1.1 Real-Time OS

[edit]

Introduction

Play Video

A computer system has many types of resources such as memory, I/O, data, and processors. A **real-time operating system** (RTOS) is software that manages these resources, guaranteeing all timing constraints are satisfied. Figure 1.2 illustrates the relationship between hardware and software. On the left is a basic system without an operating system. Software is written by a single vendor for a specific microcontroller. As the system becomes more complex (middle figure), an operating system facilitates the integration of software from multiple vendors. By providing a hardware abstraction layer (HAL) an operating system simplifies porting application code from one microcontroller to another. In order to provide additional processing power, embedded systems of the future will require multiple microcontrollers, processors with specialized coprocessors and/or a microcontroller with multiple cores (right figure). Synchronization and assigning tasks across distributed processors are important factors. As these systems become more complex, the role of the operating system will be increasingly important.

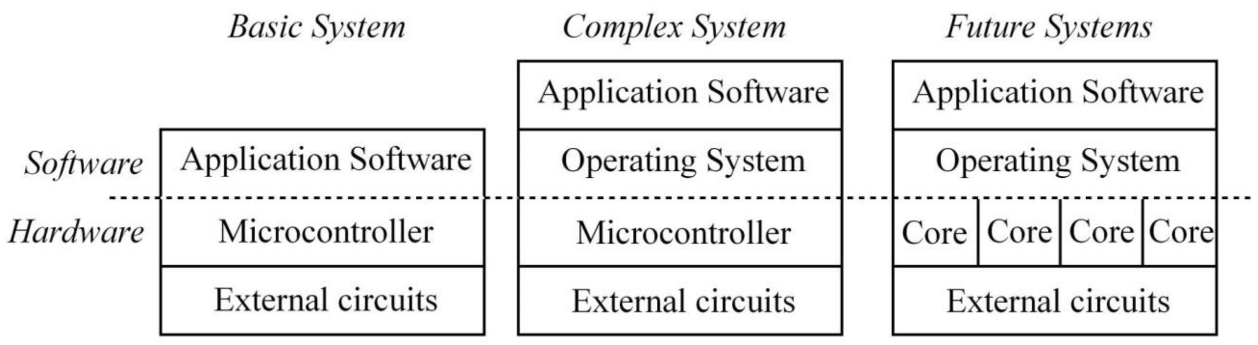


Figure 1.2. An operating system is a software layer between the application software and the hardware.

The RTOS must **manage resources** like memory, processor and I/O. The RTOS will **guarantee strict timing constraints** and provide **reliable** operation. The RTOS will support **synchronization** and **communication** between tasks. As complex systems are built the RTOS manages the **integration of components**. **Evolution** is the notion of a system changing to improve performance, features and reliability. The RTOS must manage change. When designing a new system, it is good design practice to build a new system by changing an existing system. The notion of **portability** is the ease at which one system can be changed or adapted to create another system. The **response time** or **latency** is the delay from a request to the beginning of the service of that request. There are many definitions of bandwidth. In this book we define **bandwidth** as the number of information bytes/sec that can be transferred or processed.

We can compare and contrast regular operating systems with real-time operating systems

Regular OS	**Real-time OS**
Complex	Simple
Best effort	Guaranteed response
Fairness	Strict timing constraints
Average bandwidth	Minimum and maximum limits
Unknown components	Known components
Unpredictable behavior	Predictable behavior
Plug and play	Upgradable

Table 1.1. Comparison of regular and real-time operating systems.

From Table 1.1 we see that real-time operating systems have to be simple so they may be predictable. While traditional operating systems gauge their performance in terms of response time and fairness, real-time operating systems target strict timing constraints and upper, lower bounds on bandwidth. One can expect to know all the components of the system at design time and component changes happen much more infrequently.

Embedded Systems

Play Video

An **embedded system** is a smart device with a processor that has a special and dedicated purpose. The user usually does not or cannot upgrade the hardware/software or change what the system does. **Real time** means that the embedded system must respond to critical events within a strictly defined time, called the deadline. A guarantee to meet all deadlines can only be made if the behavior of the operating system can be predicted. In other words the timing must be deterministic. There are five types of software functions the processor can perform in an embedded system. Similar to a general-purpose computer, it can perform mathematical and/or data processing operations. It can analyze data and make decisions based on the data. A second type involves handling and managing time: as an input (e.g., measure period), an output (e.g., output waveforms), and a means to synchronize tasks (e.g., run 1000 times a second). A third type involves real-time input/output for the purpose of measurement or control. The fourth type involves digital signal processing (DSP), which are mathematical calculations on data streams. Examples include audio, video, radar, and sonar. The last type is communication and networking. As embedded systems become more complex, how the components are linked together will become increasingly important.

Six **constraints** typify an embedded system. First, they are small size. For example, many systems must be handheld. Second, they must have low weight. If the device is deployed in a system that moves, e.g., attached to a human, aircraft or vehicle, then weight incurs an energy cost. Third, they often must be low power. For example, they might need to operate for a long time on battery power. Low power also impacts the amount of heat they are allowed to generate. Fourth, embedded systems often must operate in harsh environments, such as heat, pressure, vibrations, and shock. They may be subject to noisy power, RF interference, water, and chemicals. Fifth, embedded systems are often used in safety critical systems. Real-time behavior is essential. For these systems they must function properly at extremely high levels of reliability. Lastly, embedded systems are extremely sensitive to cost. Most applications are profit-driven. For high-volume systems a difference in pennies can significantly affect profit.

1.2.1 Computers, processors, memory, and microcontrollers[edit]

Given that an operating system is a manager of resources provided by the underlying architecture, it would serve the reader well to get acquainted with the architecture the OS must manage. In this section we will delve into these details of the building blocks of a computer architecture, followed by the specifics of the ARM Cortex M4 processor architecture, in particular TI's microcontrollers implementation of the ARM ISA.

Play Video

A **computer** combines a central processing unit (CPU), random access memory (RAM), read only memory (ROM), and input/output (I/O) ports. The common bus in Figure 1.3 defines the **Von Neumann architecture**.

Software is an ordered sequence of very specific instructions that are stored in memory, defining exactly what and when certain tasks are to be performed.

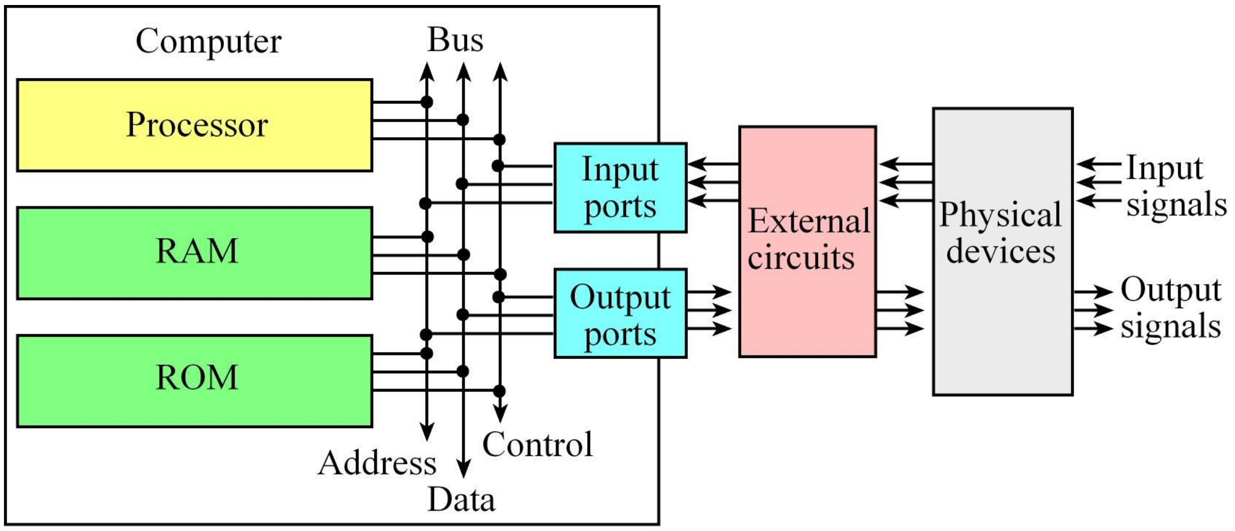


Figure 1.3. The basic components of a computer system include processor, memory and I/O.

The ARM Cortex-M processor has four major components, as illustrated in Figure 1.4. There are **bus interface units** (BIU) that read data from the bus during a read cycle and write data onto the bus during a write cycle. The BIU always drives the address bus and the control signals of the bus. The **effective address register** (EAR) contains the memory address used to fetch the data needed for the current instruction. Cortex-M microcontrollers execute Thumb instructions extended with Thumb-2 technology. An overview of these instructions will be presented in Section 1.5. Many functions in an operating system will require detailed understanding of the architecture and assembly language.

The **control unit** (CU) orchestrates the sequence of operations in the processor. The CU issues commands to the other three components. The **instruction register** (IR) contains the operation code (or op code) for the current instruction. When extended with Thumb-2 technology, op codes are either 16 or 32 bits wide.

The **arithmetic logic unit** (ALU) performs arithmetic and logic operations. Addition, subtraction, multiplication and division are examples of arithmetic operations. Examples of logic operations are, and, or, exclusive-or, and shift. Many processors used in embedded applications support specialized operations such as table lookup, multiply and accumulate, and overflow detection.

