

If the input switch is closed, its resistance will be about $0.1\,\Omega$, and the output of the 74HC14 will be high (logic 1). If the input switch is open, its resistance will be infinite, and the output of the 74HC14 will be low (logic 0). The $22\,\Omega$ is used to limit the discharge current when the switch is pressed (which causes sparks that produce carbon deposits to build up until the switch no longer works).

8.6

Switch touch bounce is removed by the capacitor

The touch timing with and without the capacitor is shown below:

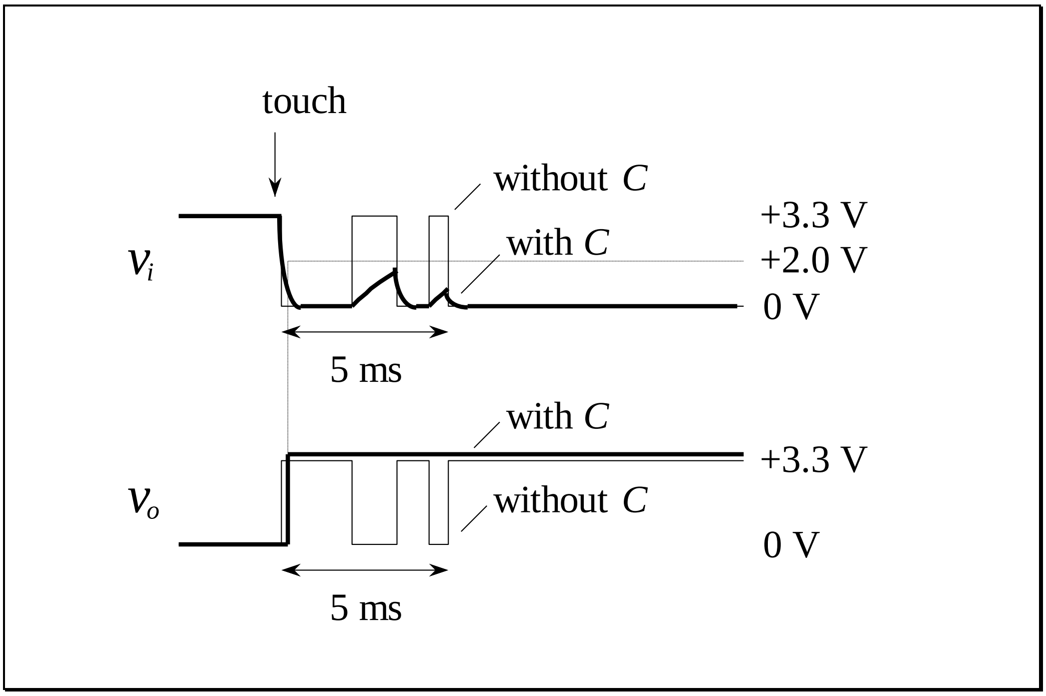


Figure 8.4

Notice that there is minimal delay between the touching of the switch and the transition of the Schmitt inverter output. This is because the capacitor is quickly discharged through the $22\,\Omega$ resistor.

With a capacitor-based debounced switch, there is minimal delay between the closing of the switch and the rising edge at the microcontroller input.	(9.1)

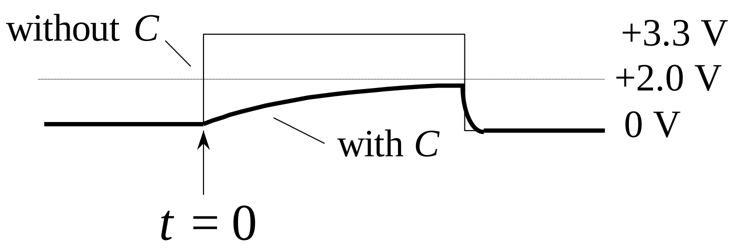
The voltage rise during a bounce interval when the switch is open is given by:

$$v(t) = V_{OH}\left(1 - e^{-t/RC}\right) \tag{9.2}$$

The capacitor is chosen such that the input voltage does not exceed the input high threshold voltage of the Schmitt trigger during the bouncing.

EXAMPLE 8.2 Choosing a Capacitor for Debouncing

In the example, $R = 1\,\text{k}\Omega$, and the bounce time is $\Delta t = 5\,\text{ms}$.



We choose C so that the voltage rise doesn't pass the Schmitt trigger input high threshold of $V_{T+} = 2\,\text{V}$ until $5\,\text{ms}$ has passed:

Timing used to calculate the capacitor value

$$\begin{aligned}
V_{T+} &\geq V_{OH}\left(1 - e^{-\Delta t/RC}\right) \\
1 - \frac{V_{T+}}{V_{OH}} &\leq e^{-\Delta t/RC} \\
\ln\left(1 - \frac{V_{T+}}{V_{OH}}\right) &\leq \frac{-\Delta t}{RC} \\
C &\geq \frac{-\Delta t}{R\ln\left(1 - \dfrac{V_{T+}}{V_{OH}}\right)} \\
&\geq \frac{-5 \times 10^{-3}}{1 \times 10^{3}\ln\left(1 - \dfrac{2}{3.3}\right)} \\
&\geq 5.367\ \mu\text{F}
\end{aligned}$$

Therefore, choose $C = 10\ \mu\text{F}$.

Switch release bounce is also removed by the capacitor

The release timing with and without the capacitor is shown below:

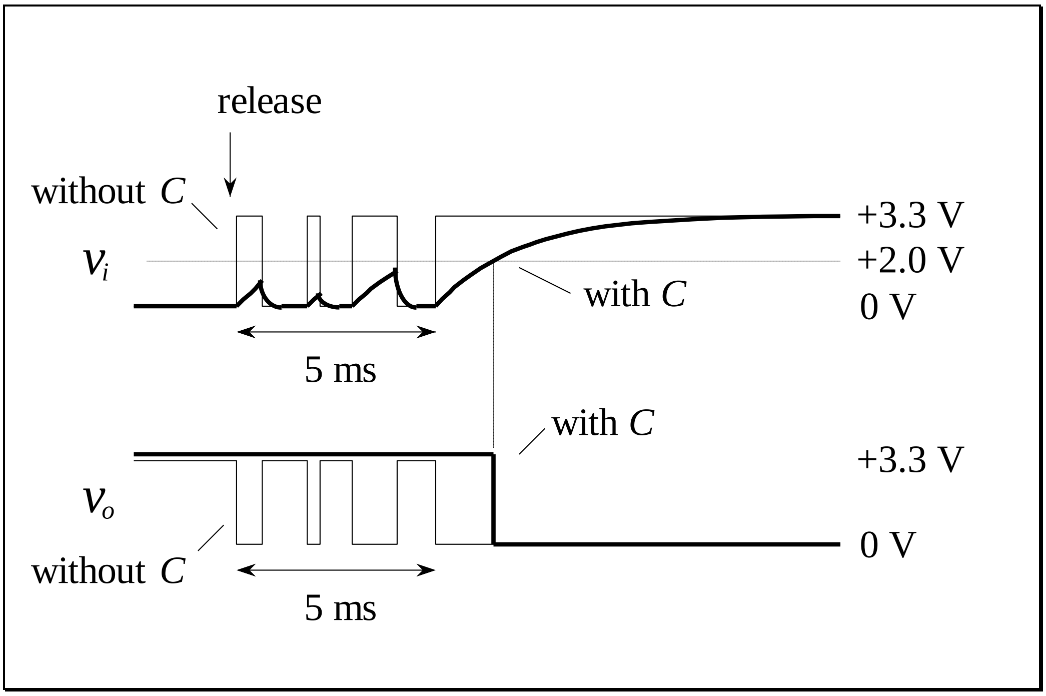
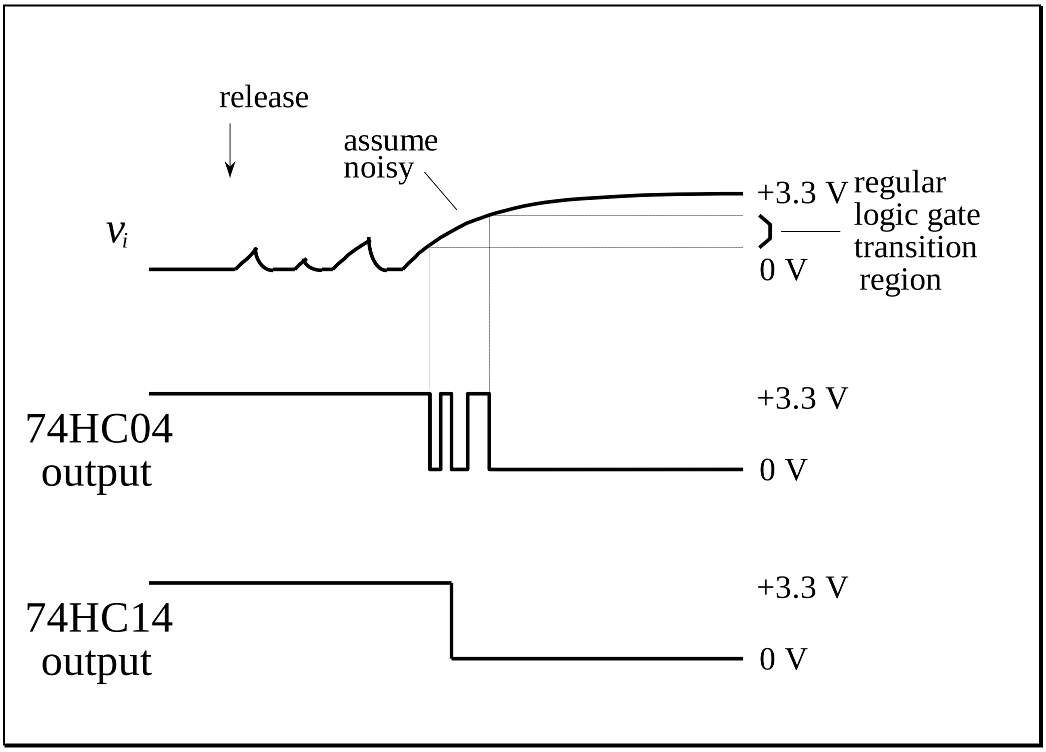


Figure 8.5

There is a significant delay from the release of the switch until the fall of the output, since the capacitor charges up slowly through the $1\,\text{k}\Omega$ resistor.

With a capacitor-based debounced switch, there is a large delay between the opening of the switch and the falling edge at the microcontroller input.	(9.3**)**

Hysteresis is required on the inverter logic gate because the capacitor causes the "logic" input to rise very slowly. Thus, while the input voltage is in the transition region between "low" and "high", a regular logic gate will be operating in its linear region, and the output will be undefined. Furthermore, any noise on the input whilst in the transition region would cause a regular gate to toggle with the noise. The hysteresis removes the extra transitions that might occur with a regular gate:



Timing showing why a logic gate with hysteresis is used instead of a regular logic gate

Figure 8.6

8.1.3 Software Debouncing

It is less expensive to remove switch bounce using software methods. It is appropriate to use a software approach because the software is fast compared to the bounce time. Typically we use a pull-up resistor to convert the switch position into a CMOS-level digital signal.

A switch interface for software debouncing

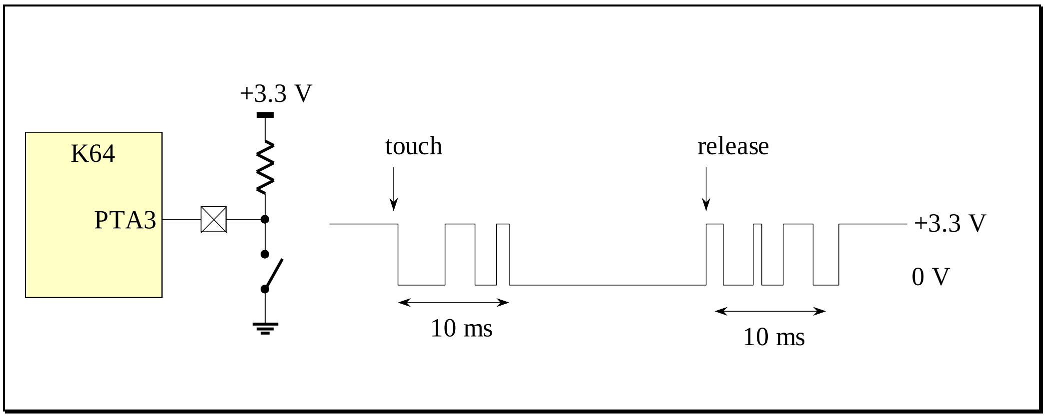


Figure 8.7

There are several ways to implement software debouncing. In the examples below, it is assumed that the switch bounce is less than 10 ms.

EXAMPLE 8.3 Software Debouncing – Simple Time Delay

In this example, the microcontroller is dedicated to the interface and does not perform any other functions while the routines are running. The routine waits for the switch to be pressed (`PTA3` low) and returns 10 ms after the switch is pressed.

We set up the FlexTimer module 0 (`FTM0`) as a free-running counter, and the clock module is 50 MHz. We use Channel 1 as a simple output compare and poll its flag:

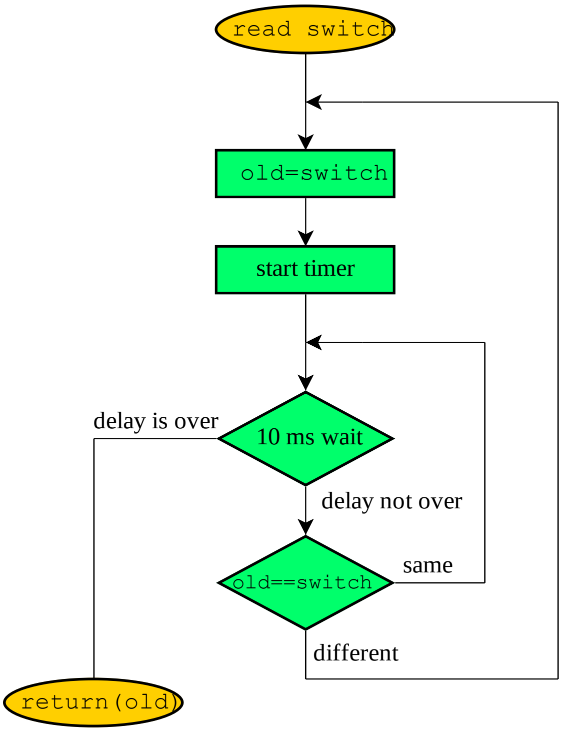
```
void WaitPress(void)
{
  // Loop here until switch is pressed
  while ((GPIOA->PDIR & 0x00000008) == 0x00000008);

  // Set the compare value 10 ms into the future
  FTM0->CONTROLS[1].CnV = FTM0->CNT + 15625;

  // Wait for switch to stop bouncing
  while ((FTM0->CONTROLS[1].CnSC & FTM_CnSC_CHF_MASK) == 0);
}

void WaitRelease(void)
{
  // Loop here until switch is released
  while ((GPIOA->PDIR & 0x00000008) == 0);

  // Set the compare value 10 ms into the future
  FTM0->CONTROLS[1].CnV = FTM0->CNT + 15625;

  // Wait for switch to stop bouncing
  while ((FTM0->CONTROLS[1].CnSC & FTM_CnSC_CHF_MASK) == 0);
}

void PortA_Init(void)
{
  // Enable clock gate for Port A to enable pin routing
  SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

  // Set Port A Bit 0 as an input
  GPIOA->PDDR &= ~0x00000001;
  // PORTA_PCR0: ISF=0, MUX=1, PE = 0, PS = 0
  PORTA->PCR0 = PORT_PCR_ISF_MASK | PORT_PCR_MUX(1);
}
```

8.12

```c
void FTM_Init(void)
{
  // Enable clock gate to FTM0 module
  SIM->SCGC6 |= SIM_SCGC6_FTM0_MASK;

  // Ensure the counter is a free-running counter
  FTM0->CNTIN = 0;
  FTM0->MOD = 0xffff;
  FTM0->CNT = 0;

  // Use the system clock for the counter (50 MHz)
  // and set the prescale to 32
  FTM0->SC |= FTM_SC_CLKS(1) | FTM_SC_PS(5);

  // Set Channel 1 as an output compare
  FTM0->CONTROLS[1].CnSC |= FTM_CnSC_MSA_MASK;

  // Enable the timer module in FTM mode
  FTM0->MODE |= FTM_MODE_FTMEN_MASK;
}
```

EXAMPLE 8.4 Software Debouncing - Waiting for Stability

In this example, the microcontroller reads the current value of the switch. If the switch is currently bouncing, it will wait for stability.