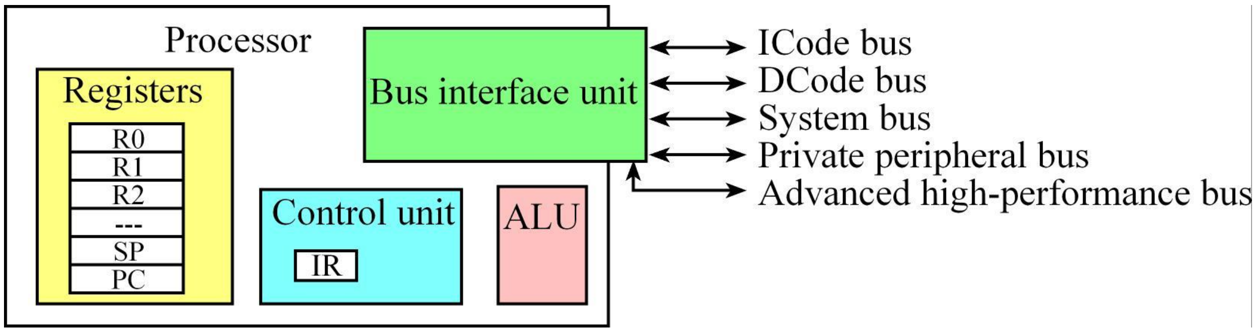


Figure 1.4. The four basic components of a processor.

A very small microcomputer, called a **microcontroller**, contains all the components of a computer (processor, memory, I/O) on a single chip. The Atmel ATtiny and the TI TM4C123 are examples of microcontrollers. Because a microcomputer is a small computer, this term can be confusing because it is used to describe a wide range of systems from a 6-pin ATtiny4 running at 1 MHz with 512 bytes of program memory to a personal computer with state-of-the-art 64-bit multi-core processor running at multi-GHz speeds having terabytes of storage.

In an embedded system the software is converted to machine code, which is a list of instructions, and stored in nonvolatile flash ROM. As instructions are fetched, they are placed in a **pipeline**. This allows instruction fetching to run ahead of execution. Instructions on the Cortex-M processor are fetched in order and executed in order. However, it can execute one instruction while fetching the next. Many high-speed processors allow out of order execution, support parallel execution on multiple cores, and employ branch prediction.

On the ARM Cortex-M processor, an instruction may read memory or write memory, but does not read and write memory in the same instruction. Each of the phases may require one or more bus cycles to complete. Each bus cycle reads or writes one piece of data. Because of the multiple bus architecture, most instructions execute in one or two cycles. For more information on the time to execute instructions, see Table 3.1 in the Cortex-M Technical Reference Manual.

Figure 1.5 shows a simplified block diagram of a microcontroller based on the ARM Cortex-M processor. It is a **Harvard architecture** because it has separate data and instruction buses. The instruction set combines the high performance typical of a 32-bit processor with high code density typical of 8-bit and 16-bit microcontrollers. Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface. There are many sophisticated debugging features utilizing the DCode bus. An **interrupt** is a hardware-triggered software function, which is extremely important for real-time embedded systems. The **latency** of an interrupt service is the time between hardware trigger and software response. Some internal peripherals, like the nested vectored interrupt controller (NVIC), communicate directly with the processor via the private peripheral bus (PPB). The tight integration of the processor and interrupt controller provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency.

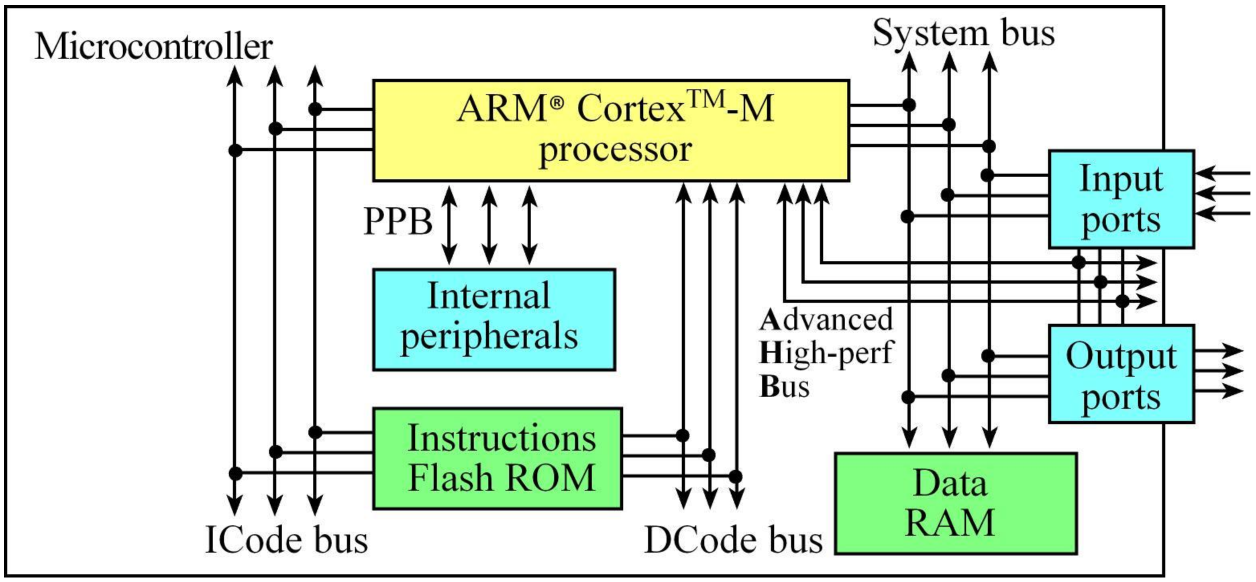


Figure 1.5. Harvard architecture of an ARM Cortex-M-based microcontroller.

1.2.2. Memory[edit]

One kibibyte (KiB) equals 1024 bytes of memory. The TM4C123 has 256 kibibytes (2^{18} bytes) of flash ROM and 32 kibibytes (2^{15} bytes) of RAM. The MSP432 also has 256 kibibytes (2^{18} bytes) of flash ROM but has 64 kibibytes (2^{16} bytes) of RAM. We view the memory as continuous virtual address space with the RAM beginning at 0x2000.0000, and the flash ROM beginning at 0x0000.0000.

Play Video

The microcontrollers in the Cortex-M family differ by the amount of memory and by the types of I/O modules. There are hundreds of members in this family; some of them are listed in Table 1.2. The **memory maps** of TM4C123 and MSP432 are shown in Figure 1.6. Although this course focuses on two microcontrollers from Texas Instruments, all ARM Cortex-M microcontrollers have similar memory maps. In general, Flash ROM begins at address 0x0000.0000, RAM begins at 0x2000.0000, the peripheral I/O space is from 0x4000.0000 to 0x5FFF.FFFF, and I/O modules on the private peripheral bus exist from 0xE000.0000 to 0xE00F.FFFF. In particular, the only differences in the memory map for the various members of the Cortex-M family are the ending addresses of the flash and RAM.

Part number	RAM	Flash	I/O	I/O modules
MSP432P401RIPZ	64	256	84	floating point, DMA
LM4F120H5QR	32	256	43	floating point, CAN, DMA, USB
TM4C123GH6PM	32	256	43	floating point, CAN, DMA, USB, PWM
STM32F051R8T6	8	64	55	DAC, Touch sensor, DMA, I2S, HDMI, P
MKE02Z64VQH2	4	64v	53	PWM
	KiB	KiB	pins	

Table 1.2. Memory and I/O modules (all have SysTick, RTC, timers, UART, I2C, SSI, and ADC).

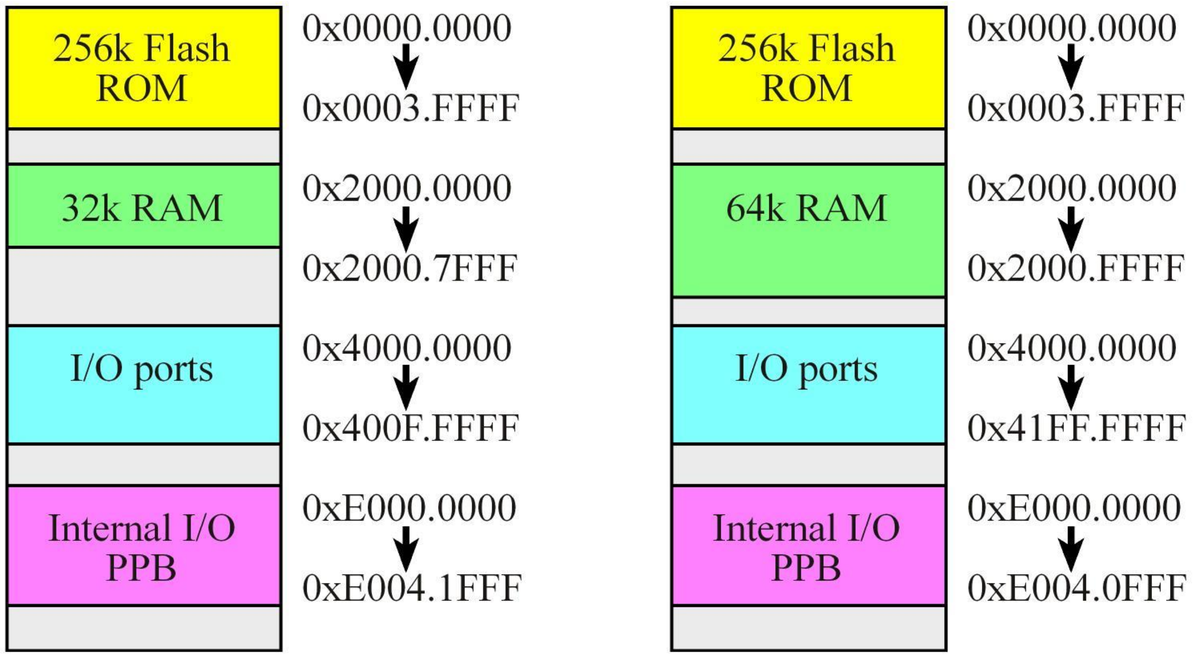


Figure 1.6. Memory map of the TM4C123 with 256k ROM and 32k RAM and the MSP432 with 256k ROM and 64k RAM.

Having **multiple buses** means the processor can perform multiple tasks in parallel. On the TM4C123, general purpose input/output (GPIO) ports can be accessed using either the PPB or AHPB. The following is some of the tasks that can occur in parallel

-
  - ICode bus Fetch opcode from ROM
  - DCode bus Read constant data from ROM
  - System bus Read/write data from RAM or I/O, fetch opcode from RAM
  - PPB Read/write data from internal peripherals like the NVIC
  - AHPB Read/write data from internal peripherals like the USB

Instructions and data are accessed using a common bus on a von Neumann machine. The Cortex-M processor is a Harvard architecture because instructions are fetched on the ICode bus and data accessed on the system bus. The address signals on the ARM Cortex-M processor include 32 lines, which together specify the memory address (0x0000.0000 to 0xFFFF.FFFF) that is currently being accessed. The address specifies both which module (input, output, RAM, or ROM) as well as which cell within the module will communicate with the processor. The data signals contain the information that is being transferred and also include 32 bits. However, on the system bus it can also transfer 8-bit or 16-bit data. The control signals specify the timing, the size, and the direction of the transfer.

1.3.1. Registers[edit]

Play Video

The registers on an ARM Cortex-M processor are depicted in Figure 1.7. R0 to R12 are general purpose registers and contain either data or addresses. Register R13 (also called the stack pointer, SP) points to the top element of the stack. Actually, there are two stack pointers: the main stack pointer (MSP) and the process stack pointer (PSP). Only one stack pointer is active at a time. In a high-reliability operating system, we could activate the PSP for user software and the MSP for operating system software. This way the user program could crash without disturbing the operating system. Most of the commercially available real-time operating systems available on the Cortex M will use the PSP for user code and MSP for OS code. Register R14 (also called the link register, LR) is used to store the return location for functions. The LR is also used in a special way during exceptions, such as interrupts. Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC by the length (in bytes) of the instruction fetched.

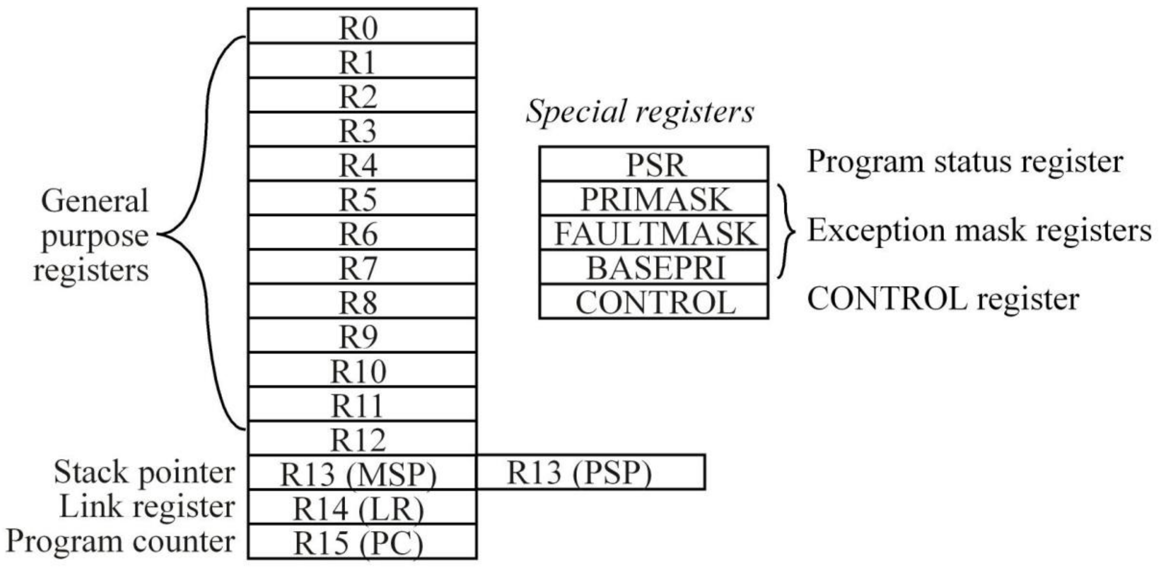


Figure 1.7. The registers on the ARM Cortex-M processor.

The **ARM Architecture Procedure Call Standard**, AAPCS, part of the ARM Application Binary Interface (ABI), uses registers R0, R1, R2, and R3 to pass input parameters into a C function or an assembly subroutine. Also according to AAPCS we place the return parameter in Register R0. The standard requires functions to preserve the contents of R4-R11. In other words, functions save R4-R11, use R4-R11, and then restore R4-R11 before returning. Another restriction is to keep the stack aligned to 64 bits, by pushing and popping an even number of registers.

There are three status registers named Application Program Status Register (APSR), the Interrupt Program Status Register (IPSR), and the Execution Program Status Register (EPSR) as shown in Figure 1.8. These registers can be accessed individually or in combination as the Program Status Register (PSR).

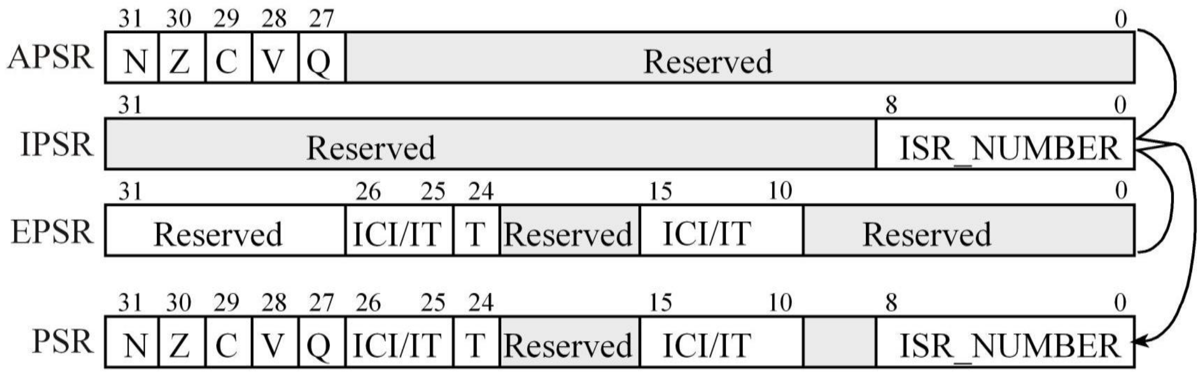


Figure 1.8. The program status register of the ARM Cortex-M processor.

The N, Z, V, C, and Q bits signify the status of the previous ALU operation. Many instructions set these bits to signify the result of the operation. In general, the **N bit** is set after an arithmetical or logical operation signifying whether or not the result is negative. Similarly, the **Z bit** is set if the result is zero. The **C bit** means carry and is set on an unsigned overflow, and the **V bit** signifies signed overflow. The **Q bit** is the sticky saturation flag, indicating that “saturation” has occurred, and is set by the SSAT and USAT instructions.

The **T bit** will always be 1, indicating the ARM Cortex-M processor is executing Thumb instructions. The ICI/IT bits are used by interrupts and by IF-THEN instructions. The ISR_NUMBER indicates which interrupt if any the processor is handling. Bit 0 of the special register PRIMASK is the interrupt mask bit, or **I bit**. If this bit is 1 most interrupts and exceptions are not allowed. If the bit is 0, then interrupts are allowed. Bit 0 of the special register FAULTMASK is the fault mask bit. If this bit is 1 all interrupts and faults are disallowed. If the bit is 0, then interrupts and faults are allowed. The nonmaskable interrupt (NMI) is not affected by these mask bits. The BASEPRI register defines the priority of the executing software. It prevents interrupts with lower or equal priority from interrupting the current execution but allows higher priority interrupts. For example if BASEPRI equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. The details of interrupt processing will be presented in Chapters 2 and 3.

1.3.2. Stack[edit]

Play Video

The **stack** is a last-in-first-out temporary storage. Managing the stack is an important function for the operating system. To create a stack, a block of RAM is allocated for this temporary storage. On the ARM Cortex-M processor, the stack always operates on 32-bit data. All stack accesses are word aligned, which means the least significant two bits of SP must always be 0. The stack pointer (SP) points to the 32-bit data on the top of the stack.

To **push** data we first decrement the SP by 4 then store 32-bit data at the SP. We refer to the most recent item pushed as the "top of the stack". If though it is called the "top", this item is actually the stored at the lowest address! When data is pushed it is saved on the stack.

To **pop** data from the stack, the 32-bit information pointed to by SP is first retrieved, and then the stack pointer is incremented by 4. SP points to the last item pushed, which will also be the next item to be popped. A stack is a last in first out (LIFO) storage, meaning the pop operation will retrieve the newest or most recently saved value. When data is popped it is removed from the stack.