A return value of 0 means pressed (`PTA3 = 0`), and 1 means not pressed (`PTA3 = 1`). Notice that the software always waits in a "do nothing" loop for 10 ms. This inefficiency can be eliminated by placing the switch I/O in a foreground interrupt-driven thread.

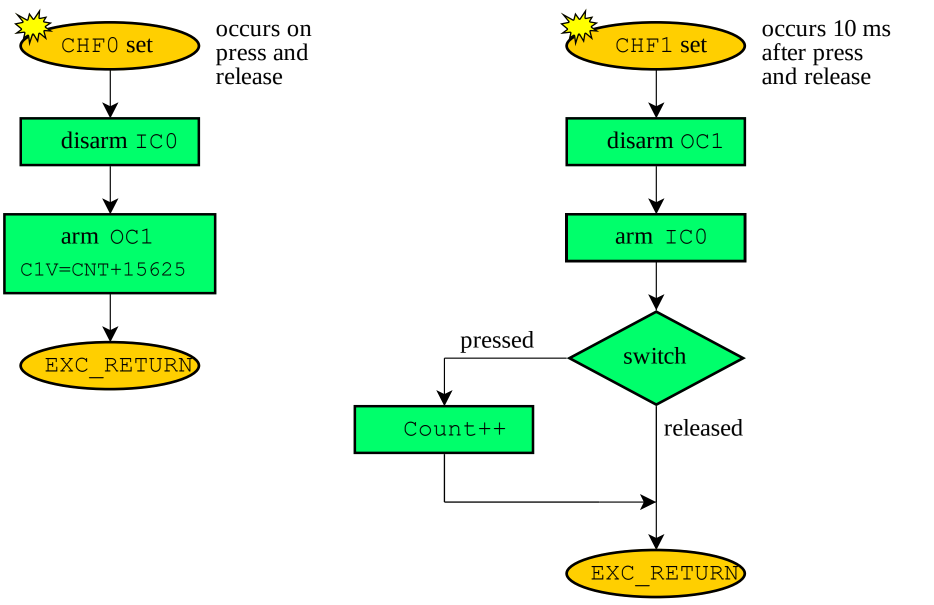
```c
uint32_t ReadPTA3(void)
{
  uint32_t old;

  // Get current value
  old = (GPIOA->PDIR & 0x00000008);
  // Set the compare value 10 ms into the future
  FTM0->CONTROLS[1].CnV = FTM0->CNT + 15625;
  // Unchanged for 10 ms?
  while ((FTM0->CONTROLS[1].CnSC & FTM_CnSC_CHF_MASK) == 0)
  {
    // Changed?
    if (old != (GPIOA->PDIR & 0x00000008))
    {
      old = (GPIOA->PDIR & 0x00000008); // new value
      FTM0->CONTROLS[1].CnV = FTM0->CNT + 15625; // restart delay
    }
  }
  return old;
}

void PortA_Init(void)
{
  // As before...
}

void FTM_Init(void)
{
  // As before...
}
```

With a software-based debounced switch, the signal arrives at the microcontroller input without delay, but software delays may occur at either touch or release. (9.4)

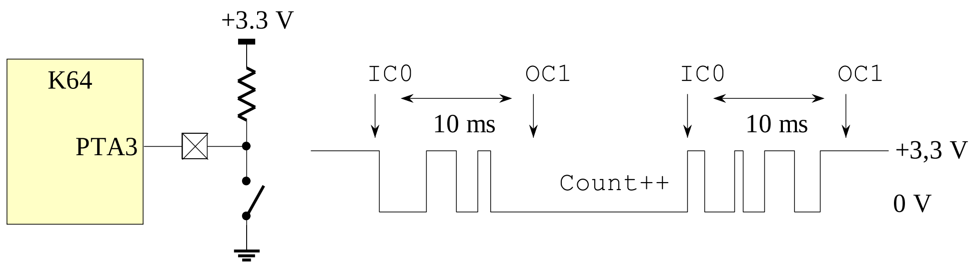
Input capture is a convenient mechanism to detect changes on the digital signal. The input capture can be configured to interrupt either on the rise , the fall or both the rise and fall. Because of the bounce, any of these modes will generate an interrupt request when the key is touched or released. A combination of input capture and output compare interrupts allows the switch interface to be performed in the background.

EXAMPLE 8.5 Software Debouncing - Interrupts

This example simply counts the number of times the switch is pressed. The `IC0` interrupt occurs immediately after the switch is pressed and released. Because the `IC0` handler disarms itself, the bounce will not cause additional interrupts. The `OC1` interrupt occurs 10 ms after the switch is pressed and 10 ms after the switch is released. At this time the switch position is stable (no bounce).



The first `IC0` interrupt occurs when the switch is first touched. The first `OC1` interrupt occurs 10 ms later. At this time the global variable `Count` is incremented. The second `IC0` interrupt occurs when the switch is released. The second `OC1` interrupt does not increment the `Count` but simply rearms the input capture system. The initialization routine initializes the system with `IC0` armed and `OC1` disarmed.



8.16

```c
// Counts the number of button pushes
// Button connected to PTA3 = Ch0 of FTM0

uint32_t Count;              // Number of button pushes
const uint16_t WAIT = 15625; // The bounce wait time

void PortA_Init(void)
{
  // Enable clock gate for Port A to enable pin routing
  SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

  // Set up PTA3 to be an input on FTM0 Ch0
  // PORTA_PCR3: ISF=0, MUX=3
  PORTA->PCR3 = PORT_PCR_MUX(3);
}

void FTM_Init(void)
{
  // Disable interrupts
  __disable_irq();

  // Enable clock gate to FTM0 module
  SIM->SCGC6 |= SIM_SCGC6_FTM0_MASK;

  // Ensure the counter is a free-running counter
  FTM0->CNTIN = 0;
  FTM0->MOD = 0xffff;
  FTM0->CNT = 0;

  // Use the system clock for the counter (50 MHz)
  // and set the prescale to 32
  FTM0->SC |= FTM_SC_CLKS(1) | FTM_SC_PS(5);

  // Set Channel 0 as an input capture
  FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_MSA_MASK
  // Capture on rising or falling edge
  FTM0->CONTROLS[0].CnSC |=   FTM_CnSC_ELSB_MASK | FTM_CnSC_ELSA_MASK;
  // Clear the IC0 flag if it is set
  FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHF_MASK;
  // Enable interrupts on Channel 0
  FTM0->CONTROLS[0].CnSC |= FTM_CnSC_CHIE_MASK;

  // Set Channel 1 as an output compare - output pin disabled
  FTM0->CONTROLS[1].CnSC |= FTM_CnSC_MSA_MASK;
  // Disable interrupts on Channel 1
  FTM0->CONTROLS[1].CnSC &= ~FTM_CnSC_CHIE_MASK;

  // Enable the timer module in FTM mode
  FTM0->MODE |= FTM_MODE_FTMEN_MASK;

  // Reset counter to zero
  Count = 0;

  // Enable interrupts
  __enable_irq();
}
```

```
void FTM0_IRQHandler(void)
{
  uint8_t channelNb;

  // Respond to IC0 interrupt if it occurred
  if (FTM0->CONTROLS[0].CnSC & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0->CONTROLS[0].CnSC & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt off
      FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHIE_MASK;
      // Turn OC1 interrupt on
      FTM0->CONTROLS[1].CnSC |= FTM_CnSC_CHIE_MASK;
      // Wait for 10 ms
      FTM0->CONTROLS[1].CnV = FTM0->CNT + WAIT;
    }
  }

  // Respond to OC1 interrupt if it occurred
  if (FTM0->CONTROLS[1].CnSC & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0->CONTROLS[1].CnSC & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0->CONTROLS[1].CnSC &= ~FTM_CnSC_CHF_MASK;
      // Turn OC1 interrupt off
      FTM0->CONTROLS[1].CnSC &= ~FTM_CnSC_CHIE_MASK;
      // Clear the IC0 flag if it is set
      FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt on
      FTM0->CONTROLS[0].CnSC |= FTM_CnSC_CHIE_MASK;
      // Increment counter if button is pressed
      if ((GPIOA->PDIR & 0x00000008) == 0)
        Count++;
    }
  }
}
```

Note that the FTM0 ISR has taken on the pre-declared name of **`FTM0_IRQHandler`** which will be automatically placed in the vector table by the linker.

EXAMPLE 8.6 Software Debouncing – Interrupts with Low Latency

The latency of the previous example is defined as the time when the switch is touched until the time when the count is incremented. Because of the delay introduced by the `OC1` interrupt, the latency is 10 ms. If we assume the switch is not bouncing (currently being touched or released) at the time of the initialization, we can reduce this latency to less than 1 μs by introducing a global Boolean variable called `SwitchPushed`. If `SwitchPushed` is false, then the switch is currently not pushed and the software is searching for a touch. If `SwitchPushed` is true, then the switch is currently pushed and the software is searching for a release.

```
// Counts the number of button pushes
// Button connected to PTA3 = Ch0 of FTM0

uint32_t Count;              // Number of button pushes
const uint16_t WAIT = 15625; // The bounce wait time
bool SwitchPushed;           // State of button
  // bFALSE means open,  looking for a touch
  // bTRUE means closed, looking for release

void FTM_Init(void)
{
  // Disable interrupts
  __disable_irq();

  // Enable clock gate to FTM0 module
  SIM->SCGC6 |= SIM_SCGC6_FTM0_MASK;

  // Ensure the counter is a free-running counter
  FTM0->CNTIN = 0;
  FTM0->MOD = 0xffff;
  FTM0->CNT = 0;

  // Use the system clock for the counter (50 MHz)
  // and set the prescale to 32
  FTM0->SC |= FTM_SC_CLKS(1) | FTM_SC_PS(5);

  // Set Channel 0 as an input capture
  FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_MSA_MASK
  // Capture on rising or falling edge
  FTM0->CONTROLS[0].CnSC |=   FTM_CnSC_ELSB_MASK | FTM_CnSC_ELSA_MASK;
  // Clear the IC0 flag if it is set
  FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHF_MASK;
  // Enable interrupts on Channel 0
  FTM0->CONTROLS[0].CnSC |= FTM_CnSC_CHIE_MASK;

  // Set Channel 1 as an output compare - output pin disabled
  FTM0->CONTROLS[1].CnSC |= FTM_CnSC_MSA_MASK;
  // Disable interrupts on Channel 1
  FTM0->CONTROLS[1].CnSC &= ~FTM_CnSC_CHIE_MASK;
```

```
  // Enable the timer module in FTM mode
  FTM0->MODE |= FTM_MODE_FTMEN_MASK;

  // Reset counter to zero
  Count = 0;

  // Get initial state of button
  SwitchPushed = ((GPIOA->PDIR & 0x00000008) == 0);

  // Enable interrupts
  __enable_irq();
}

void FTM0_ IRQHandler (void)
{
  uint8_t channelNb;

  // Respond to IC0 interrupt if it occurred
  if (FTM0->CONTROLS[0].CnSC & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0->CONTROLS[0].CnSC & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt off
      FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHIE_MASK;
      // Turn OC1 interrupt on
      FTM0->CONTROLS[1].CnSC |= FTM_CnSC_CHIE_MASK;
      // Wait for 10 ms
      FTM0->CONTROLS[1].CnV = FTM0->CNT + WAIT;
      // An edge occurred - toggle state
      SwitchPushed = !SwitchPushed;
      // If a touch occurred, increment the counter
      if (SwitchPushed)
        Count++;
    }
  }

  // Respond to OC1 interrupt if it occurred
  if (FTM0->CONTROLS[1].CnSC & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0->CONTROLS[1].CnSC & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0->CONTROLS[1].CnSC &= ~FTM_CnSC_CHF_MASK;
      // Turn OC1 interrupt off
      FTM0->CONTROLS[1].CnSC &= ~FTM_CnSC_CHIE_MASK;
      // Clear the IC0 flag if it is set
      FTM0->CONTROLS[0].CnSC &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt on
      FTM0->CONTROLS[0].CnSC |= FTM_CnSC_CHIE_MASK;
    }
  }
}
```

8.20

Note that the FTM0 ISR has taken on the pre-declared name of `FTM0_IRQHandler` which will be automatically placed in the vector table by the linker.

Now the latency is simply the time required for the microcontroller to recognize and process the input capture interrupt. Assuming there are no other interrupts, this time is less than 50 cycles.

8.2 Analog to Digital Conversion

An analog-to-digital converter (ADC) is used to quantize an external analog signal so as to represent it digitally. If the samples of an analog signal are taken at a sufficiently high rate, then the samples furnish enough information for the analog signal to be reconstructed exactly. Once the analog signal has been converted to a digital form, it can be filtered, manipulated, and processed. The processed signal can then be converted back to an analog signal through the use of a digital-to-analog converter (DAC).

8.2.1 ADC Module

The Analog-to-Digital Converter (ADC) module has the capability of sampling up to four pairs of differential and 24 single-ended analog channels with either 8-bit, 10-bit, 12-bit or 16-bit quantization. The output format is in 2's complement 16-bit sign extended for differential mode and unsigned 16-bit for single-ended modes. The signed notation is useful when converting shifted bipolar inputs (a shifted bipolar input will have its "zero" at exactly half way between the minimum and maximum conversion values). The ADC can also be set up to automatically sample a sequence of analog channels.

A simplified block diagram of the ADC module is shown below:

A simplified ADC block diagram

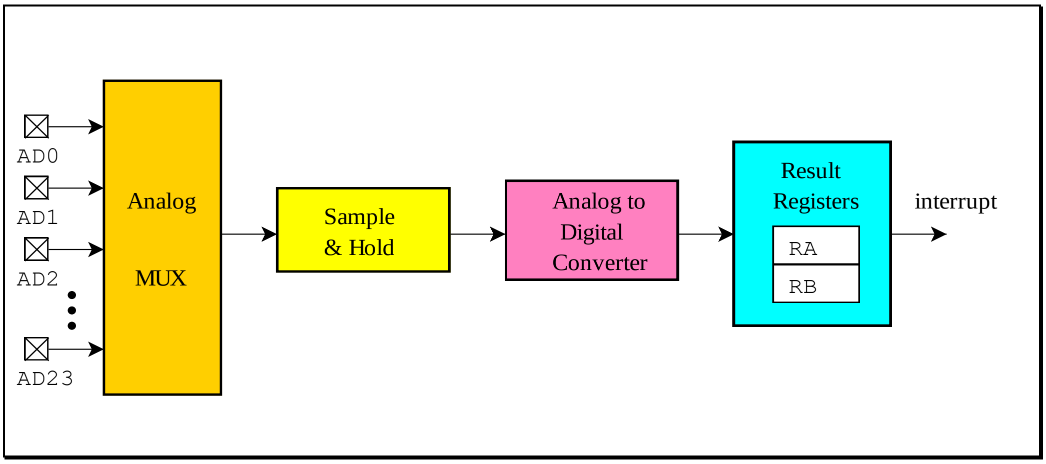


Figure 8.8

The analog multiplexer is just a switch

The analog multiplexer (MUX) is just an analog switch that connects one of the analog channels to the sample-and-hold (S/H) block.

The sample and hold block is used to charge a sample capacitor to a voltage very close to the applied analog voltage

The sample-and-hold circuit consists of a *sample capacitor* and a buffer. It is important to take into account the characteristics of the S/H block in the design of the analog interface hardware external to the microcontroller. The external hardware's output resistance and the sample capacitor form a first-order lowpass filter, also known as a single time constant (STC) circuit, or just a lowpass *RC* circuit. This first-order filter determines the amount of time that is needed to charge the sample capacitor to a voltage that is almost equal to the true analog voltage.