The boxes in Figure 1.9 represent 32-bit storage elements in RAM. The colored boxes in the figure refer to actual data stored on the stack. The white boxes refer to locations in the allocated stack area that do not contain data. These allocated but not used locations are called the **free area**. This figure illustrates how the stack is used to push the contents of Registers R1, and R2 in that order. Assume Register R0 initially contains the value 13, R1 contains 2 and R2 contains 5. The drawing on the left shows the initial stack. The software executes these three instructions, first pushing two elements, and then popping one.

```
      PUSH {R1}
      PUSH {R2}
      POP {R0}
```

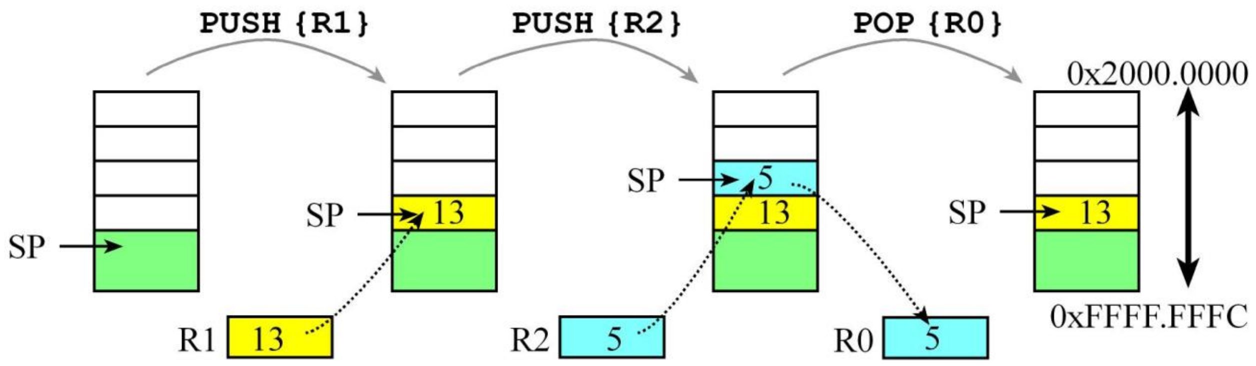


Figure 1.9. Stack picture showing two pushes and one pop. Push stores data onto the stack, pop retrieves/removes data from the stack.

The instruction **PUSH {R1}** saves the value of R1 on the stack. It first decrements SP by 4, and then it stores the contents of R1 into the memory location pointed to by SP. Assuming R1, R2 had values 13, 5 respectively, after the two push instructions the stack contains the numbers 13 and 5, with 5 on top, (third picture in Figure 1.9). The instruction **POP {R0}** retrieves the most recent data from the stack. It first moves the value from memory pointed to by SP into R0, and then it increments SP by 4.

In Figure 1.9 we pushed two elements and then popped one, so the stack has more data than when we started. Normally, all blocks of software will first push and then pop, where the number of pops equals the number of pushes. Having an equal number of pushes and pops is defined as **balancing the stack**.

We define the 32-bit word pointed to by SP as the top entry of the stack. If it exists, we define the 32-bit data immediately below the top, at SP+4, as next to top. Proper use of the stack requires following these important guidelines

1. Functions should have an equal number of pushes and pops
2. Stack accesses (push or pop) should not be performed outside the allocated area
3. Stack reads and writes should not be performed within the free area
4. Push and pop are 32-bit operation

Functions that violate rule number 1 will probably crash when incorrect data are popped off at a later time. Violations of rule number 2 usually result from a stack underflow or overflow. Overflow occurs when the number of elements became larger than the allocated space. Stack underflow is caused when there are more pops than pushes, and is always the result of a software bug. A stack overflow can be caused by two reasons. If the software mistakenly pushes more than it pops, then the stack pointer will eventually overflow its bounds. Even when there is exactly one pop for each push, a stack overflow can occur if the stack is not allocated large enough. The processor will generate a bus fault when the software tries read from or write to an address that doesn't exist. If valid RAM exists below the stack then further stack operations will corrupt data in this memory.

When debugging Lab 2, it will be important to develop techniques to visualize the stack. Stack errors represent typical failure modes of an operating system.

The stack plays an important role in interrupt processing. Executing an interrupt service routine will automatically push eight 32-bit words onto the stack. Since interrupts are triggered by hardware events, exactly when interrupts occur is not under software control. Therefore, violations of rule 3 will cause erratic behavior when operating with interrupts.

The processor allows for two stacks, the main stack (MSP) and the process stack (PSP), with independent copies of the stack pointer. The OS would run safer if the application code used the PSP and the OS code used the MSP. However to make the OS simpler we will run both the application and the OS using the MSP.

1.3.3. Reset and Operating modes[edit]

Play Video

A **reset** occurs immediately after power is applied and can also occur by pushing the reset button available on most boards. After a reset, the processor is in thread mode, running at a privileged level, and using the MSP stack pointer. The 32-bit value at flash ROM location 0 is loaded into the SP. A reset also loads the 32-bit value at location 4 into the PC. This value is called the reset vector. All instructions are halfword aligned. Thus, the least significant bit of PC must be 0. However, the assembler will set the least significant bit in the reset vector, so the processor will properly initialize the thumb bit (T) in the PSR. On the ARM Cortex-M, the T bit should always be set to 1. On reset, the processor initializes the LR to 0xFFFFFFFF.

The ARM Cortex-M processor has two privilege levels called privileged and unprivileged. Bit 0 of the **CONTROL** register is the **thread privilege level**(TPL). If TPL is 1 the processor level is privileged. If the bit is 0, then processor level is unprivileged. Running at the unprivileged level prevents access to various features, including the system timer and the interrupt controller. Bit 1 of the CONTROL register is the active stack pointer selection (ASPSEL). If ASPSEL is 1, the processor uses the PSP for its stack pointer. If ASPSEL is 0, the MSP is used. When designing a high-reliability operating system, we will run the user code at an unprivileged level using the PSP and the OS code at the privileged level using the MSP.

The processor knows whether it is running in the foreground (i.e., the main program) or in the background (i.e., an interrupt service routine). ARM defines the foreground as **thread mode**, and the background as **handler mode**. Switching between thread and handler modes occurs automatically. The processor begins in thread mode, signified by ISR_NUMBER=0. Whenever it is servicing an interrupt it switches to handler mode, signified by setting ISR_NUMBER to specify which interrupt is being processed. All interrupt service routines run using the MSP. In particular, the context is saved onto whichever stack pointer is active, but during the execution of the ISR, the MSP is used. For a high reliability operation all interrupt service routines will reside in the operating system. User code can be run under interrupt control by providing hooks, which are function pointers. The user can set function pointers during initialization, and the operating system will call the function during the interrupt service routine.

Observation: Processor modes and the stack are essential components of building a reliable operating system. In particular the processor mode is an architectural feature that allows the operating system to restrict access to critical system resources.

1.4.1. Input/Output[edit]

Play Video

I/O is an important part of embedded systems in general. One of the important features of an operating system is to manage I/O. Input and output are the means of an embedded system to interact with its world. The external devices attached to the microcontroller provide functionality for the system. These devices connect to the microcontroller through ports. A pin is a specific wire on the microcontroller through which we perform input or output. A collection of pins grouped by common functionality is called a **port**. An **input port** is hardware on the microcontroller that allows information about the external world to enter into the computer. The microcontroller also has hardware called an **output port** to send information out to the external world. The GPIO (General Purpose Input Output) pins on a microcontroller are programmable to be digital input, digital output, analog input or complex and protocol (like UART etc.) specific. Microcontrollers use most of their pins for I/O (called GPIO), see Figure 1.10. Only a few pins are not used for I/O. Examples of pins not used for I/O include power, ground, reset, debugging, and the clock. More specifically, the TM4C123 uses 43 of its 64 pins for I/O. Similarly, the MSP432 uses 84 of its 100 pins for I/O.

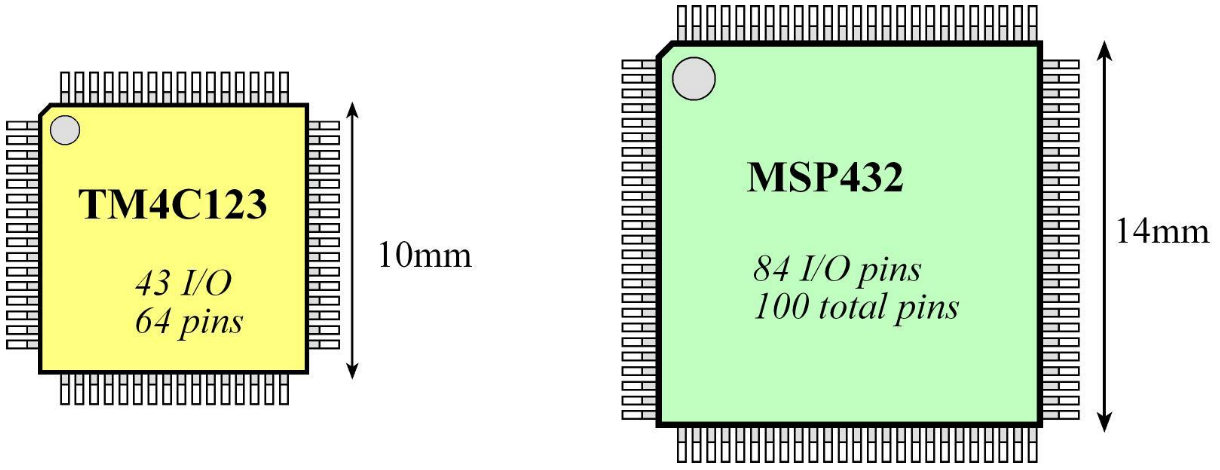


Figure 1.10. Most of the pins on the microcontroller can perform input/output.

An interface is defined as the collection of the I/O port, external electronics, physical devices, and the software, which combine to allow the computer to communicate with the external world. An example of an input interface is a switch, where the operator toggles the switch, and the software can recognize the switch position. An example of an output interface is a light-emitting diode (LED), where the software can turn the light on and off, and the operator can see whether or not the light is shining. There is a wide range of possible inputs and outputs, which can exist in either digital or analog form. In general, we can classify I/O interfaces into four categories

- **Parallel/digital** - binary data are available simultaneously on a group of lines
- **Serial** - binary data are available one bit at a time on a single line
- **Analog** - data are encoded as an electrical voltage, current or power
- **Time** - data are encoded as a period, frequency, pulse width or phase shift

In a system with **memory-mapped I/O**, as shown in Figure 1.11, the I/O ports are connected to the processor in a manner similar to memory. I/O ports are assigned addresses, and the software accesses I/O using reads and writes to the specific I/O addresses. These addresses appear like regular memory addresses, except accessing them results in manipulation of a functionality of the mapped I/O port, hence the term memory-mapped I/O. As a result, the software inputs from an input port using the same instructions as it would if it were reading from memory. Similarly, the software outputs from an output port using the same instructions as it would if it were writing to memory.