Analog input circuit, showing external resistance and sample capacitor that form a first-order lowpass filter

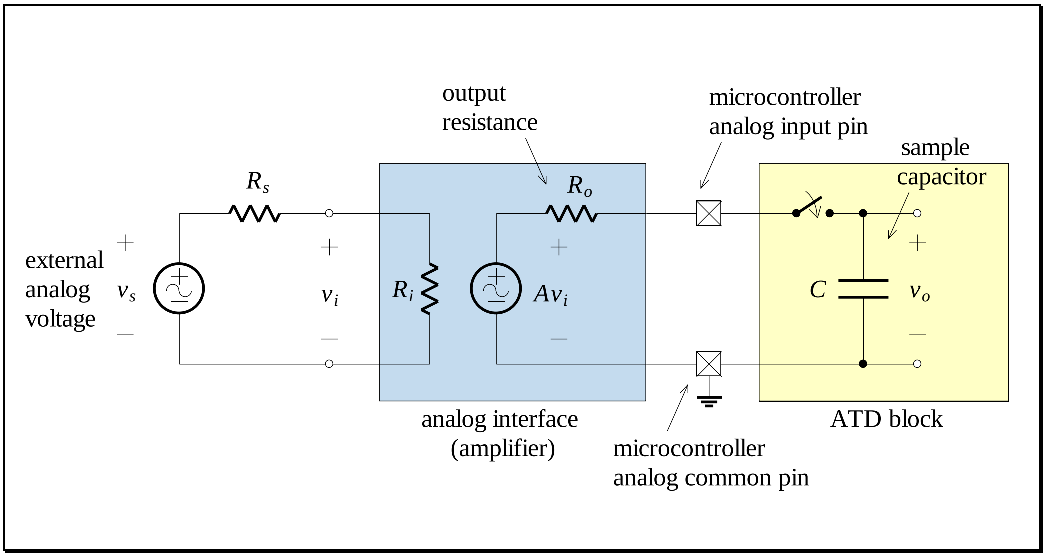


Figure 8.9

The voltage held on the sample capacitor is fed into the analog-to-digital converter. There are many types of ADC – the type used in the K64 series of microcontrollers is a successive approximation architecture (SAR). It functions by comparing the stored analog sample voltage with a series of digitally generated analog voltages. By following a binary search algorithm, the ADC locates the approximating voltage that is nearest to the sampled voltage.

The results of the analog-to-digital conversions are stored in separate result registers. The completion of an analog-to-digital conversion can be used to trigger an interrupt.

Further Information

A complete description of the ADC module can be found in Chapter 35 of Freescale's *K64 Sub-Family Reference Manual*.

8.3 Digital to Analog Conversion

The K64 has two on-board 12-bit low-power, general-purpose digital-to-analog converters. The output of the DAC can be placed on an external pin or set as one of the inputs to the analog comparators, or ADC.

A simple DAC can also be made from a pulse width modulated (PWM) waveform. If a PWM waveform is passed through a lowpass filter, and the PWM has a sufficiently high frequency, then the output of the filter will be a smooth analog waveform corresponding to the average value of the PWM taken over many periods.

8.3.1 Pulse Width Modulator

A pulse width modulator is a device which varies the duty cycle (the "on time" versus "total time", in percent) of a square wave. They can be used to turn transistors on and off in an external circuit to drive devices such as DC motors and 3-phase AC motors. They can also be used to create a simple digital-to-analog converter.

A simplified PWM block diagram

In the K64, PWM waveforms are generated by the FlexTimer module (FTM). A conceptual block diagram of the PWM functionality is shown below:

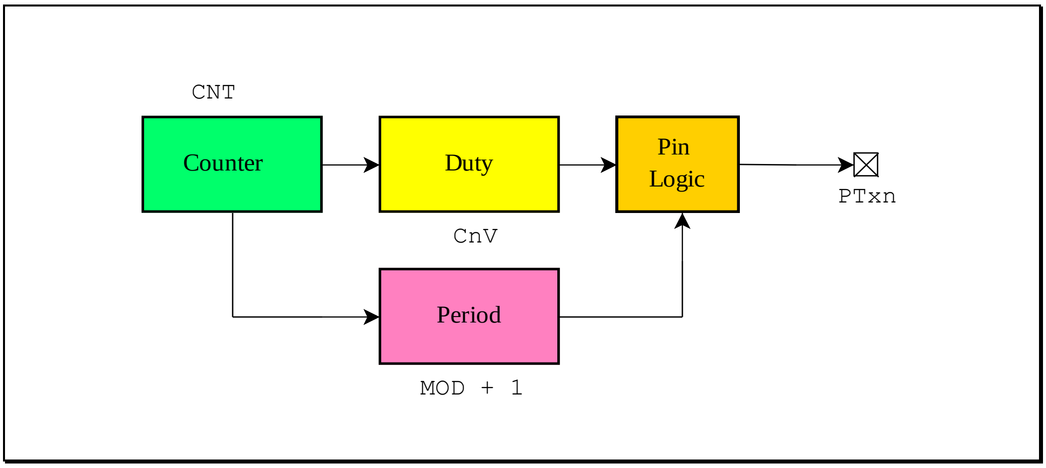


Figure 8.10

The PWM uses a duty value and a period value

Each of the 8 channels of the FTM can be set up as a PWM. Each channel uses the common 16-bit counter `CNT`, and the common modulus value held in `MOD`. For each channel, the counter compares to two values: a *duty* value held in `CnV`; and the common *period* value held in `MOD`. In its simplest mode of operation, known as edge-aligned PWM, the output is set high when the counter equals the period value, and the output is set low when the counter equals the duty value.

PWM waveform generation

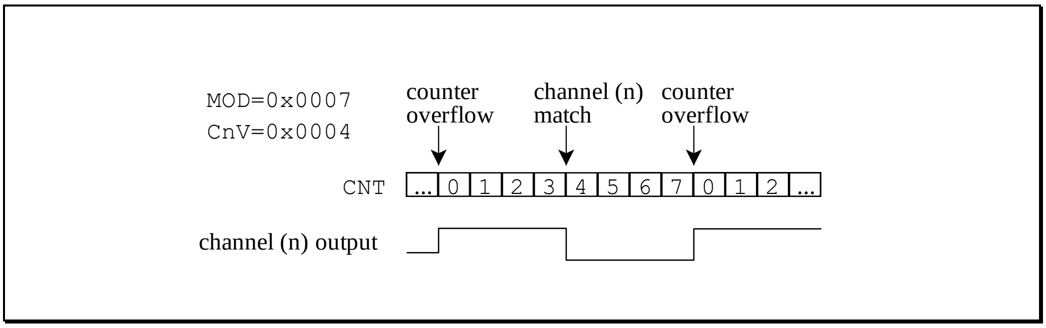
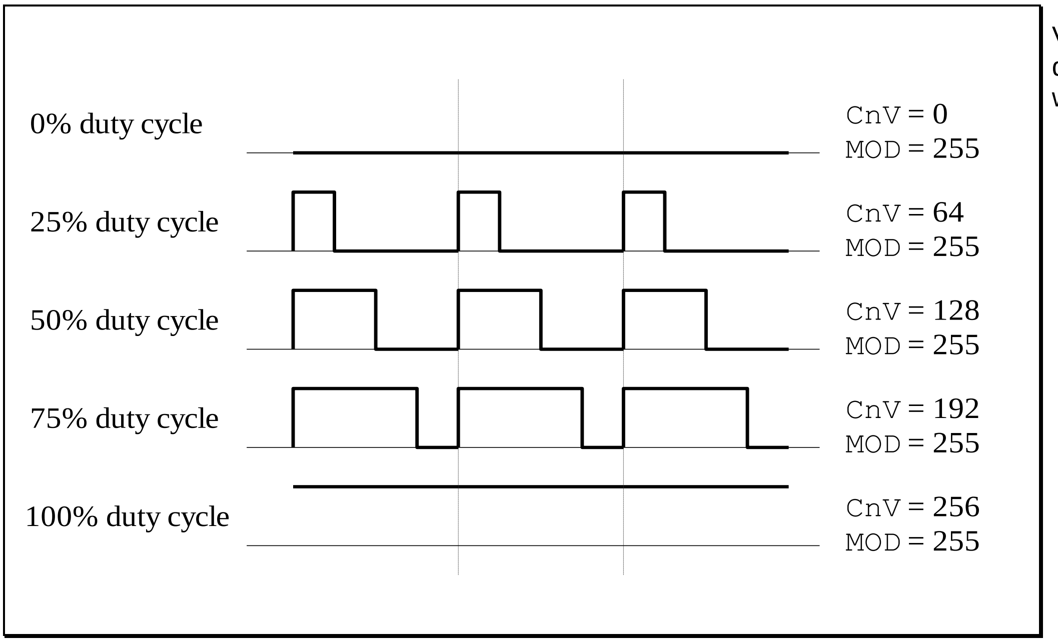


Figure 8.11

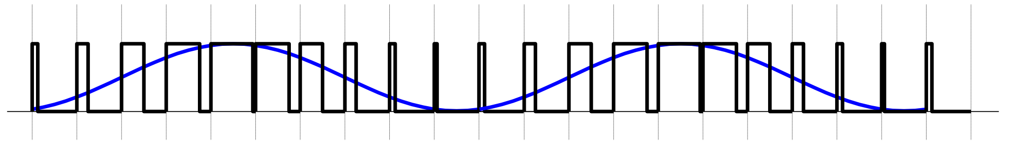
Various constant duty cycle waveforms are shown below:



Various constant duty cycle PWM waveforms

Figure 8.12

The generation of fixed duty cycle square waves is only one application of the PWM function. It is more generally used to *modulate* the pulse width of the square wave. Such a waveform is shown below:



PWM waveform that approximates a sine wave

A PWM waveform that approximates a sine wave

Figure 8.13

Such a PWM waveform can be used to generate a continually varying analog signal. The area under the PWM waveform approximates the area under the desired analog waveform. The PWM waveform needs to be filtered by external hardware to remove the high frequency components, and to leave just the fundamental and DC components. You can think of the analog output as responding to the *average* value of the PWM waveform (over a small time interval).

An external filter is usually not required when the device being driven provides an inherent filtering function. For example, a DC motor, which exhibits mechanical inertia, cannot respond to the rapid fluctuations of the PWM waveform – but it can respond to the slowly varying "average" value of the waveform. In this case, the DC motor speed would be seen to vary sinusoidally. There may also be an audible "hum" or "”buzz" due to the high frequency components being within the range of human hearing (20 Hz – 20 kHz). If such a hum is undesirable, then the designer can increase the frequency of the PWM wave so that the high frequency components are out of audio range.

If an analog voltage is required, a simple buffered *RC* circuit can be used:

Generation of an analog voltage via a PWM output

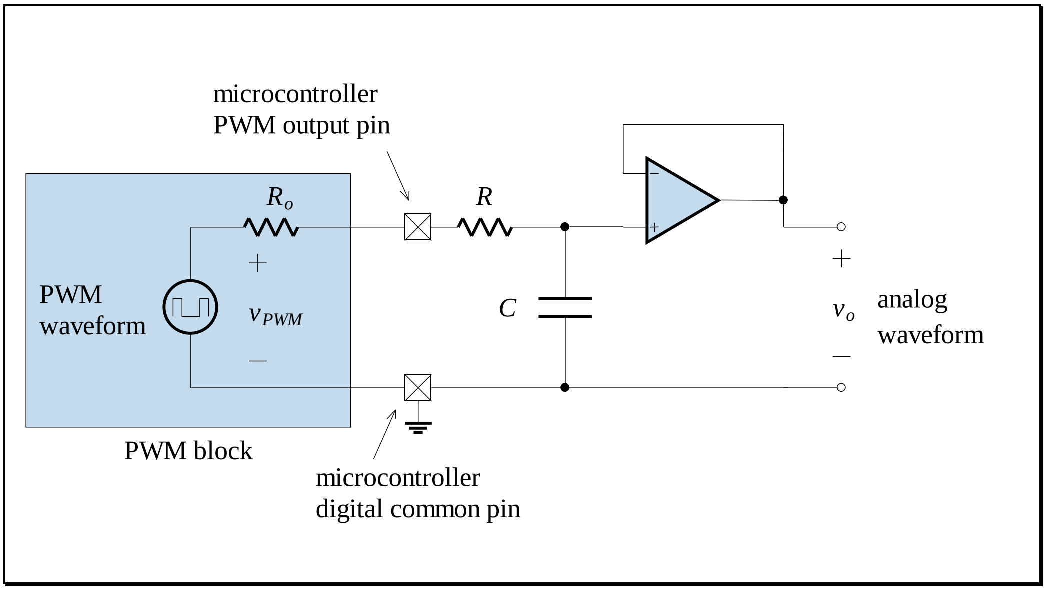


Figure 8.14

Further Information

A complete description of the FlexTimer Module can be found in Chapter 40 of Freescale's *K64 Sub-Family Reference Manual*.

9 Fixed-Point Processing

Contents

9.2

Introduction

Fixed-point means 'integer'

Most microprocessors are fixed-point devices – they only have support for arithmetic with integers. For example, the ARM® Cortex®-M3 processor does not have a floating-point unit (FPU). The Cortex®-M4 is a relatively special MCU because it has the option to include hardware that directly supports single-precision floating-point numbers – but at the expense of increased cost and power consumption. PC processors since the 80486DX (released in 1989) have a "math coprocessor on chip", and all subsequent generations have included an FPU. This is why PCs are fast, and expensive – a large proportion of the die area and power consumption of the CPU is taken up by the FPU.

If you do not use the FPU then compiled code can be used on another Cortex-M microcontroller product that does not have FPU support, such as the M3. Floating-point operations can be emulated in software on a fixed-point processor using special maths libraries, but the resulting overhead results in programs that run 40-100 times slower than a program that uses just fixed-point operations.

Fixed-point calculations are important when time is important

Therefore, when cost, power consumption and speed (i.e. time) is of primary importance in a design, it is necessary to perform arithmetic operations using a fixed-point processor. We therefore need to examine processing techniques that use integers but provide an interpretation of the resulting numbers as having fractional parts.

9.1 Q Notation

Fixed-point calculations are capable of performing fractional mathematics if an implied binary point is used in the *interpretation* of the integer used to represent a fractional quantity. In accordance with accepted digital signal processing (DSP) notation, we use what is called “Q notation”. The “Q” stands for quotient, or a number with a fractional part.

Most quantities in signal processing use either 16 bits or 32 bits for their representation. To express a fractional part, an implied binary point is required for each quantity. It is up to us as designers to keep track of these implied binary points throughout any and all calculations. For each quantity, we express its fractional part with the notation *m*Q*n* where *n* is an integer ranging from 0 to 16 for 16-bit quantities or 0-32 for 32-bit quantities. The *m* tells how many bits are used in total, either 16 or 32. The *n* tells how many bits are to the right of the implied binary point.

Just like a decimal point, a binary point interprets digits to the right of it as being negative powers of the base. A comparison of a decimal number and its equivalent binary number is given below:

Comparison of a decimal number and equivalent binary number

Decimal number	Equivalent binary number
5.625	101.101
↑ Decimal point	↑ Binary point
$=5\times10^{0}+6\times10^{-1}+2\times10^{-2}+5\times10^{-3}$	$=1\times2^{2}+0\times2^{1}+1\times2^{0}+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}$

Figure 9.1

Mapping integers to fractional quantities

For example, a 6Q3 number implies 3 bits to the right of the implied binary point. A mapping of the CPU's integer values to quantities that we interpret is made as follows:

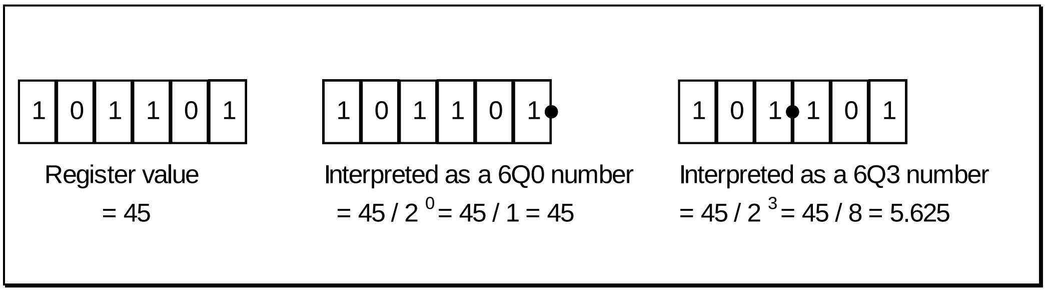


Figure 9.2

From this, it should be apparent that to interpret a register value as a *m*Q*n* value, we simply divide the raw value by 2^n. To store a fractional number in *m*Q*n* notation, we multiply it by 2^n and truncate or round the answer to an integer. This inherent round-off error cannot be prevented.

For example, if we wished to store the number 5.628 in 16Q3 notation, we get:

$$\begin{aligned} &5.628 \times 2^3 = 5.628 \times 8 = 45.024 \\ &\therefore \text{store as } 45 \end{aligned} \tag{9.1}$$

In this case it is impossible to distinguish between 5.625 and 5.628 in 16Q3 notation.