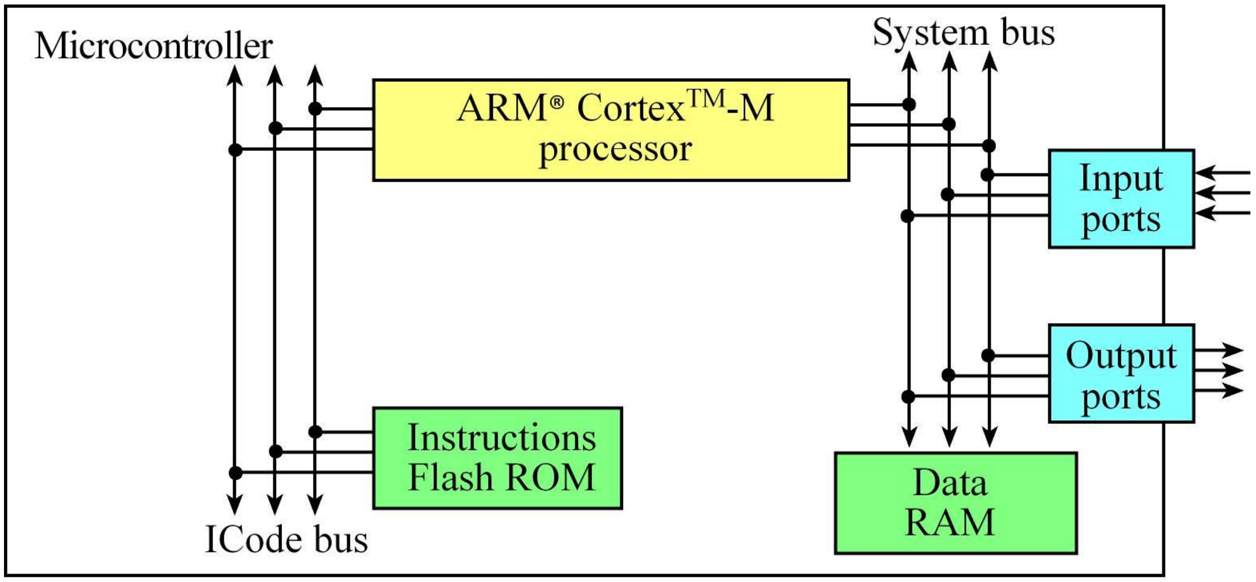


Figure 1.11. Memory-mapped input/output.

Most pins on Cortex M microcontrollers can be used for **general purpose I/O** (GPIO) called regular functions or for more complex functions called alternate functions. For example, port pins PA1 and PA0 on the TM4C123 can be either regular parallel port pins, or an asynchronous serial port called universal asynchronous receiver/transmitter (UART). Some of the alternative functions used in this class are:

- **UART** Universal asynchronous receiver/transmitter
- **SSI** or **SPI** Synchronous serial interface or serial peripheral interface
- **I2C** Inter-integrated circuit
- **Timer** Periodic interrupts
- **PWM** Pulse width modulation
- **ADC** Analog to digital converter, measurement analog signals

The **UART** can be used for serial communication between computers. It is asynchronous and allows for simultaneous communication in both directions. The **SSI** (also called **SPI**) is used to interface medium-speed I/O devices. In this class, we will use **SSI** to interface a graphics display. **I2C** is a simple I/O bus that we will use to interface low speed peripheral devices. In this class we use **I2C** to interface a light sensor and a temperature sensor. We will use the timer modules to create periodic interrupts. **PWM** outputs could be used to apply variable power to motor interfaces. However, in this class we use **PWM**to adjust the volume of the buzzer.
The **ADC** will be used to measure the amplitude of analog signals, and will be important in data acquisition systems. In this class we will connect the microphone, joystick and accelerometer to the ADC.

Joint Test Action Group (**JTAG**), standardized as the IEEE 1149.1, is a standard test access port used to program and debug the microcontroller board. Each microcontroller uses four port pins for the JTAG interface.

1.4.2. TM4C123 Microcontroller[edit]

Play Video

Figure 1.12 draws the I/O port structure for the TM4C123GH6PM, the microcontroller is used on the EK-TM4C123GXL LaunchPad. Pins on the TM4C family can be assigned to as many as eight different I/O functions. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PB4 can be a digital I/O, ADC, SSI, PWM, timer or CAN pin. The TM4C123GH6PM has eight UART ports, four SSI ports, four I2C ports, two 12-bit ADCs, twelve timers, two PWMs, a CAN port, and a USB interface. There are 43 I/O lines. There are twelve ADC inputs; each ADC can convert up to 1M samples per second.

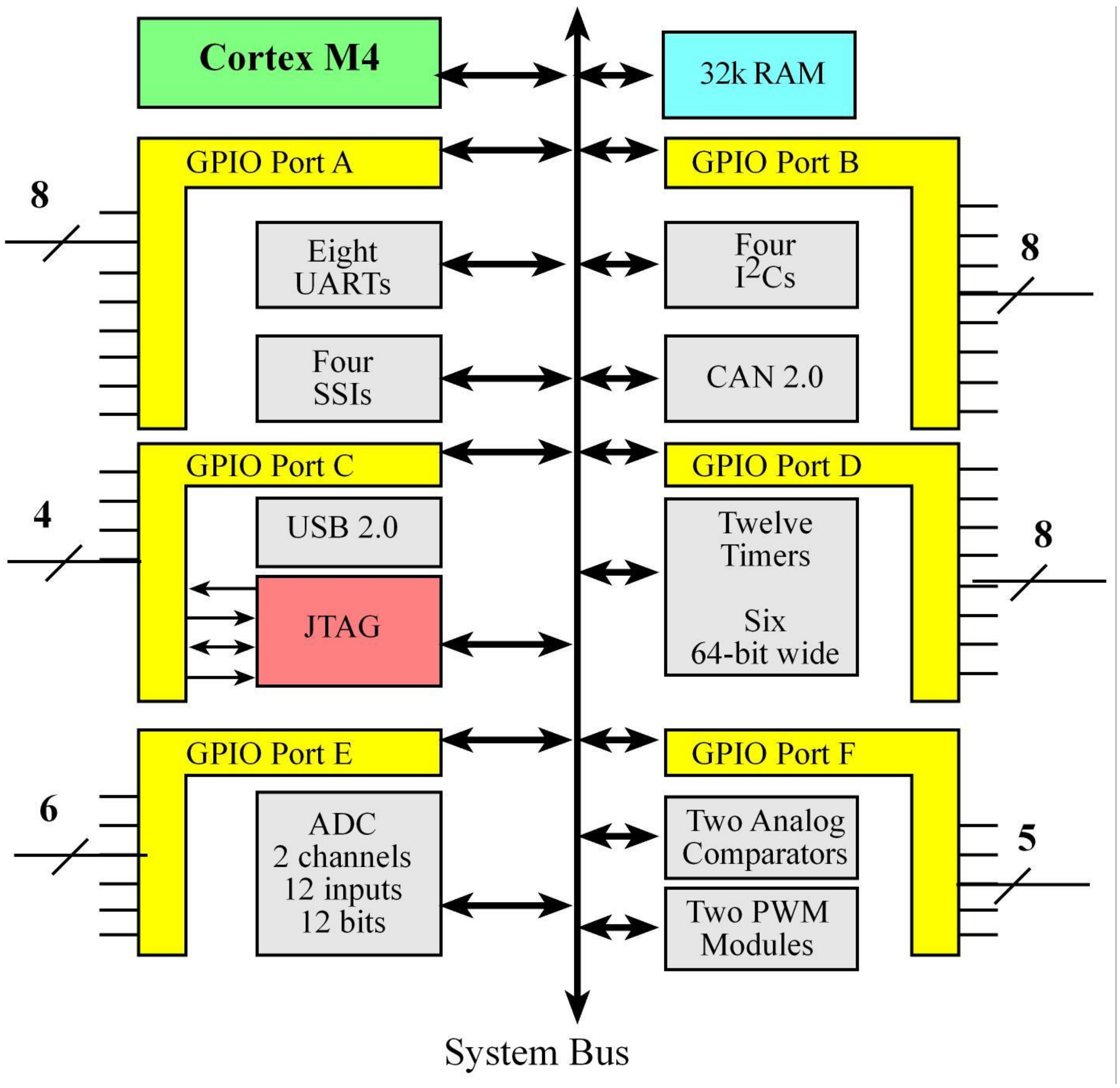


Figure 1.12. All the I/O port pins for the TM4C123GH6PM microcontroller.

Figure 1.13 shows the port pin connections for the hardware using in this class. There are six ports (A, B, C, D, E, and F). You can see from this figure that all of the ports share operation with multiple devices. For example, Port A is used for

- UART to PC
- Light sensor input
- Temperature sensor input
- LCD output

This overlap of features must be managed by the operating system. More information about the hardware/software interfaces used in this class will be presented later in section 1.6.