The resolution of *m*Q*n* numbers can therefore be expressed as 2^{-n}. For example, in 16Q3 notation the resolution of the stored numbers is $2^{-3} = 0.125$. Every number in Q3 notation will be a multiple of 0.125. Clearly it is desirable to have a large n to store fractional values with the greatest accuracy. It is in fact impossible to store 5.628 exactly (no round-off error). The best we can do using 32-bits is to store the integer part (5 in 5.628) using the least amount of bits (3 in this case) and use the rest for the fractional part.

We therefore would use a 32Q29 number:

5.628 as a 32Q29 number

1	0	1	1	0	1	0	0	0	0	0	1	1	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	0

Register value = 3021509492

1	0	1.	1	0	1	0	0	0	0	0	1	1	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	0

Interpreted as a Q29 number = 3021509492 / 2^{29} = 3021509492 / 536870912 = 5.6279999999

Figure 9.3

The reason we can't store this number exactly is because when we multiply 5.628 by successive powers of two to obtain an integer, the last digits form a cyclic pattern, that will never reach a multiple of 10:

$$\begin{aligned} &5.628 \times 2 = 11.256 \\ &11.256 \times 2 = 22.512 \\ &22.512 \times 2 = 45.024 \\ &45.024 \times 2 = 90.048 \\ &90.048 \times 2 = 180.096 \\ &180.096 \times 2 = 360.192 \\ &\text{etc.} \end{aligned} \tag{9.2}$$

This shouldn't really worry us, because a 32Q29 number has a resolution of $2^{-29} = 1.8626451 \times 10^{-9}$. The error in storing the above number as shown is therefore less than 0.00000003 %.

As an aside, we should not forget that using floating-point numbers does not increase our accuracy. Accuracy is determined purely by the number of bits, not in the *way* the number is stored. It shocks some people to find that floating-point units cannot store the number 0.1, precisely because of the problem stated above. However, the floating-point number can get very close to 0.1 in the same way that we can get very close to 5.628.

9.2 Other Notations

The Q notation is convenient because it expresses a number as powers of two. It will be shown later that this provides an efficient method to convert numbers from one Q representation to another.

We can also express numbers using a base other than 2. For example, suppose we say that the number 1000 is to be interpreted as 1. We say that the number has 1000 as a base, or unity value, and that 1000 = 1 per unit (p.u.). The number 5.628 in this method would be represented as 5628, which is exact. Why don't we use this method over Q notation? The answer is because other numbers can now not be represented exactly. Remember – the fundamental limit in accuracy is set by the number of bits, and not how they are interpreted.

Complications arise in calculations involving multiplications and divisions. For example, multiplying two numbers with a base of 1000 produces a number whose base is 1000000. To *normalise* this result back to 1000, the result would have to be divided by 1000 – this division is expensive in terms of CPU time and is to be avoided.

It should be noted that Q notation is just representing numbers with bases that are multiples of two. For example a 16Q3 number is a number with a base or p.u. value of 8.

9.3 Fixed-Point Calculations

9.3.1 Multiplication

Multiplying two numbers together changes the base or “per unit” value. For example, consider the following multiplication:

Multiplication changes the base value

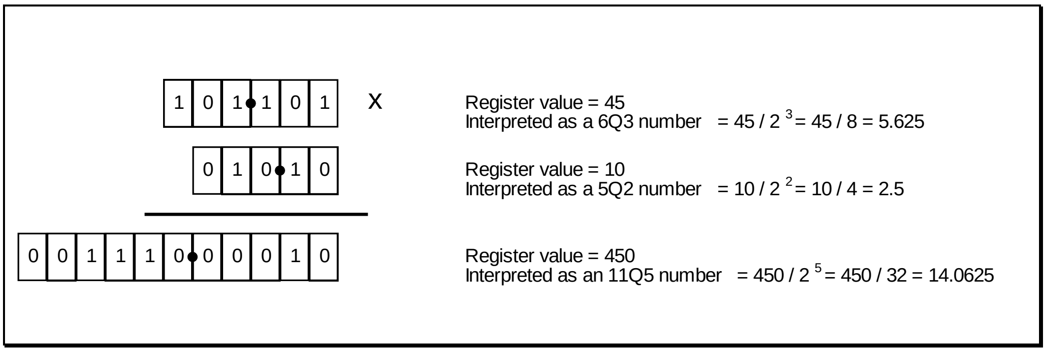


Figure 9.4

Two things happen – 1) the length of the result is equal to the sum of the lengths of the two multiplicands and 2) the Q notation of the result is equal to the sum of the individual Q notations.

We can state this formally as follows:

$$m\mathrm{Q}n \times i\mathrm{Q}j = (m+i)\mathrm{Q}(n+j) \tag{9.3}$$

The ARM® Cortex®-M4 processor has two multiply instructions – one giving a 32-bit result, and one giving a 64-bit result. The C compiler will not automatically increase the result length – two 32-bit operands will theoretically give a 64-bit result for multiplication, but the C compiler will use the instruction with a 32-bit result to preserve “type”. Even if we could arrange for a 64-bit result (we can with assembly language), we can’t multiply the result by another number, because that would involve a 64-bit x 32-bit multiplication which is not directly supported by a 32-bit CPU. We have to emulate what a floating-point unit would do – *normalise*. This means the 64-bit result must be converted back to a 32-bit number that has some arbitrary Q notation. For example, if we wished to convert a result from a 64Q5 number (base 32) to a 32Q3 (base 8) number, we shift it right 2 bits (divide by 4 which is the amount the base has changed), and only keep the lower 32 bits. We should note that in shifting, we inevitably lose accuracy. This is the price paid for maintaining successive calculation results within 32-bits.

We can see now why Q notation is efficient – normalisation is carried out by shifts which are very quick in terms of CPU time (much quicker than a divide – 1 cycle time for a shift, compared with 2-12 cycles for a divide on the K64).

9.3.2 Division

For division, we similarly have:

$$m\mathrm{Q}n \div i\mathrm{Q}j = m\,\mathrm{Q}(n-j), \quad m > i \quad n > j \tag{9.4}$$

For example, a 32Q16 number divided by a 16Q8 number results in a 32Q8 quotient.

9.3.3 Addition

Normalisation before addition

Additions must be performed with numbers of the same Q notation. If they are different, then normalisation to the larger base is required. For example, to add a 6Q3 number and a 5Q2 number, we have to shift the 5Q2 number to the left by one to create a 6Q3 number before adding:

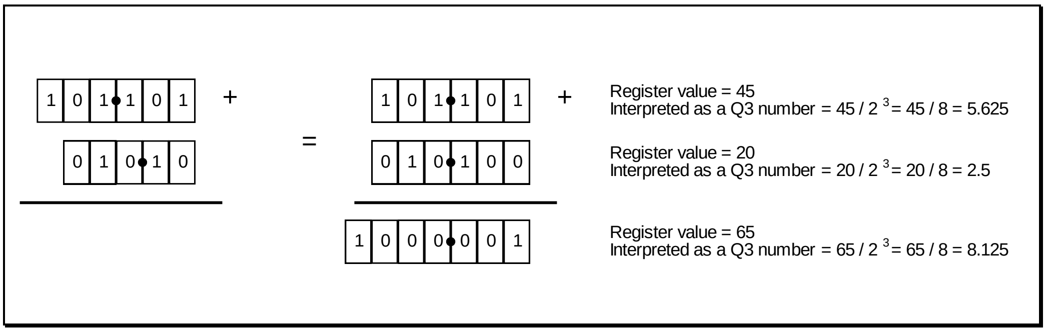


Figure 9.5

9.3.4 Subtraction

Similarly, subtraction requires normalisation of the bases so that the larger base is common.

EXAMPLE 9.1 Fixed-Point Calculations

We will develop the equations that K64 software could need to implement a digital scale. Assume the range of a position measurement system is 0 to 3 m, and the system uses the K64's ADC to perform the measurement. We will assume that the ADC has been put into single-ended 10-bit mode so that the digital output varies from 0 to 1023. Suppose also that the analog input range is 0 to +3.3 V. Let x be the distance to be measured in metres, V_{in} be the analog voltage in volts and N be the 10-bit digital ADC output. Then the equations that relate the variables are:

$$V_{in} = 3.3 * N / 1024 \qquad \text{and} \qquad x = 3\,\mathrm{m} * V_{in} / 3.3\,\mathrm{V}$$

Thus:

$$x = 3 * N / 1024 = 0.0029296875 \; * N \qquad \text{where } x \text{ is in m}$$

From this equation, we can see that the smallest change in distance that the ADC can detect is about 0.003 m. In other words, the distance must increase or decrease by 0.003 m for the digital output of the ADC to change by at least one number. It would be inappropriate to save the distance as an integer, because the only integers in this range are 0, 1, 2 and 3. To save power, we decide not to use the K64's FPU and therefore the distance data will be saved in fixed-point format. Decimal fixed-point is chosen because the distance data for this distance-meter will be displayed for a human to read. A fixed-point resolution of 0.001 m could be chosen, because it matches the resolution determined by the hardware. The table below shows the performance of the system with the resolution set to 0.001 m. The table shows us that we need to store the fixed-point number in a signed or an unsigned 16-bit variable.

x (m) distance	V_{in} (V) analog input	N ADC input	I internal representation	Approximation `(41 * N + 7) / 14`
0	0.000	0	0	0
0.003	0.003	1	3	3
0.600	0.660	205	600	600
1.500	1.650	512	1500	1499
3.000	3.300	1023	3000	2996

9.10

It is very important to carefully consider the order of operations when performing multiple integer calculations. There are two mistakes that can happen. The first error is *overflow*, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. The following fixed-point calculation, although mathematically correct, has an overflow bug:

```
I = (3000 * N) / 1024;
```

because when `N` is greater than 21, `3000*N` exceeds the range of a 16-bit unsigned integer. If possible, we try to reduce the size of the integers. In this case, an approximate calculation can be performed without overflow

```
I = (41 * N) / 14;
```

You can add one-half of the divisor to the dividend to implement rounding. In this case:

```
I = (41 * N + 7) / 14;
```

The addition of "`7`" has the effect of rounding to the closest integer.

For example, when `N = 4`, the calculation `(41 * 4) / 14 = 11`, whereas the "`(41 * 4 + 7) / 14`" calculation yields the better answer of `12`.

No overflow occurs with this equation using unsigned 16-bit maths, because the maximum value of `41 * N` is 41943. If you cannot rework the problem to eliminate overflow, the best solution is to use promotion. Promotion is the process of performing the operation in a higher precision. For example, in C we cast the input as **`unsigned long`**, and cast the result as **`unsigned short`**:

```
I = (unsigned short)((3000 * (unsigned long)N)/1024);
```

Again, you can add one-half of the divisor to the dividend to implement rounding. In this case:

```
I = (unsigned short)((3000 * (unsigned long)N + 512) / 1024);
```

The other type of error we may experience with fixed-point arithmetic is called *drop out*. Drop out occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. It is very important to divide last when performing multiple integer calculations. If you divided first:

```
I = 41 * (N / 14);
```

then the values of `I` would be only 0, 41, 82, … or 2993.

The display algorithm for the unsigned decimal fixed-point number with 0.001 resolution is simple:

1) display `(I / 1000)` as a single digit value

2) display a decimal point

3) display `(I % 1000)` as a three-digit value

4) display the units "m"

9.3.5 Fixed-Point Operations Using a Universal 32Q16 Notation

Finding the optimum choice of Q notation for a fixed-point variable requires knowing what range of values it will have during execution. When range and resolution requirements are modest, however, a simple approach is to use 32 bits for all fixed-point numbers, with 16 bits in both the whole and the fractional parts, i.e. a 32Q16 notation:

32Q16 notation

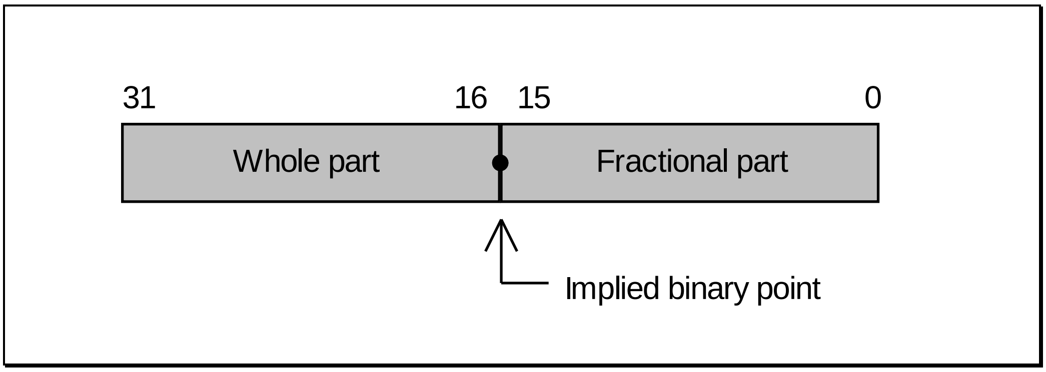


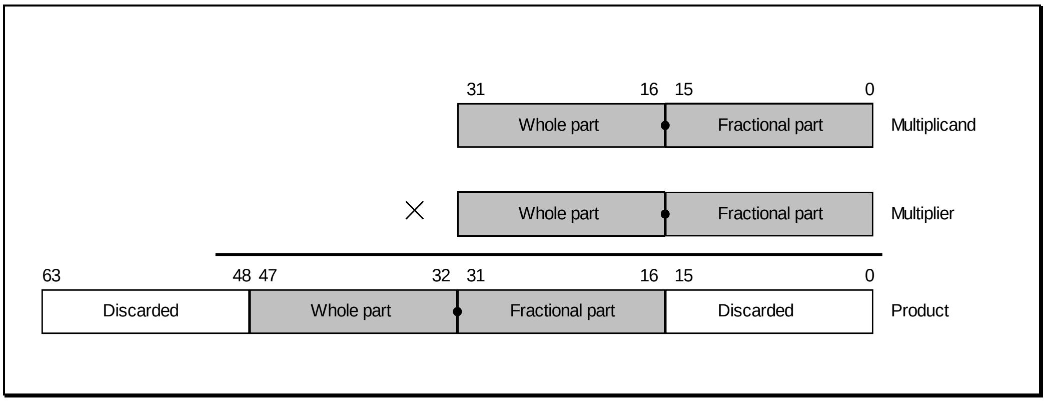
Figure 9.6

This is the method used by Sony (original Playstation) and Nintendo (DS, Gamecube, Gameboy Advance) in their 3D graphics engines to achieve fast processing performance without an FPU.

With all operands using the same notation, addition and subtraction no longer require pre-alignment of operands. Multiplication and division, however, still require *some* adjustment or else the result will not have the same notation as the operands. Remember that when you multiply two fixed-point operands together, their Q notations add:

$$32\text{Q}16 \times 32\text{Q}16 = 64\ \text{Q}32 \tag{9.5}$$

What we need is a product in 32Q16 format. In other words, the integer product needs to be right-shifted by 16 bits. Multiplying the two 32-bit integers produces a 64-bit product, with an implied binary point in the middle. Right-shifting this product by 16 bits and then putting the result back into a 32-bit location means we are discarding 16 bits from each end of the integer product:



Using integer multiplication to produce a fixed-point product

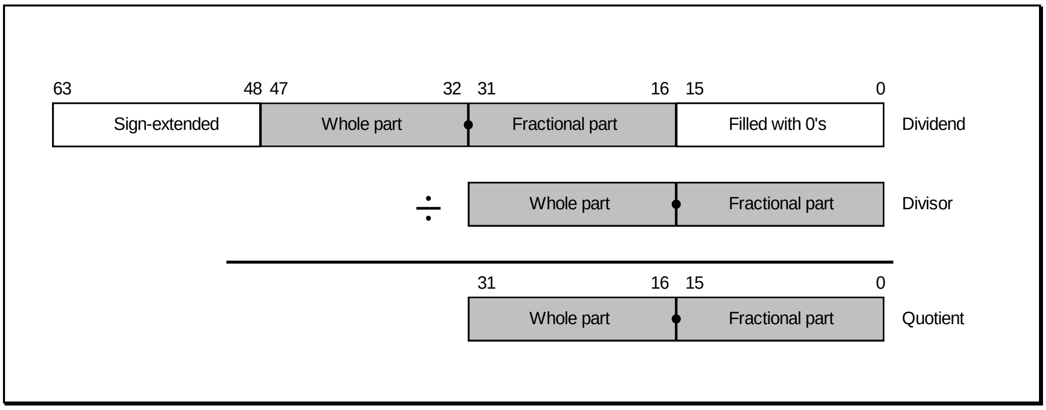
Figure 9.7

Discarding the least significant 16 bits simply causes some loss of precision; discarding the most significant 16 bits requires imposing a maximum magnitude restriction on the operands to avoid overflow.