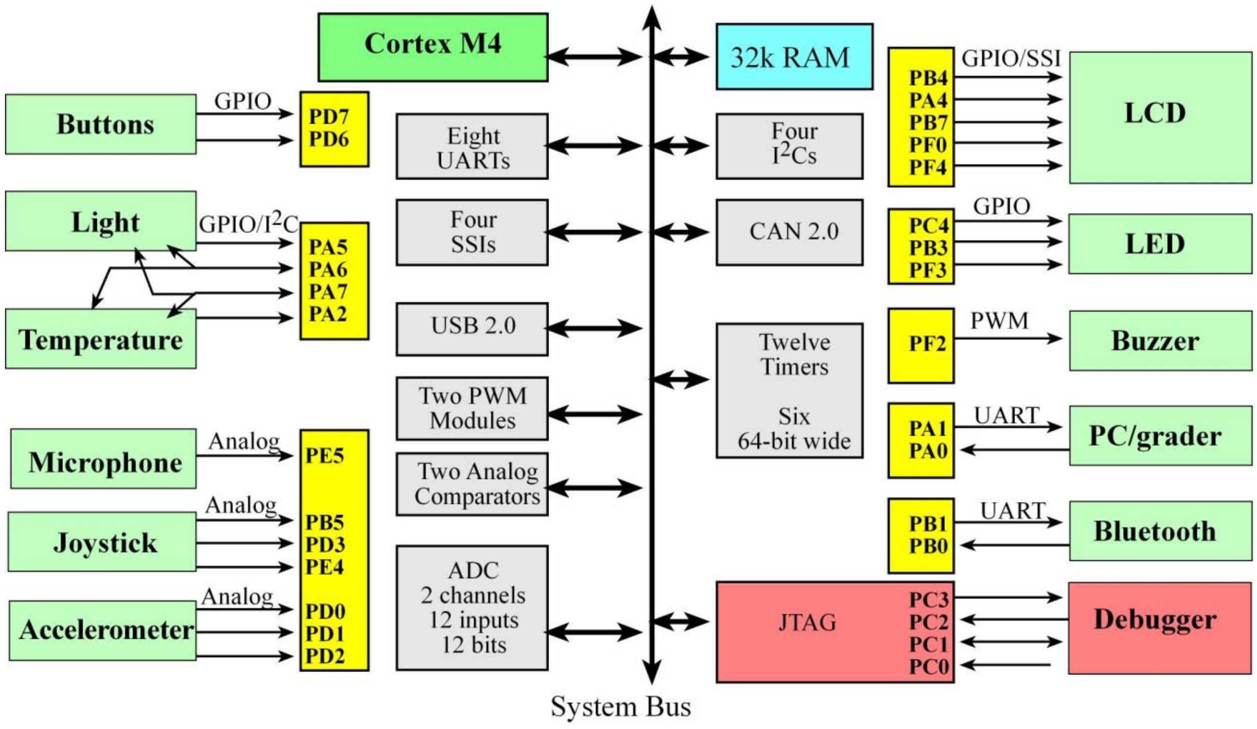


Figure 1.13. I/O port pins for the TM4C123GH6PM used in this class with the Educational MKII BoosterPack (BOOSTXL-EDUMKII). PD7 means Port D pin 7.

Figure 1.14 shows the TM4C123 LaunchPad. In this class you can use either the TM4C123 LaunchPad or the MSP432 LaunchPad. There are some older LaunchPads based on the LM4F120, which are virtually identical with the TM4C123. If you have an LM4F120 system all the TM4C123 code will run on the LM4F120 without modification.

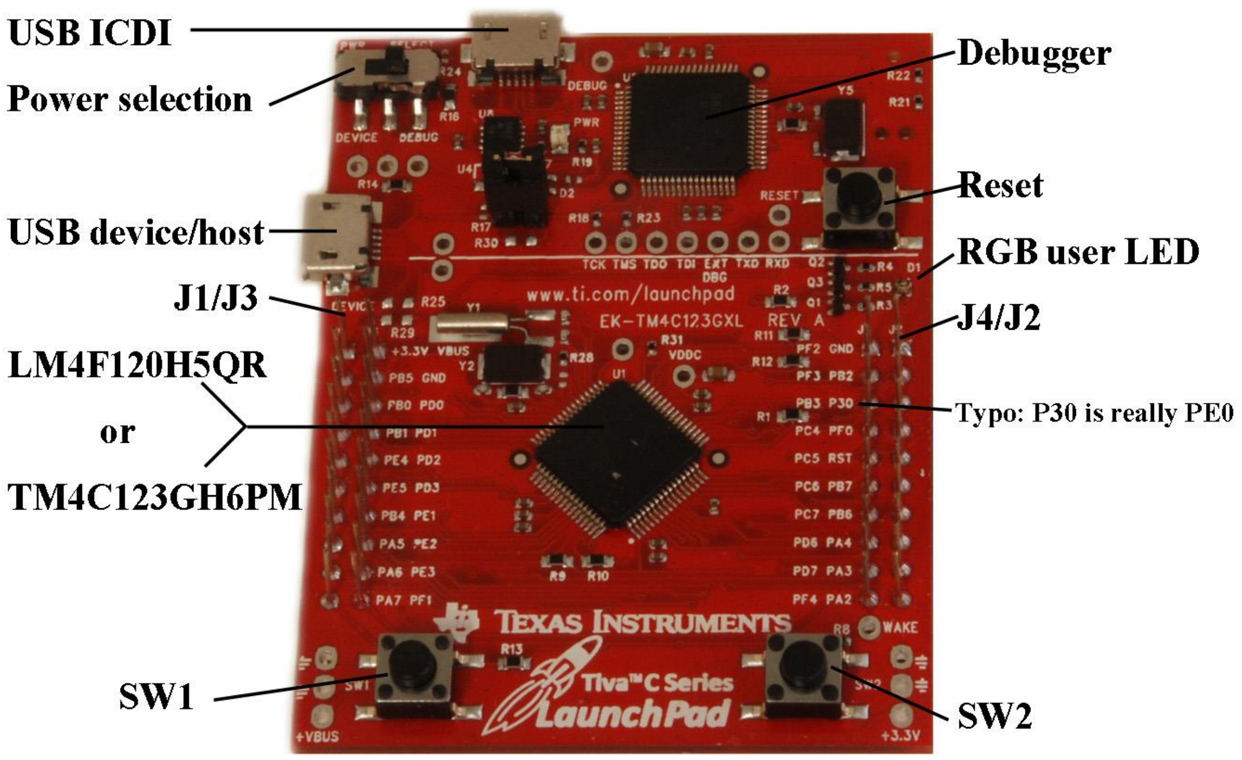


Figure 1.14. Tiva TM4C123 Launchpad Evaluation Board based on the TM4C123GH6PM.

Unfortunately, the TM4C123/LM4F120 LaunchPad connects PB6 to PD0, and PB7 to PD1. For this class you **MUST** remove the R9 and R10 resistor in order for the LCD to operate properly.

Play Video

The TM4C123 LaunchPad evaluation board has two switches and one 3-color LED. See Figure 1.15. In this class we will not use the switches and LED on the LaunchPad, but rather focus on the hardware provided by the MK-II BoosterPack.

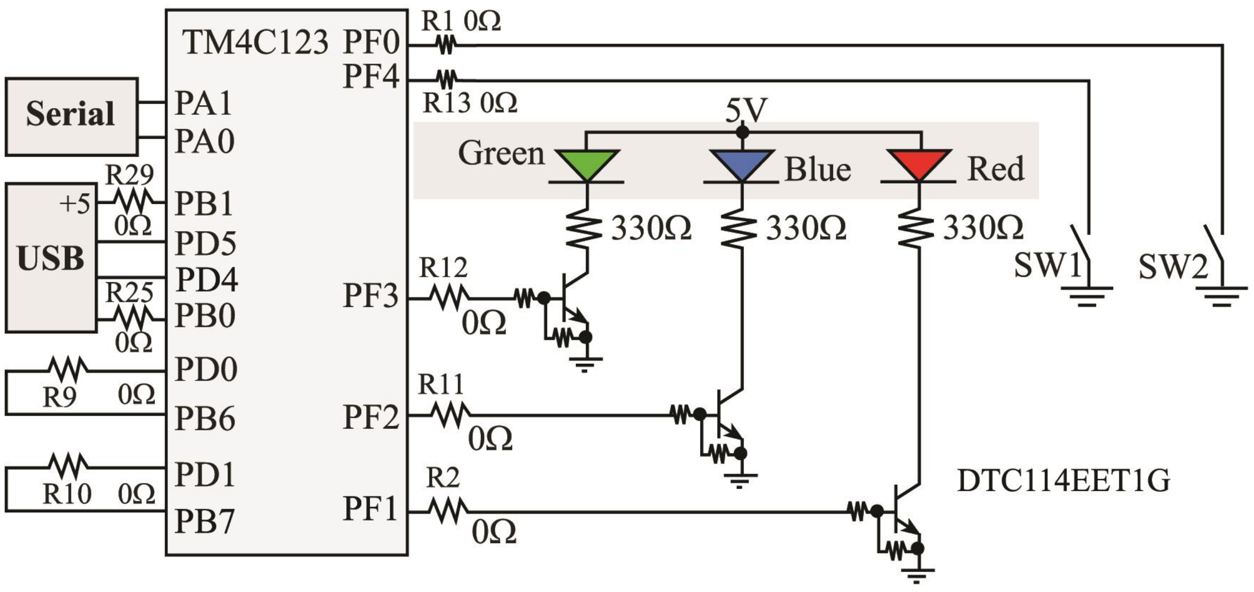


Figure 1.15. Switch and LED interfaces on the Texas Instruments TM4C123 LaunchPad Evaluation Board. The zero ohm resistors can be removed so the corresponding pin can be used for its regular purpose. The LM4F120 is similar (except for the USB interface).

The LaunchPad has four 10-pin connectors, labeled as J1 J2 J3 J4 in Figures 1.14 and 1.16, to which you can attach your external signals. The top side of these connectors has male pins and the bottom side has female sockets. The intent is to stack boards together to make a layered system. Texas Instruments also supplies BoosterPacks, which are pre-made external devices that will plug into this 40-pin connector.

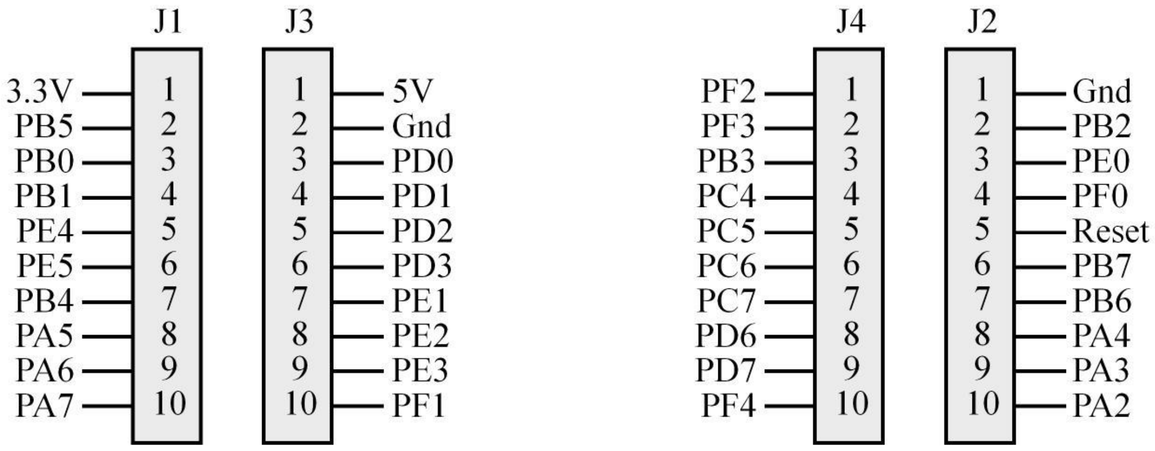


Figure 1.16. Interface connectors on the Texas Instruments TM4C123 LaunchPad Evaluation Board.

The intent is to stack boards together to make a layered system, see Figure 1.17. The engineering community has developed BoosterPacks, which are pre-made external devices that will plug into this 40-pin connector. Figure 1.17 shows a system with a LaunchPad and two BoosterPacks.

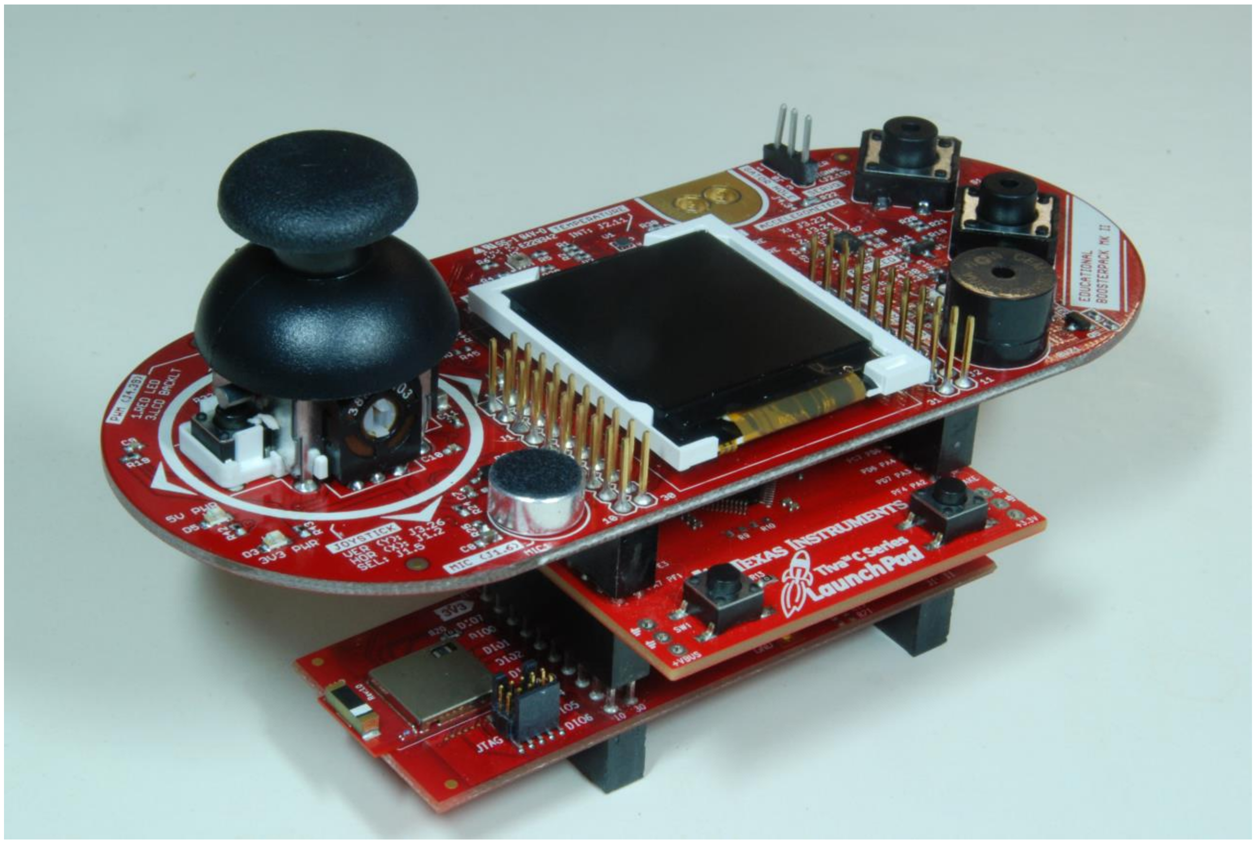


Figure 1.17. A BoosterPack plugs into either the top or bottom of a LaunchPad. In this figure the CC2650 BoosterPack is on the bottom and the MKII BoosterPack is on the top.

There are a number of good methods to connect external circuits to the LaunchPad. One method is to purchase a male to female jumper cable (e.g., item number 826 at www.adafruit.com). A second method is to solder a solid wire into a female socket (e.g., Hirose DF11-2428SCA) creating a male to female jumper wire. In this class we will use BoosterPacks, so you will not need to connect individual wires to the LaunchPad.

It is not our goal to teach I/O interfacing in this class, but rather use the I/O as a platform to develop and test real-time operating systems with Bluetooth connectivity. If you would like more information on the hardware/software aspects of interfacing, see Volume 2 of the series.

1.5.1. Assembly language syntax[edit]

This section focuses on the ARM Cortex-M assembly language. There are many ARM processors, and this book focuses on Cortex-M microcontrollers, which executes Thumb instructions extended with Thumb-2 technology. This section does not present all the Thumb instructions. Rather, we present a few basic instructions. In particular, we will show only twelve instructions, which will be both necessary and sufficient to construct your operating system. For further details, please refer to the appendix or to the ARM Cortex-M Technical Reference Manual.

Play Video

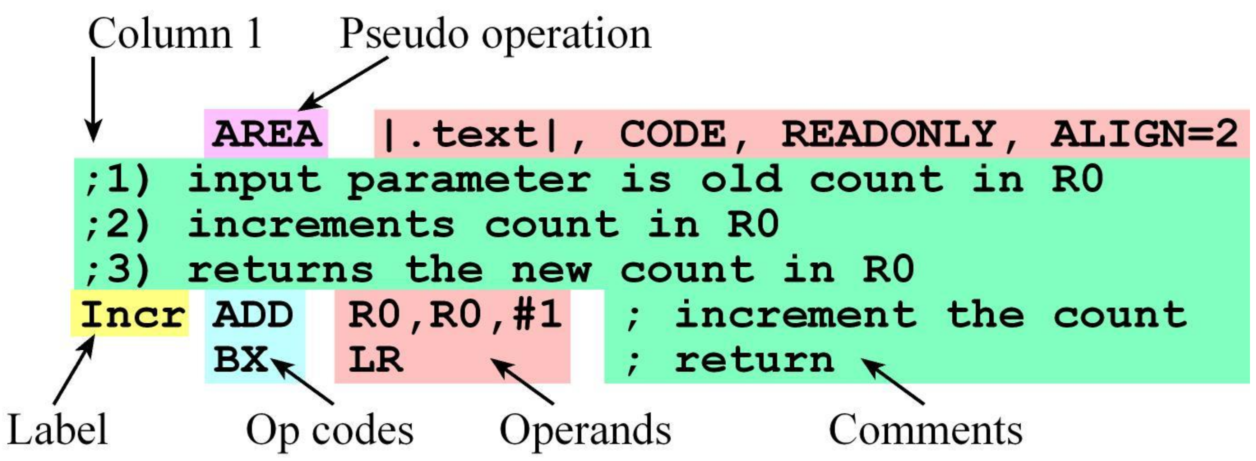
Assembly language instructions have four fields separated by spaces or tabs as illustrated in Figure 1.25.

Labels: The label field is optional and starts in the first column and is used to identify the position in memory of the current instruction. You must choose a unique name for each label.

Opcodes or pseudo-ops: The opcode field specifies which processor command to execute. The twelve op codes we will present in this class are **LDR STR MOV PUSH POP B BL BX ADD SUB CPSID** and **CPSIE**. If there is a label there must be at least one space or one tab between the label and the opcode. If there is no label then there must be at least one space or one tab at the beginning of the line. There are also pseudo-ops that the assembler uses to control features of the assembly process. Examples of pseudo-ops you will encounter in this class are **AREA EQU IMPORT EXPORT** and **ALIGN**. An op code generates machine instructions that get executed by the processor at run time, while a pseudo-op code generates instructions to the assembler that get interpreted at assembly time.

Operands: The operand field specifies where to find the data to execute the instruction. Thumb instructions have 0, 1, 2, 3, or more operands, separated by commas.