When you *divide* one 32Q16 fixed-point operand by another, we require the result to be a 32Q16 number. We therefore need a 64Q32 dividend, since:

$$64Q32 \div 32Q16 = 32\,Q16 \tag{9.6}$$

We create a 64Q32 dividend by sign extending the original 32Q16 dividend, and then left-shifting by 16 bits. The division is then done with a 64-bit dividend and a 32-bit divisor, to give a 32-bit quotient:



Using integer division to produce a fixed-point quotient

Figure 9.8

EXAMPLE 9.2 Fixed-Point Calculations Using Universal 32Q16 Notation

We can use fixed-point algorithms to perform complex operations using the integer functions of our K64. For example, consider the following digital filter calculation:

```
y = x - 0.0532672 * x1 + x2 + 0.0506038 * y1 - 0.9025 * y2;
```

In this case, the variables `y`, `y1`, `y2`, `x`, `x1`, and `x2` are all 32Q16 fixed-point integers, and we need to express the constants in 32Q16 fixed-point format. The value -0.0532672 is approximated by $-0.0532672 \times 65536 \approx -3491$. The value 0.0506038 will be approximated by $0.0506038 \times 65536 \approx 3316$. Lastly, the value -0.9025 will be approximated by $-0.9025 \times 65536 \approx -59146$. The fixed-point implementation of this digital filter is:

```
int64_t t1, t2, t3, t4;

t1 =  -3491 * (int64_t)x1;
t2 =   3316 * (int64_t)y1;
t3 = -59146 * (int64_t)y2;
t4 = t1 + t2 - t3;
y = x + x2 + (int32_t)(t4 >> 16);
```

Note that since we are using C types, we need to allocate space for a 64-bit product, and thus the 32-bit integer variables are promoted and sign-extended to 64-bits using a typecast. If we did not do this, then the multiplication of two 32-bit quantities may overflow the 32-bit storage space.

The approximations of the constants using 32Q16 notation may be unsuitable if they do not give us enough resolution. In that case, we have to sacrifice speed and use a different non-power-of-2 base or increase the resolution of the Q notation numbers.

9.4 Square Root Algorithm for a Fixed-Point Processor

The evaluation of the square root of a number using integer arithmetic is a common operation in many embedded systems. For example, in the calculation of RMS quantities, such as voltage and current, a square root is involved. Any time a complex number is used (such as in an FFT), it is convenient to know its magnitude, which involves Pythagoras' Theorem and a square root operation.

There are numerous algorithms used on fixed-point processors to find square roots. Many of them are optimised in assembly language, and many are optimised on a "per bit" basis, i.e. a square root may take 3 or 4 instruction cycles per bit, plus a few cycles extra for initialising/ finalising. We will examine a more general technique which may be optimal in scenarios where a good estimate of the root is already known (as occurs often in practice).

To evaluate the square root of a number, we can use Newton's method to solve the equation:

$$f(x) = R - x^2 = 0 \tag{9.7}$$

where R is the number whose square root we wish to evaluate. According to a first-order Taylor series approximation of any function, we have:

$$f(x+h) \approx f(x) + hf'(x) \tag{9.8}$$

If we have an estimate of the square root, x_*, then we can use the above formula to determine an h to add to x_*, which will hopefully be a better estimate of the square root. We therefore seek an h that satisfies:

$$\begin{aligned} f(x_* + h) &\approx 0 \\ f(x_*) + hf'(x_*) &= 0 \\ h &= \frac{-f(x_*)}{f'(x_*)} \end{aligned} \tag{9.9}$$

This process is then repeated in an iterative manner until a desired accuracy is reached:

$$\begin{aligned} x_* &= x_* + h \\ \lim_{n\to\infty} x_* &= x \end{aligned} \tag{9.10}$$

Applying the above analysis to Eq. (9.7) gives a formula for the new estimate of the square root as:

$$\begin{aligned} x_* &= x_* - \frac{f(x_*)}{f'(x_*)} \\ &= x_* - \frac{R - x_*^2}{-2x_*} \\ &= x_* + \frac{R}{2x_*} - \frac{x_*}{2} \\ &= \frac{x_*}{2} + \frac{R}{2x_*} \\ &= \frac{\left(\frac{R}{x_*} + x_*\right)}{2} \end{aligned} \tag{9.11}$$

This is easily performed on an integer processor and involves only one division, one addition and a shift, which is very efficient.

When calculating an RMS value, we can calculate Eq. (9.11) once every sample time, and use the previous RMS value as the initial estimate. In many instances we don't need to iterate more than once since the previous RMS value will always be a good estimate of the current RMS value.

If we understand fixed-point techniques, we can optimize performance

C maths libraries provide square root routines, but when we understand their operation, we can optimise our code for performance.

EXAMPLE 9.3 Magnitude of a Complex Number

The following C function calculates the approximate magnitude of a complex number.

```
// Number of iterations to perform for square-root algorithm
const uint8_t NB_ITERATIONS = 5;

uint16_t Magnitude(int16_t real, int16_t imag)
{
  uint32_t magSquared;
  uint16_t mag;
  uint8_t i;

  magSquared = (uint32_t)((int32_t)real * (int32_t)real +
                          (int32_t)imag * (int32_t)imag);

  // Initial guess = magSquared / 2
  mag = (uint16_t)(magSquared / 2);

  // Estimate magnitude using Newton's method
  for (i = 0; i < NB_ITERATIONS; i++)
    mag = (uint16_t)((magSquared / mag + mag) / 2);

  return mag;
}
```

The function above will return an approximate result since the number of iterations is fixed. This may be acceptable in certain applications – otherwise the error between the square of the current root estimate and the original number to be squared can be used to terminate the iterations.

The function also contains two bugs:

1. The initial estimate of the magnitude may exceed the range of a `uint16_t`.
2. Division by zero is not tested for or handled. The ARM® Cortex®-M4 can produce an exception (usage fault) on division by zero, so we would need to write an exception handler.

Obviously a more robust function would need to handle these sources of potential error.

9.4.1 Number of Iterations

On an ARM® Cortex®-M, the unsigned integer division assembly instruction `UDIV` takes between 2 and 12 clock cycles to complete, with the number of cycles dependent on the input operands. It is therefore an "expensive" operation, timewise, and is something that we want to minimise.

Normally, in a mathematical operation involving iteration, we do not know beforehand the number of iterations to perform to achieve a desired level of accuracy. We need a way to stop iterating when the "error" reduces to a certain level (therefore saving redundant operations that will not increase the accuracy).

For the square root algorithm, we can perform the following analysis. The relative error in the square root "answer", x_*, is to be less than a certain value, ε, so we have the relation:

$$\left| \frac{x - x_*}{x_*} \right| < \varepsilon \tag{9.12}$$

Squaring both sides:

$$\frac{\left(x - x_*\right)^2}{x_*^2} < \varepsilon^2 \tag{9.13}$$

Since x_* is the square root, we have:

$$x_*^2 = R \tag{9.14}$$

so that:

$$\frac{x^2 - 2xx_* + R}{R} < \varepsilon^2 \tag{9.15}$$

Multiplying both sides by R, we get:

$$x^2 - 2xx_* + R < R\varepsilon^2 \tag{9.16}$$

If we make the approximation that our iterated value is close to the real answer, then:

$$xx_* \approx R \tag{9.17}$$

and substituting this into Eq. (9.17) we get:

$$x^2 - R < R\varepsilon^2 \tag{9.18}$$

To keep the mathematics integer based, suppose that the allowed relative error ε is expressed to the resolution of 1%:

$$\varepsilon = \frac{e}{100} \qquad e \text{ is an integer} \tag{9.19}$$

Then we have:

$$x^2 - R < R\frac{e^2}{10000} \tag{9.20}$$

Multiplying both sides by 10000 gives a relationship that uses integers only that we can use in C code to halt the iterative process:

```
do
  {
    // Do an iteration to find a new sqrt value
    ...
    // Find error
    error = sqrt * sqrt - sqr;
    // Take absolute error
    if (error < 0)
      error = -error;
  } while ((error * 10000) > (sqr * tolerance * tolerance));
```

Listing 9.1 – Detecting when to stop iterating the `sqrt` algorithm

10 Real-Time Operating Systems

Contents

10.2

Introduction

A real-time operating system (RTOS) for an embedded system simplifies the design of real-time software by allowing the application to be divided into multiple threads managed by the RTOS. The *kernel* of an embedded RTOS needs to support multithreading, pre-emption, and thread priority. The RTOS will also provide services to threads for communication, synchronization and coordination. A RTOS is to be used for a "hard" real-time system – i.e. threads have to be performed not only correctly but also in a timely fashion.

Operating systems for larger computers (such as the PC) are non-real-time operating systems and usually provide a much larger range of application services, such as memory management and file management which normally do not apply to embedded systems.

10.1 Real-Time Kernel Concepts

The following sections describe real-time kernel concepts.

10.1.1 Threads

A thread is a simple program that thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into threads which are responsible for a portion of the problem. Each thread is assigned a priority, its own set of CPU registers and its own stack area.

Each thread is typically an infinite loop that can be in one of four states: `READY`, `RUNNING`, `WAITING` or `INTERRUPTED`.

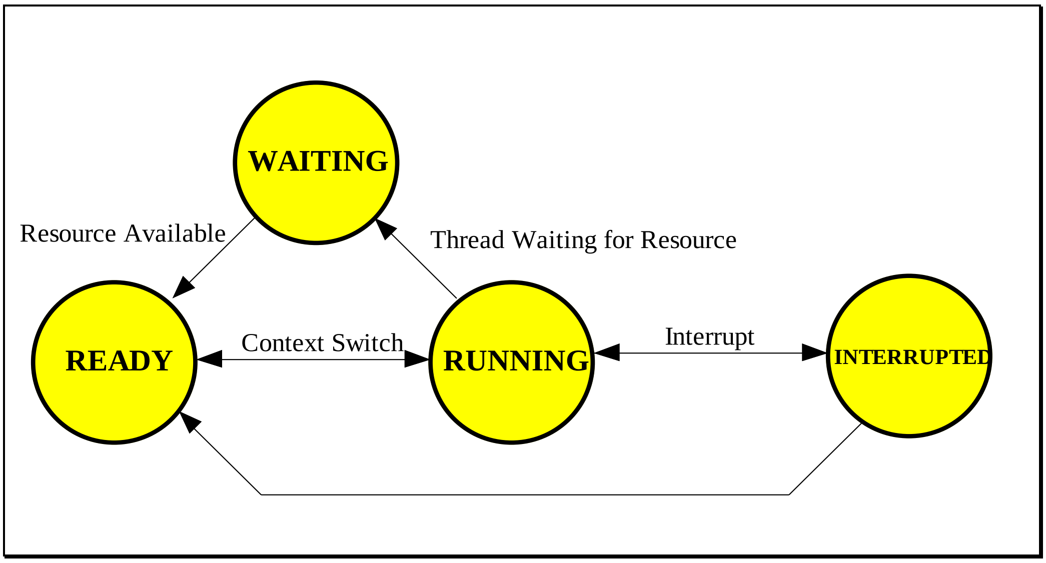


Figure 10.1 – Thread states

A thread is `READY` when it can execute but its priority is less than the current running thread. A thread is `RUNNING` when it has control of the CPU. A thread is `WAITING` when the thread suspends itself until a certain amount of time has elapsed, or when it requires the occurrence of an event: waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur etc. Finally, a thread is `INTERRUPTED` when an interrupt occurred and the CPU is in the process of servicing the interrupt.

10.1.2 Context Switch

When the multithreading kernel decides to run a different thread, it simply saves the current thread's context (CPU registers) in the current thread's context storage area (the *thread control block*, or TCB). Once this operation is performed, the new thread's context is restored from its TCB and the CPU resumes execution of the new thread's code. This process is called a context switch. Context switching adds overhead to the application.

10.1.3 Kernel

The kernel is the part of an OS that is responsible for the management of threads (i.e., managing the CPU's time) and for communication between threads. The fundamental service provided by the kernel is context switching.

10.1.4 Scheduler

The scheduler is the part of the kernel responsible for determining which thread will run next. Most real-time kernels are priority based. Each thread is assigned a priority based on its importance. Establishing the priority for each thread is application specific. In a priority-based kernel, control of the CPU will always be given to the highest priority thread ready to run. In a preemptive kernel, when a thread makes a higher priority thread ready to run, the current thread is preempted (suspended) and the higher priority thread is immediately given control of the CPU. If an interrupt service routine (ISR) makes a higher priority thread ready, then when the ISR is completed the interrupted thread is suspended and the new higher priority thread is resumed.

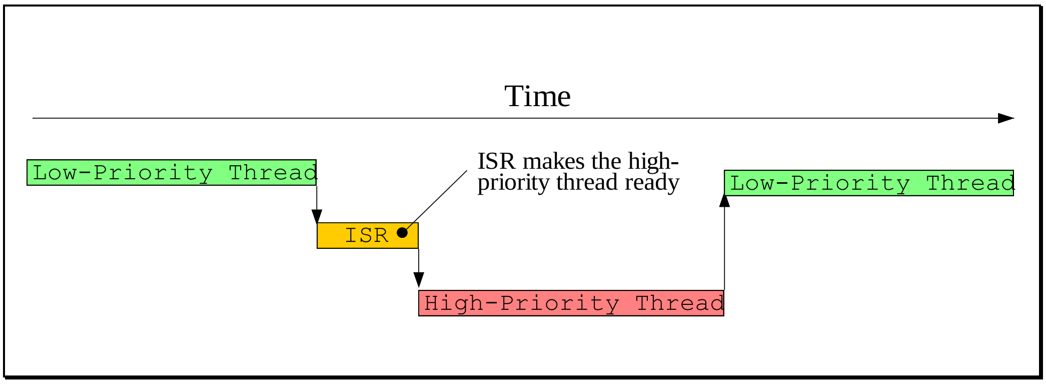


Figure 10.2 – Preemptive kernel

With a preemptive kernel, execution of the highest priority thread is deterministic; you can determine *when* the highest priority thread will get control of the CPU.

Application code using a preemptive kernel should not use non-reentrant functions, unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores, because both a low- and a high-priority thread can use a common function. Corruption of data may occur if the higher priority thread preempts a lower priority thread that is using the function.

To summarize, a preemptive kernel always executes the highest priority thread that is ready to run. An interrupt preempts a thread. Upon completion of an ISR, the kernel resumes execution to the highest priority thread ready to run (not the interrupted thread). Thread-level response is optimum and deterministic.

10.2 Reentrancy

A reentrant function can be used by more than one thread without fear of data corruption. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables (i.e., CPU registers or variables on the stack) or protect data when global variables are used. An example of a reentrant function is shown below:

```
char* strcpy(char* dst, const char* src)
{
  char* ptr = dst;
  while (*dst++ = *src++);
  return ptr;
}
```

Since copies of the arguments to **strcpy**() are placed on the thread's stack, and the local variable is created on the thread’s stack, **strcpy**() can be invoked by multiple threads without fear that the threads will corrupt each other's pointers.

An example of a non-reentrant function is shown below:

```
static int Temp;

void swap(int* x, int* y)
{
  Temp = *x;
  *x   = *y;
  *y   = Temp;
}
```

swap() is a simple function that swaps the contents of its two arguments. Since `Temp` is a global variable, if the **swap**() function gets preempted after the first line by a higher priority thread which also uses the **swap**() function, then when the low priority thread resumes it will use the `Temp` value that was used by the high priority thread.

You can make **swap**() reentrant with one of the following techniques:

- Declare `Temp` local to **swap**().
- Disable interrupts before the operation and enable them afterwards.
- Use a semaphore.

10.3 Thread Priority

A priority is assigned to each thread. The more important the thread, the higher the priority given to it.

10.3.1 Static Priorities

Thread priorities are said to be static when the priority of each thread does not change during the application's execution. Each thread is thus given a fixed priority at compile time. All the threads and their timing constraints are known at compile time in a system where priorities are static.

10.3.2 Dynamic Priorities

Thread priorities are said to be dynamic if the priority of threads can be changed during the application's execution; each thread can change its priority at run time. This is a desirable feature to have in a real-time kernel to avoid priority inversions.

10.3.3 Priority Inversions

Priority inversion is a problem in real-time systems and occurs mostly when you use a real-time kernel. Priority inversion is any situation in which a low priority thread holds a resource while a higher priority thread is ready to use it. In this situation the low priority thread prevents the high priority thread from executing until it releases the resource.

To avoid priority inversion a multithreading kernel should change the priority of a thread automatically to help prevent priority inversions. This is called priority inheritance.

10.4 Mutual Exclusion

The easiest way for threads to communicate with each other is through shared data structures. This is especially easy when all threads exist in a single address space and can reference global variables, pointers, buffers, linked lists, FIFOs, etc. Although sharing data simplifies the exchange of information, you must ensure that each thread has exclusive access to the data to avoid contention and data corruption. The most common methods of obtaining exclusive access to shared resources are:

- disabling interrupts,
- performing test-and-set operations,
- disabling scheduling, and
- using semaphores.

10.4.1 Disabling and Enabling Interrupts

The easiest and fastest way to gain exclusive access to a shared resource is by disabling and enabling interrupts, as shown in the pseudocode:

```
Disable interrupts;
Access the resource (read/write from/to variables);
Reenable interrupts;
```

Kernels use this technique to access internal variables and data structures. In fact, kernels usually provide two functions that allow you to disable and then enable interrupts from your C code: **OS_EnterCritical**() and **OS_ExitCritical**(), respectively. You need to use these functions in tandem, as shown below:

```c
void Function(void)
{
  OS_EnterCritical();
  .
  .     /* You can access shared data in here */
  .
  OS_ExitCritical();
}
```

You must be careful, however, not to disable interrupts for too long because this affects the response of your system to interrupts. This is known as interrupt latency. You should consider this method when you are changing or copying a few variables. Also, this is the only way that a thread can share variables or data structures with an ISR. In all cases, you should keep interrupts disabled for as little time as possible.

If you use a kernel, you are basically allowed to disable interrupts for as much time as the kernel does without affecting interrupt latency. Obviously, you need to know how long the kernel will disable interrupts.

10.4.2 Semaphores

The semaphore was invented by Edgser Dijkstra in the mid-1960s. It is a protocol mechanism offered by most multithreading kernels. Semaphores are used to:

- control access to a shared resource (mutual exclusion),
- signal the occurrence of an event, and
- allow two threads to synchronize their activities.

A semaphore is a key that your code acquires in order to continue execution. If the semaphore is already in use, the requesting thread is suspended until the semaphore is released by its current owner. In other words, the requesting thread says: "Give me the key. If someone else is using it, I am willing to wait for it!" There are two types of semaphores: binary semaphores and counting semaphores. As its name implies, a binary semaphore can only take two values: 0 or 1. A counting semaphore allows values between 0 and 255, 65535, or 4294967295, depending on whether the semaphore mechanism is implemented using 8, 16, or 32 bits, respectively. The actual size depends on the kernel used. Along with the semaphore's value, the kernel also needs to keep track of threads waiting for the semaphore's availability.

Generally, only three operations can be performed on a semaphore: **`Create`**`()`, **`Wait`**`()`, and **`Signal`**`()`. The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of threads is always initially empty.

A thread desiring the semaphore will perform a **`Wait`**`()` operation. If the semaphore is available (the semaphore value is greater than 0), the semaphore value is decremented and the thread continues execution. If the semaphore's value is 0, the thread performing a **`Wait`**`()` on the semaphore is placed in a waiting list. Most kernels allow you to specify a timeout; if the semaphore is not available within a certain amount of time, the requesting thread is made ready to run and an error code (indicating that a timeout has occurred) is returned to the caller.

A thread releases a semaphore by performing a **`Signal`**`()` operation. If no thread is waiting for the semaphore, the semaphore value is simply incremented. If any thread is waiting for the semaphore, however, one of the threads is made ready to run and the semaphore value is not incremented; the key is given to one of the threads waiting for it. Depending on the kernel, the thread that receives the semaphore is either:

- the highest priority thread waiting for the semaphore, or
- the first thread that requested the semaphore (First In First Out).

Some kernels have an option that allows you to choose either method when the semaphore is initialized. For the first option, if the readied thread has a higher priority than the current thread (the thread releasing the semaphore), a context switch occurs (with a preemptive kernel) and the higher priority thread resumes execution; the current thread is suspended until it again becomes the highest priority thread ready to run.

Listing 10.1 shows how you can share data using a semaphore. Any thread needing access to the same shared data calls **`OS_SemaphoreWait`**`()`, and when the thread is done with the data, the thread calls **`OS_SemaphoreSignal`**`()`. Both of these functions are described later. You should note that a semaphore is an object that needs to be initialized before it is used; for mutual exclusion, a semaphore is initialized to a value of 1. Using a semaphore to access shared data doesn't affect interrupt latency. If an ISR or the current thread makes a higher priority thread ready to run while accessing shared data, the higher priority thread executes immediately.

```
OS_ECB* SharedDataSemaphore;
```

```
void Function(void)
{
  OS_ERROR error;

  error = OS_SemaphoreWait(SharedDataSemaphore, 0);
  .
  .    // You can access shared data in here
  .    // (interrupts are recognized)
  .
  error = OS_SemaphoreSignal(SharedDataSemaphore);
}
```

Listing 10.1 – Accessing shared data by obtaining a semaphore

Semaphores are especially useful when threads share I/O devices. Imagine what would happen if two threads were allowed to send characters to a printer at the same time. The printer would contain interleaved data from each thread. For instance, the printout from Thread 1 printing "`I am Thread 1!`" and Thread 2 printing "`I am Thread 2!`" could result in:

“`I Ia amm T Threahread d1 !2!`”

In this case, use a semaphore and initialize it to 1 (i.e., a binary semaphore). The rule is simple: to access the printer each thread first must obtain the resource's semaphore.

Figure 10.3 shows threads competing for a semaphore to gain exclusive access to the printer. Note that the semaphore is represented symbolically by a key, indicating that each thread must obtain this key to use the printer.

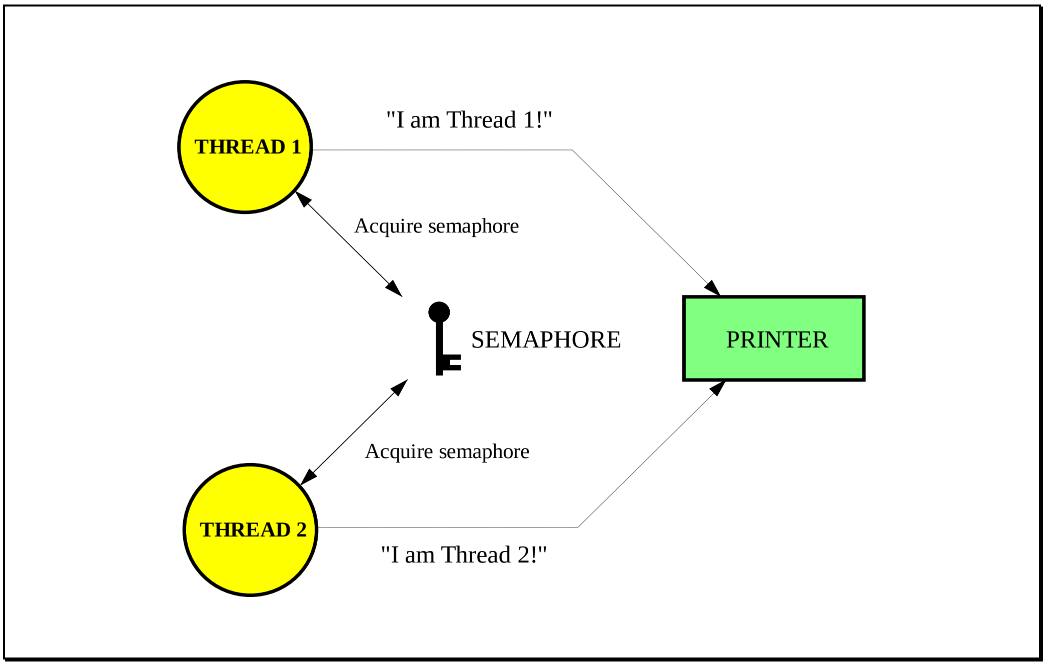


Figure 10.3 – Using a semaphore to get permission to access a printer

The above example implies that each thread must know about the existence of the semaphore in order to access the resource. There are situations when it is better to encapsulate the semaphore. Each thread would thus not know that it is actually acquiring a semaphore when accessing the resource. For example, the UART port may be used by multiple threads to send commands and receive responses from a PC:

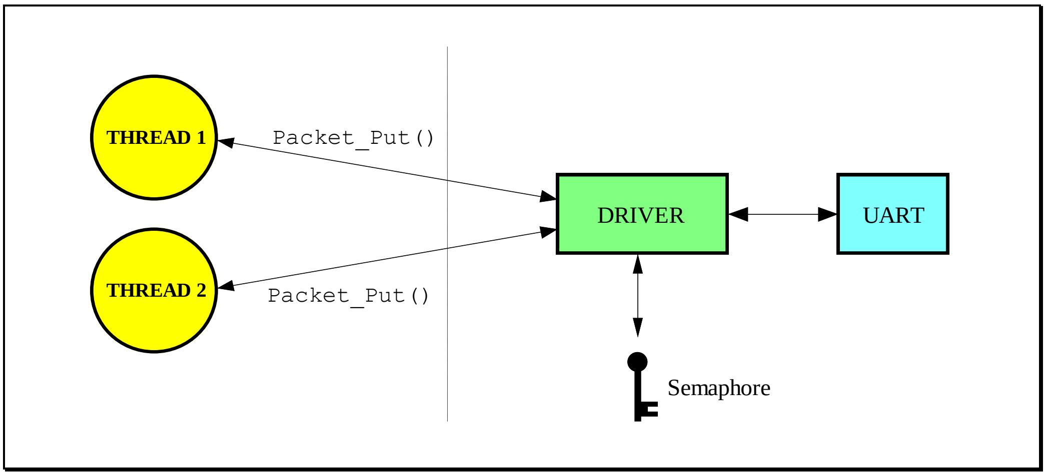


Figure 10.4 – Hiding a semaphore from threads

The function **`Packet_Put()`** is called with two arguments: the packet and a timeout in case the device doesn't respond within a certain amount of time. The pseudocode for this function is shown in Listing 10.2.

```
uint8_t Packet_Put(TPacket* packet, const uint16_t timeout)
{
  Acquire serial port's semaphore;
  Send packet to device;
  Wait for response (with timeout);
  Release semaphore;
  if (timed out)
    return (error code);
  else
    return (no error);
}
```

Listing 10.2 – Encapsulating a semaphore

Each thread that needs to send a packet to the serial port has to call this function. The semaphore is assumed to be initialized to 1 (i.e., available) by the communication driver initialization routine. The first thread that calls **`Packet_Put()`** acquires the semaphore, proceeds to send the packet, and waits for a response. If another thread attempts to send a command while the port is busy, this second thread is suspended until the semaphore is released. The second thread appears simply to have made a call to a normal function that will not return until the function has performed its duty. When the semaphore is released by the first thread, the second thread acquires the semaphore and is allowed to use the serial port.

A *counting semaphore* is used when a resource can be used by more than one thread at the same time. For example, a counting semaphore is used in the management of a buffer pool as shown in Figure 10.5.

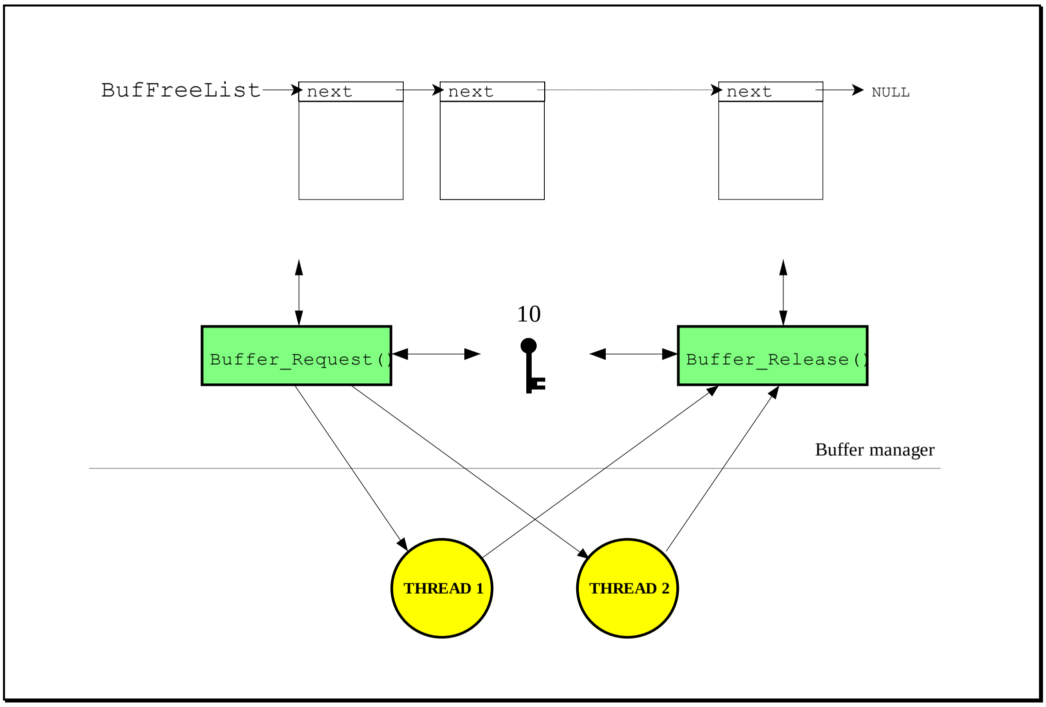


Figure 10.5 – Using a counting semaphore

Assume that the buffer pool initially contains 10 buffers. A thread would obtain a buffer from the buffer manager by calling **`Buffer_Request`**`()`. When the buffer is no longer needed, the thread would return the buffer to the buffer manager by calling **`Buffer_Release`**`()`. The pseudocode for these functions is shown in Listing 10.3.

```
BUF* Buffer_Request(void)
{
  BUF* ptr;

  Acquire a semaphore;
  Disable interrupts;
  ptr         = BufFreeList;
  BufFreeList = ptr->next;
  Enable interrupts;
  return (ptr);
}
```

```
void Buffer_Release(BUF* ptr)
{
  Disable interrupts;
  ptr->next = BufFreeList;
  BufFreeList = ptr;
  Enable interrupts;
  Release semaphore;
}
```

Listing 10.3 – Buffer management using a semaphore

The buffer manager will satisfy the first 10 buffer requests because there are 10 keys. When all semaphores are used, a thread requesting a buffer is suspended until a semaphore becomes available. Interrupts are disabled to gain exclusive access to the linked list (this operation is very quick). When a thread is finished with the buffer it acquired, it calls `Buffer_Release()` to return the buffer to the buffer manager; the buffer is inserted into the linked list before the semaphore is released. By encapsulating the interface to the buffer manager in `Buffer_Request()` and `Buffer_Release()`, the caller doesn't need to be concerned with the actual implementation details.

Semaphores are often overused. The use of a semaphore to access a simple shared variable is overkill in most situations. The overhead involved in acquiring and releasing the semaphore can consume valuable time. You can do the job just as efficiently by disabling and enabling interrupts. Suppose that two threads are sharing a 32-bit integer variable. The first thread increments the variable while the other thread clears it. If you consider how long a processor takes to perform either operation, you will realize that you do not need a semaphore to gain exclusive access to the variable. Each thread simply needs to disable interrupts before performing its operation on the variable and enable interrupts when the operation is complete. A semaphore should be used, however, if the variable is a floating-point variable and the microprocessor doesn't support floating point in hardware. In this case, the processing time involved in processing the floating-point variable could have affected interrupt latency if you had disabled interrupts.

10.4.3 Deadlock (or Deadly Embrace)

A *deadlock*, also called a *deadly embrace*, is a situation in which two threads are each unknowingly waiting for resources held by the other. Assume thread `T1` has exclusive access to resource `R1` and thread `T2` has exclusive access to resource `R2`. If `T1` needs exclusive access to `R2` and `T2` needs exclusive access to `R1`, neither thread can continue. They are deadlocked. The simplest way to avoid a deadlock is for threads to:

- acquire all resources before proceeding,
- acquire the resources in the same order, and
- release the resources in the reverse order

Most kernels allow you to specify a timeout when acquiring a semaphore. This feature allows a deadlock to be broken. If the semaphore is not available within a certain amount of time, the thread requesting the resource resumes execution. Some form of error code must be returned to the thread to notify it that a timeout occurred. A return error code prevents the thread from thinking it has obtained the resource. Deadlocks generally occur in large multithreading systems, not in embedded systems.

10.5 Synchronization

A thread can be synchronized with an ISR (or another thread when no data is being exchanged) by using a semaphore as shown in Figure 10.6.

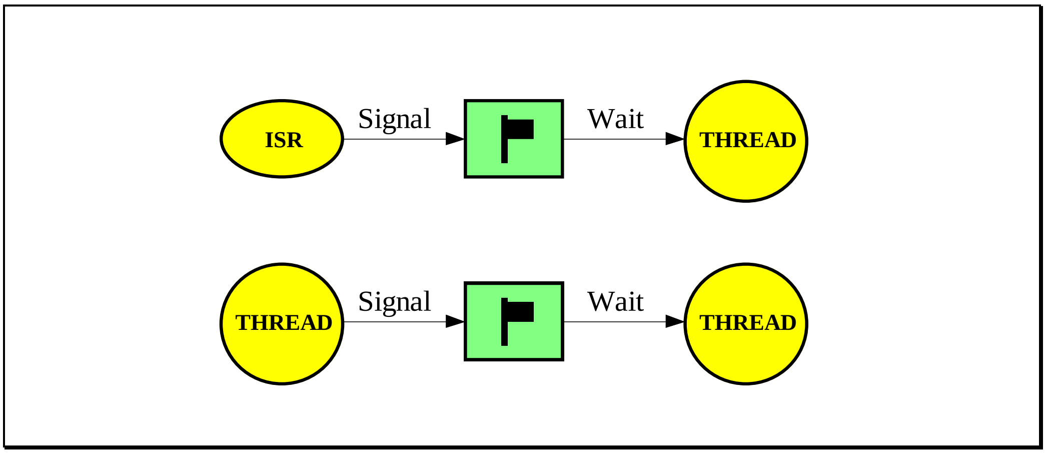


Figure 10.6 – Synchronizing threads and ISRs

Note that, in this case, the semaphore is drawn as a flag to indicate that it is used to signal the occurrence of an event (rather than to ensure mutual exclusion, in which case it would be drawn as a key). When used as a synchronization mechanism, the semaphore is initialized to 0. Using a semaphore for this type of synchronization is called a *unilateral rendezvous*. A thread initiates an I/O operation and waits for the semaphore. When the I/O operation is complete, an ISR (or another thread) signals the semaphore and the thread is resumed.

If the kernel supports counting semaphores, the semaphore would accumulate events that have not yet been processed. Note that more than one thread can be waiting for an event to occur. In this case, the kernel could signal the occurrence of the event either to:

- the highest priority thread waiting for the event to occur or
- the first thread waiting for the event.

Depending on the application, more than one ISR or thread could signal the occurrence of the event.

Two threads can synchronize their activities by using two semaphores, as shown in Figure 10.7. This is called a *bilateral rendezvous*. A bilateral rendezvous is similar to a unilateral rendezvous, except both threads must synchronize with one another before proceeding.

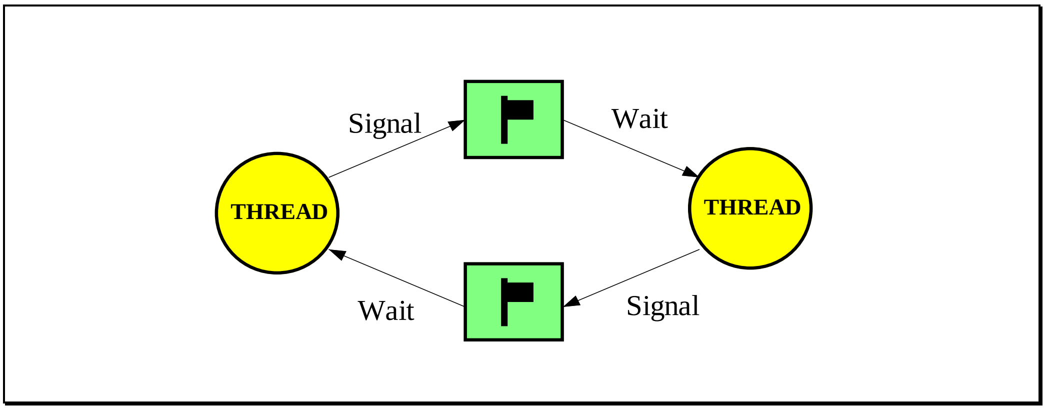


Figure 10.7 – Threads synchronizing their activities

For example, two threads are executing as shown in Listing 10.4. When the first thread reaches a certain point, it signals the second thread (1) then waits for a return signal (2). Similarly, when the second thread reaches a certain point, it signals the first thread (3) and waits for a return signal (4). At this point, both threads are synchronized with each other. A bilateral rendezvous cannot be performed between a thread and an ISR because an ISR cannot wait on a semaphore.

```
void Thread1(void)
{
  for (;;)
  {
    Perform operation 1;
    Signal thread #2;               (1)
    Wait for signal from thread #2; (2)
    Continue operation 1;
  }
}

void Thread2(void)
{
  for (;;)
  {
    Perform operation 2;
    Signal thread #1;               (3)
    Wait for signal from thread #1; (4)
    Continue operation 2;
  }
}
```

Listing 10.4 – Bilateral rendezvous

10.6 Interthread Communication

It is sometimes necessary for a thread or an ISR to communicate information to another thread. This information transfer is called interthread communication. Information may be communicated between threads in two ways: through global data or by sending messages.

When using global variables, each thread or ISR must ensure that it has exclusive access to the variables. If an ISR is involved, the only way to ensure exclusive access to the common variables is to disable interrupts. If two threads are sharing data, each can gain exclusive access to the variables either by disabling and enabling interrupts or with the use of a semaphore (as we have seen). Note that a thread can only communicate information to an ISR by using global variables. A thread is not aware when a global variable is changed by an ISR, unless the ISR signals the thread by using a semaphore or unless the thread polls the contents of the variable periodically. To correct this situation, you should consider using either a message mailbox or a message queue.

10.6.1 Message Mailboxes

Messages can be sent to a thread through kernel services. A Message Mailbox, also called a message exchange, is typically a pointer-size variable. Through a service provided by the kernel, a thread or an ISR can deposit a message (the pointer) into this mailbox. Similarly, one or more threads can receive messages through a service provided by the kernel. Both the sending thread and receiving thread agree on what the pointer is actually pointing to.

A waiting list is associated with each mailbox in case more than one thread wants to receive messages through the mailbox. A thread desiring a message from an empty mailbox is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the thread waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting thread is made ready to run and an error code (indicating that a timeout has occurred) is returned to it. When a message is deposited into the mailbox, either the highest priority thread waiting for the message is given the message (priority based) or the first thread to request a message is given the message (First-In-First-Out, or FIFO). Figure 10.8 shows a thread depositing a message into a mailbox. Note that the mailbox is represented by an I-beam and the timeout is represented by an hourglass. The number next to the hourglass represents the number of clock ticks the thread will wait for a message to arrive.

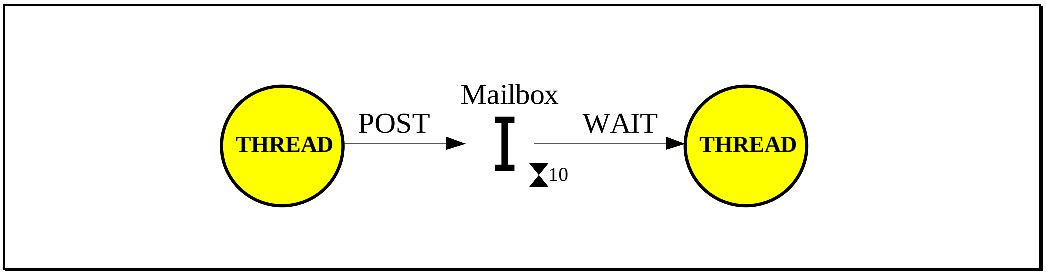


Figure 10.8 – Message mailbox

Kernels typically provide the following mailbox services:

- Initialize the contents of a mailbox. The mailbox initially may or may not contain a message.
- Deposit a message into the mailbox (POST).
- Wait for a message to be deposited into the mailbox (WAIT).
- Get a message from a mailbox if one is present, but do not suspend the caller if the mailbox is empty (ACCEPT). If the mailbox contains a message, the message is extracted from the mailbox. A return code is used to notify the caller about the outcome of the call.

Message mailboxes can also simulate binary semaphores. A message in the mailbox indicates that the resource is available, and an empty mailbox indicates that the resource is already in use by another thread.

10.6.2 Message Queues

A message queue is used to send one or more messages to a thread. A message queue is basically an array of mailboxes. Through a service provided by the kernel, a thread or an ISR can deposit a message (the pointer) into a message queue. Similarly, one or more threads can receive messages through a service provided by the kernel. Both the sending thread and receiving thread agree as to what the pointer is actually pointing to. Generally, the first message inserted in the queue will be the first message extracted from the queue (FIFO).

As with the mailbox, a waiting list is associated with each message queue, in case more than one thread is to receive messages through the queue. A thread desiring a message from an empty queue is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the thread waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting thread is made ready to run and an error code (indicating a timeout has occurred) is returned to it. When a message is deposited into the queue, either the highest priority thread or the first thread to wait for the message is given the message. Figure 10.9 shows an ISR (Interrupt Service Routine) depositing a message into a queue. Note that the queue is represented graphically by a double I-beam. The "10" indicates the number of messages that can accumulate in the queue. A "0" next to the hourglass indicates that the thread will wait forever for a message to arrive.

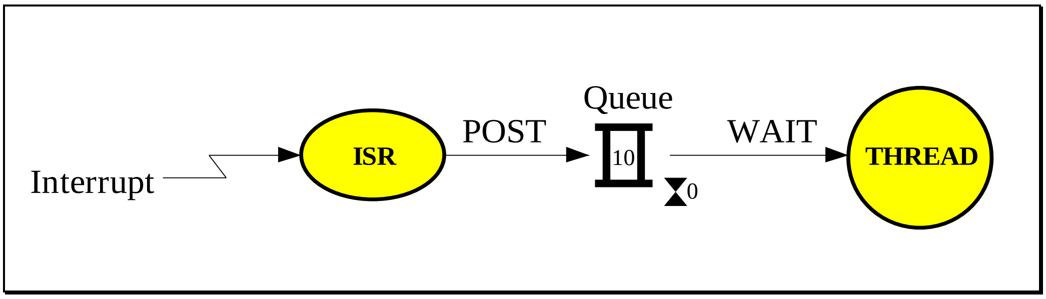


Figure 10.9 – Message queue

Kernels typically provide the message queue services listed below.

- Initialize the queue. The queue is always assumed to be empty after initialization.
- Deposit a message into the queue (POST).
- Wait for a message to be deposited into the queue (WAIT).
- Get a message from a queue if one is present, but do not suspend the caller if the queue is empty (ACCEPT). If the queue contains a message, the message is extracted from the queue. A return code is used to notify the caller about the outcome of the call.

10.7 Interrupts

An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves all of its context (i.e., registers) and jumps to a special subroutine called an Interrupt Service Routine, or ISR. The ISR processes the event, and upon completion of the ISR, the program returns to:

- the background for a foreground / background system,
- the interrupted thread for a non-preemptive kernel, or
- the highest priority thread ready to run for a preemptive kernel.

Interrupts allow a microprocessor to process events when they occur. This prevents the microprocessor from continuously polling an event to see if it has occurred. Microprocessors allow interrupts to be ignored and recognized through the use of two special instructions: disable interrupts and enable interrupts, respectively. In a real-time environment, interrupts should be disabled as little as possible. Disabling interrupts affects interrupt latency and may cause interrupts to be missed. Processors generally allow interrupts to be nested. This means that while servicing an interrupt, the processor will recognize and service other (more important) interrupts, as shown in Figure 10.10.

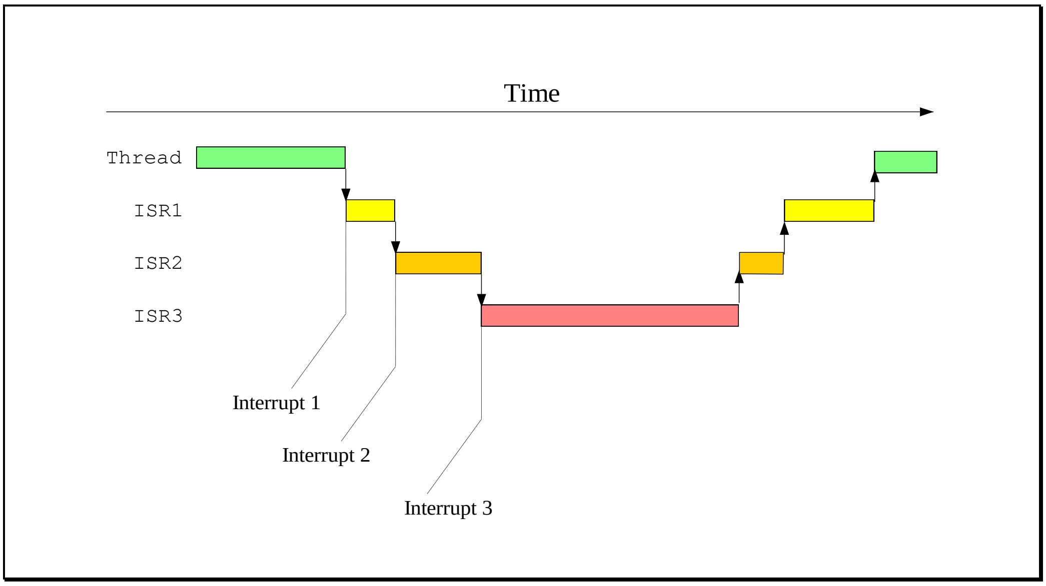


Figure 10.10 – Interrupt nesting

10.7.1 Interrupt Latency

Probably the most important specification of a real-time kernel is the amount of time interrupts are disabled. All real-time systems disable interrupts to manipulate critical sections of code and reenable interrupts when the critical section has executed. The longer interrupts are disabled, the higher the interrupt latency. Interrupt latency is given by Eq. (10.1).

Interrupt latency

= Maximum amount of time interrupts are disabled (10.1)

+ Time to start executing the first instruction in the ISR

10.7.2 Interrupt Response

Interrupt response is defined as the time between the reception of the interrupt and the start of the user code that handles the interrupt. The interrupt response time accounts for all the overhead involved in handling an interrupt.

For a foreground / background system, the user ISR code is executed immediately. The response time is given by Eq. (10.2).

Interrupt response time

= Interrupt latency (10.2)

For a preemptive kernel, a special function provided by the kernel needs to be called. This function notifies the kernel that an ISR is in progress and allows the kernel to keep track of interrupt nesting. This function is called **`OS_ISREnter()`**. The response time to an interrupt for a preemptive kernel is given by Eq. (10.3).

Interrupt response time

= Interrupt latency (10.3)

+ Execution time of the kernel ISR entry function

A system's worst case interrupt response time is its only response time. Your system may respond to interrupts in 50ms 99 percent of the time, but if it responds to interrupts in 250ms the other 1 percent, you must assume a 250ms interrupt response time.

10.7.3 Interrupt Recovery

Interrupt recovery is defined as the time required for the processor to return to the interrupted code. Interrupt recovery in a foreground / background system simply involves restoring the processor's context and returning to the interrupted thread. Interrupt recovery is given by Eq. (10.4).

$$\begin{aligned}&\text{Interrupt recovery time}\\&= \text{Time to execute the return from interrupt instruction}\end{aligned} \tag{10.4}$$

For a preemptive kernel, interrupt recovery is more complex. Typically, a function provided by the kernel is called at the end of the ISR. This function is called **`OS_ISRExit()`** and allows the kernel to determine if all interrupts have nested. If they have nested (i.e., a return from interrupt would return to thread-level code), the kernel determines if a higher priority thread has been made ready to run as a result of the ISR. If a higher priority thread is ready to run as a result of the ISR, this thread is resumed. Note that, in this case, the interrupted thread will resume only when it again becomes the highest priority thread ready to run. For a preemptive kernel, interrupt recovery is given by Eq. (10.5).

$$\begin{aligned}&\text{Interrupt recovery time}\\&= \text{Time to determine if a higher priority thread is ready}\\&+ \text{Time to restore the CPU's context of the highest priority thread}\\&+ \text{Time to execute the return from interrupt instruction}\end{aligned} \tag{10.5}$$

10.7.4 Interrupt Latency, Response, and Recovery

Figure 10.11 and Figure 10.12 show the interrupt latency, response, and recovery for a foreground / background system and a preemptive kernel, respectively.

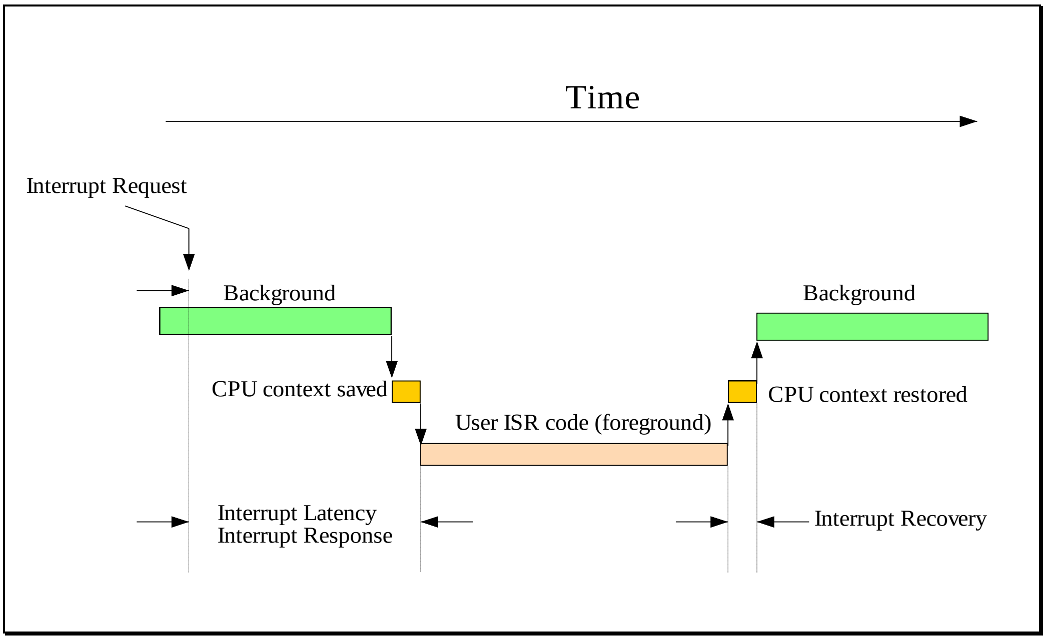


Figure 10.11 – Interrupt latency, response, and recovery (foreground / background)

You should note that for a preemptive kernel, the exit function either decides to return to the interrupted thread (A) or to a higher priority thread that the ISR has made ready to run (B). In the latter case, the execution time is slightly longer because the kernel has to perform a context switch.

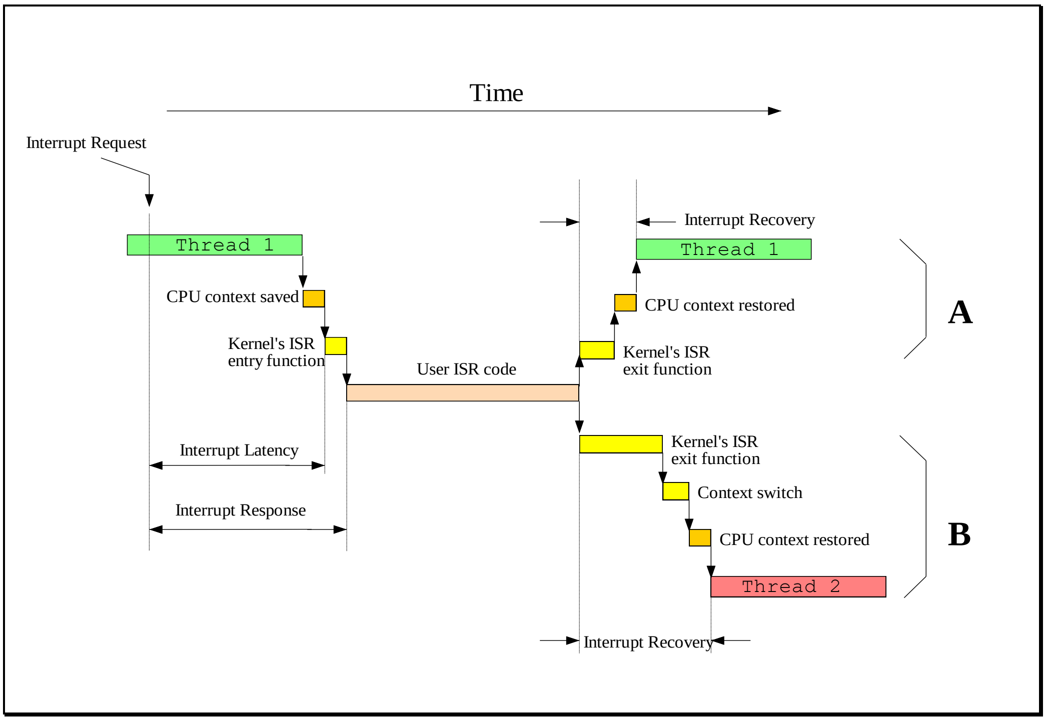


Figure 10.12 – Interrupt latency, response, and recovery (preemptive kernel)

10.7.5 ISR Processing Time

Although ISRs should be as short as possible, there are no absolute limits on the amount of time for an ISR. One cannot say that an ISR must always be less than 100 ms, 500 ms, or l ms. If the ISR code is the most important code that needs to run at any given time, it could be as long as it needs to be. In most cases, however, the ISR should recognize the interrupt, obtain data or a status from the interrupting device, and signal a thread to perform the actual processing. You should also consider whether the overhead involved in signalling a thread is more than the processing of the interrupt. Signalling a thread from an ISR (i.e., through a semaphore, a mailbox, or a queue) requires some processing time. If processing your interrupt requires less than the time required to signal a thread, you should consider processing the interrupt in the ISR itself and allowing higher priority interrupts to be recognized and serviced.

10.7.6 Clock Tick

A clock tick is a special interrupt that occurs periodically. This interrupt can be viewed as the system's heartbeat. The time between interrupts is application specific and is generally between 1 and 200 ms. The clock tick interrupt allows a kernel to delay threads for an integral number of clock ticks and to provide timeouts when threads are waiting for events to occur. The faster the tick rate, the higher the overhead imposed on the system.

All kernels allow threads to be delayed for a certain number of clock ticks. The resolution of delayed threads is one clock tick; however, this does not mean that its accuracy is one clock tick.

Figure 10.13 through Figure 10.15 are timing diagrams showing a thread delaying itself for one clock tick. The shaded areas indicate the execution time for each operation being performed. Note that the time for each operation varies to reflect typical processing, which would include loops and conditional statements (i.e., `if`/`else`, `switch`, and `?:`). The processing time of the Tick ISR has been exaggerated to show that it too is subject to varying execution times.

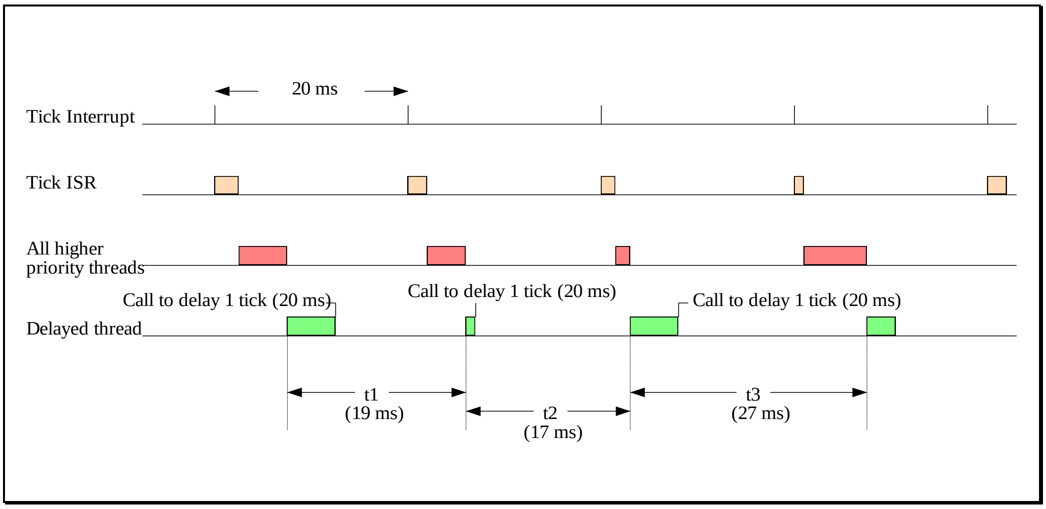


Figure 10.13 – Delaying a thread for one tick (Case 1)

Case 1 (Figure 10.13) shows a situation where higher priority threads and ISRs execute prior to the thread, which needs to delay for one tick. The thread attempts to delay for 20ms but because of its priority, it actually executes at varying intervals. This causes the execution of the thread to jitter.

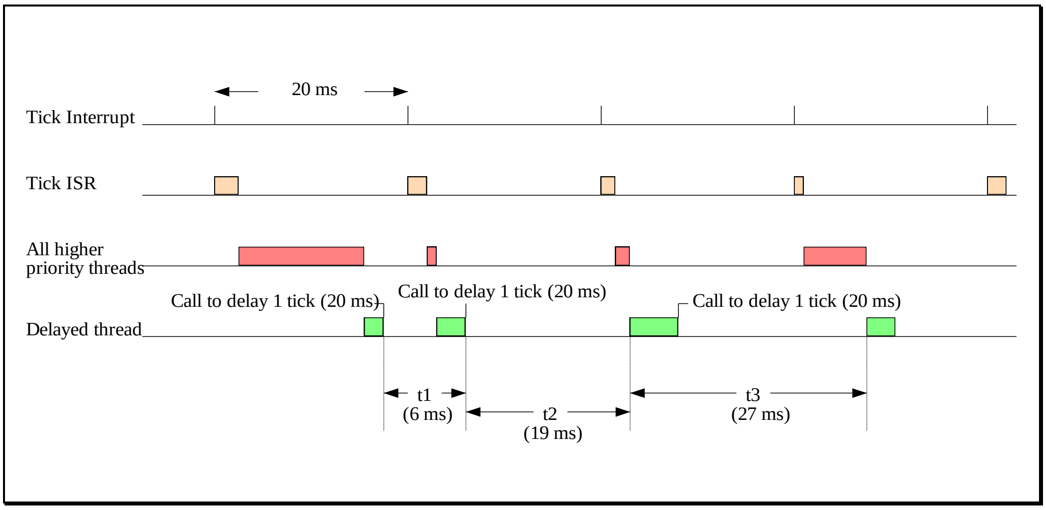


Figure 10.14 – Delaying a thread for one tick (Case 2)

Case 2 (Figure 10.14) shows a situation where the execution times of all higher priority threads and ISRs are slightly less than one tick. If the thread delays itself just before a clock tick, the thread will execute again almost immediately! Because of this, if you need to delay a thread at least one clock tick, you must specify one extra tick. In other words, if you need to delay a thread for *at least* five ticks, you must specify six ticks!

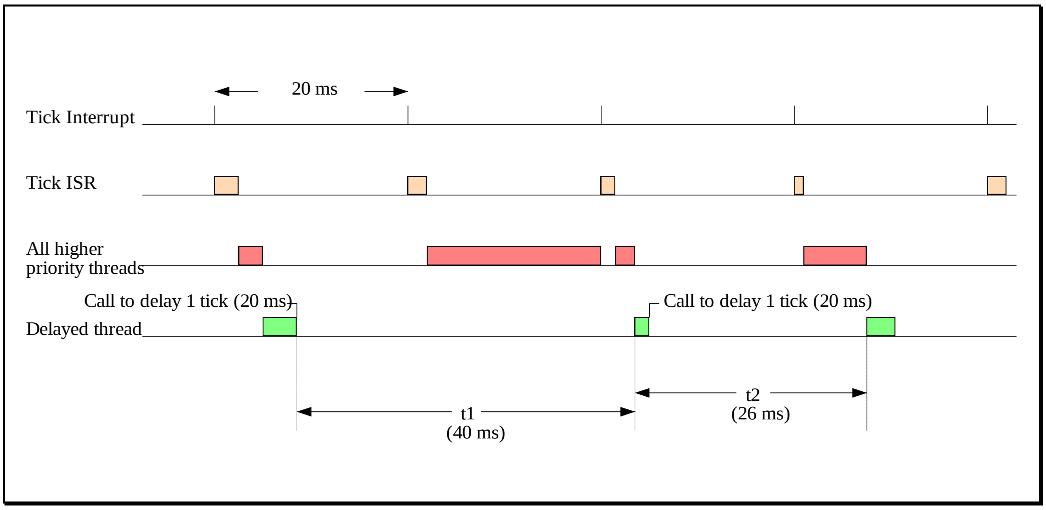


Figure 10.15 – Delaying a thread for one tick (Case 3)

Case 3 (Figure 10.15) shows a situation in which the execution times of all higher priority threads and ISRs extend beyond one clock tick. In this case, the thread that tries to delay for one tick actually executes two ticks later and misses its deadline. This might be acceptable in some applications, but in most cases it isn't.

These situations exist with all real-time kernels. They are related to CPU processing load and possibly incorrect system design. Here are some possible solutions to these problems:

- Increase the clock rate of your microprocessor.
- Increase the time between tick interrupts.
- Rearrange thread priorities.
- Avoid using floating-point maths (if you must, use single precision).
- Get a compiler that performs better code optimization.
- Write time-critical code in assembly language.
- If possible, upgrade to a faster microprocessor in the same family; that is, Cortex®-M0+ to Cortex®-M3, etc.

Regardless of what you do, jitter will always occur.

10.8 Memory Requirements

If you are designing a foreground / background system, the amount of memory required depends solely on your application code. With a multithreading kernel, things are quite different. To begin with, a kernel requires extra code space (Flash). The size of the kernel depends on many factors. Depending on the features provided by the kernel, you can expect anywhere from 1 to 100 KiB. A minimal kernel for a 32-bit CPU that provides only scheduling, context switching, semaphore management, delays, and timeouts should require about 1 to 3 KiB of code space.

Because each thread runs independently of the others, it must be provided with its own stack area (RAM). As a designer, you must determine the stack requirement of each thread as closely as possible (this is sometimes a difficult undertaking). The stack size must not only account for the thread requirements (local variables, function calls, etc.), it must also account for maximum interrupt nesting (saved registers, local storage in ISRs, etc.). Depending on the target processor and the kernel used, a separate stack can be used to handle all interrupt-level code. This is a desirable feature because the stack requirement for each thread can be substantially reduced. Another desirable feature is the ability to specify the stack size of each thread on an individual basis. Conversely, some kernels require that all thread stacks be the same size. All kernels require extra RAM to maintain internal variables, data structures, queues, etc. The total RAM required if the kernel does not support a separate interrupt stack is given by Eq. (10.6).

$$\begin{aligned}&\text{Total RAM requirements}\\&= \text{Application code requirements}\\&+ \text{Data space (i.e., RAM) needed by the kernel}\\&+ \text{SUM(thread stacks + MAX(ISR nesting))}\end{aligned} \tag{10.6}$$

Unless you have large amounts of RAM to work with, you need to be careful how you use the stack space. To reduce the amount of RAM needed in an application, you must be careful how you use each thread's stack for:

- large arrays and structures declared locally to functions and ISRs,
- function (i.e., subroutine) nesting,
- interrupt nesting,
- library functions stack usage, and
- function calls with many arguments.

To summarize, a multithreading system requires more code space (Flash) and data space (RAM) than a foreground / background system. The amount of extra Flash depends only on the size of the kernel, and the amount of RAM depends on the number of threads in your system.

10.9 Advantages and Disadvantages of Real-Time Operating Systems

An RTOS allows real-time applications to be designed and expanded easily; functions can be added without requiring major changes to the software. The use of an RTOS simplifies the design process by splitting the application code into separate threads. With a preemptive RTOS, all time-critical events are handled as quickly and as efficiently as possible. An RTOS allows you to make better use of your resources by providing you with valuable services, such as semaphores, mailboxes, queues, time delays, timeouts, etc.

You should consider using a real-time kernel if your application can afford the extra requirements: extra cost of the kernel, more ROM/RAM, and 2 to 4 percent additional CPU overhead.