Comments: The comment field is optional and is ignored by the assembler, but allows you to describe the software, making it easier to understand. You can add optional spaces between operands in the operand field. However, a semicolon must separate the operand and comment fields. Good programmers add comments to explain what you are doing, why you are doing it, how it was tested, and how to change it in the future. Everything after the semicolon is a comment.



```
     AREA  |.text|, CODE, READONLY, ALIGN=2
;1) input parameter is old count in R0
;2) increments count in R0
;3) returns the new count in R0
Incr ADD   R0,R0,#1   ; increment the count
     BX    LR         ; return
```

Figure 1.25. Assembly instructions have four fields: labels, opcodes, operands, and comments.

The **assembler** translates assembly source code into **object code**, which are the machine instructions executed by the processor. All object code is halfword-aligned. With Thumb-2, instructions can be 16 or 32 bits wide, and the program counter bit 0 will always be 0.
The **listing** is a text file containing a mixture of the object code generated by the assembler together with our original source code.

```
Address    Object code Label Opcode Operand   comment
0000006A   F1000001    Incr  ADD    R0,R0,#1 ;increment the count
0000006E   4770              BX     LR       ;return
```

When we **build** a project all files are assembled or compiled, then linked together. The address values shown in the listing are the relative to the particular file being assembled. When the entire project is built, the files are linked together, and the linker decides exactly where in memory everything will be. After building the project, it can be downloaded, which programs the object code into flash ROM.

In general, the assembler creates for each label an entry in the symbol table that maps the symbolic label to the address in memory of that line of code. The exception to this rule is when a label is used with the **EQU** pseudo-op. The result of an **EQU** pseudo-op is to place an entry in the symbol table mapping the symbolic label with the value of the operand.

1.5.2. Addressing modes[edit]

A fundamental issue in software design is the differentiation between data and addresses. Another name for address is pointer. It is in assembly language programming in general and **addressing modes** in specific that this differentiation becomes clear. When we put the number 1000 into Register R0, whether this is data or address depends on how the 1000 is used.

The addressing mode is the format the instruction uses to specify the memory location to read or write data. We will see five addressing modes in this class:

Immediate	Data within the instruction
Indexed	Data pointed to by register
Indexed with offset	Data pointed to by register
PC-relative	Location is offset relative to PC
Register-list	List of registers

No addressing mode: Some instructions operate completely within the processor and require no memory data fetches. For example, the **ADD R1,R2,R3**instruction performs R2+R3 and stores the sum into R1.

Immediate addressing mode: If the data is found in the instruction itself, like **MOV R0,#1**, the instruction uses immediate addressing mode.

Indexed addressing mode: A register that contains the address or location of data is called a **pointer** or **index** register. Indexed addressing mode uses a register pointer to access memory. There are many variations of indexed addressing. In this class, you will use two types of indexed addressing. The form **[Rx]** uses Register **Rx** as a pointer, where **Rx** is any of the Registers from **R0** to **R12**. The second type you will need is called indexed with offset, which has the form **[Rx,#n]**, where **n** is a number from -255 to 4095. This addressing mode will access memory at **Rx+n**, without modifying **Rx**.

PC-relative addressing mode: The addressing mode that uses the PC as the pointer is called PC-relative addressing mode. It is used for branching, for calling functions, and accessing constant data stored in ROM. The addressing mode is called PC-relative because the machine code contains the address difference between where the program is now and the address to which the program will access.

There are many more addressing modes, but for now, these few addressing modes, as illustrated below, are enough to get us started.

1.5.3. List of twelve instructions[edit]

We will only need 12 assembly instructions in order to design our own real-time operating system. The following lists the load and store instructions we will need.

```
LDR Rd, [Rn]     ; load 32-bit memory at [Rn] to Rd
STR Rt, [Rn]     ; store Rt to 32-bit memory at [Rn]
LDR Rd, [Rn, #n] ; load 32-bit memory at [Rn+n] to Rd
STR Rt, [Rn, #n] ; store Rt to 32-bit memory at [Rn+n]
```

Let M be the 32-bit value specified by the 12-bit constant #imm12. When Rd is absent for add and subtract, the result is placed back in Rn. The following lists a few more instructions we will need.

```
MOV   Rd, Rn          ;Rd = Rn
MOV   Rd, #imm12      ;Rd = M
ADD   Rd, Rn, Rm      ;Rd = Rn + Rm
ADD   Rd, Rn, #imm12 ;Rd = Rn + M
SUB   Rd, Rn, Rm      ;Rd = Rn - Rm
SUB   Rd, Rn, #imm12 ;Rd = Rn - M
CPSID I               ;disable interrupts, I=1
CPSIE I               ;enable interrupts, I=0
```

Normally the computer executes one instruction after another in a linear fashion. In particular, the next instruction to execute is typically found immediately following the current instruction. We use branch instructions to deviate from this straight line path. These branches use PC-relative addressing.

```
B  label ;branch to label
BX Rm    ;branch indirect to location specified by Rm
BL label ;branch to subroutine at label
```

These are the push and pop instructions we will need

```
PUSH {Rn,Rm} ; push Rn and Rm onto the stack
PUSH {Rn-Rm} ; push all registers from Rn to Rm onto the stack
POP  {Rn,Rm} ; pop two 32-bit numbers off stack into Rn, Rm
POP  {Rn-Rm} ; pop multiple 32-bit numbers off stack to Rn through Rm
```

When pushing and popping multiple registers, it does not matter the order specified in the instruction. Rather, the registers are stored in memory such that the register with the smaller number is stored at the address with a smaller value. For example, consider the execution of PUSH {R1,R4-R6}. Assume the registers R1, R4, R5, and R6 initially contain the values 1, 4, 5, and 6 respectively. Figure 1.26 shows the value from lowest-numbered R1 is positioned at the lowest stack address. If four entries are popped with the POP {R0,R2,R7,R9} instruction, the value from the lowest stack address is loaded into the lowest-numbered R0.

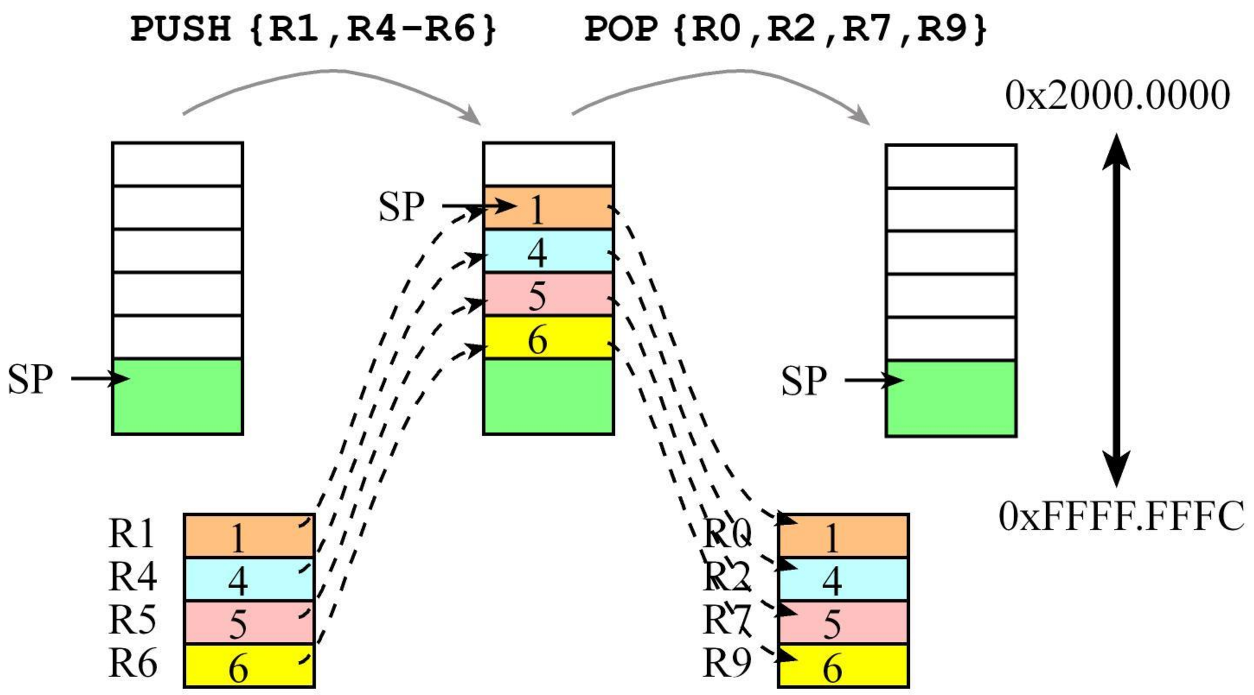


Figure 1.26. Stack drawings showing how multiple registered are pushed and popped.

1.5.4. Accessing memory[edit]

One of the basic operations we must perform is reading and writing global variables. Since all calculations are performed in registers, we must first bring the value into a register, modify the register value, and then store the new value back into memory. Consider a simple operation of incrementing a global variable in both C and assembly language. Variables can exist anywhere in RAM, however for this illustration assume the variable count is located in memory at 0x20000100. The first **LDR** instruction gets a pointer to the variable in R0 as illustrated in Figure 1.27. This means R0 will have the value 0x20000100. This value is a pointer to the variable count. The way it actually works is the assembler places a constant 0x20000100 in code space and translates the **=count** into the correct PC-relative access to the constant (e.g., **LDR R0,[PC,#28]**). The second **LDR** dereferences the pointer to fetch the value of the variable into R1. More specifically, the second **LDR** will read the 32-bit contents at 0x20000100 and put it in R1. The **ADD** instruction increments the value, and the **STR** instruction writes the new value back into the global variable. More specifically, the **STR** instruction will store the 32-bit value from R1 into at memory at 0x20000100. The following assembly implements the C code **count = count+1;**

```
  LDR R0,=count
  LDR R1,[R0] ;value of count
  ADD R1,#1
  STR R1,[R0] ;store new value
```

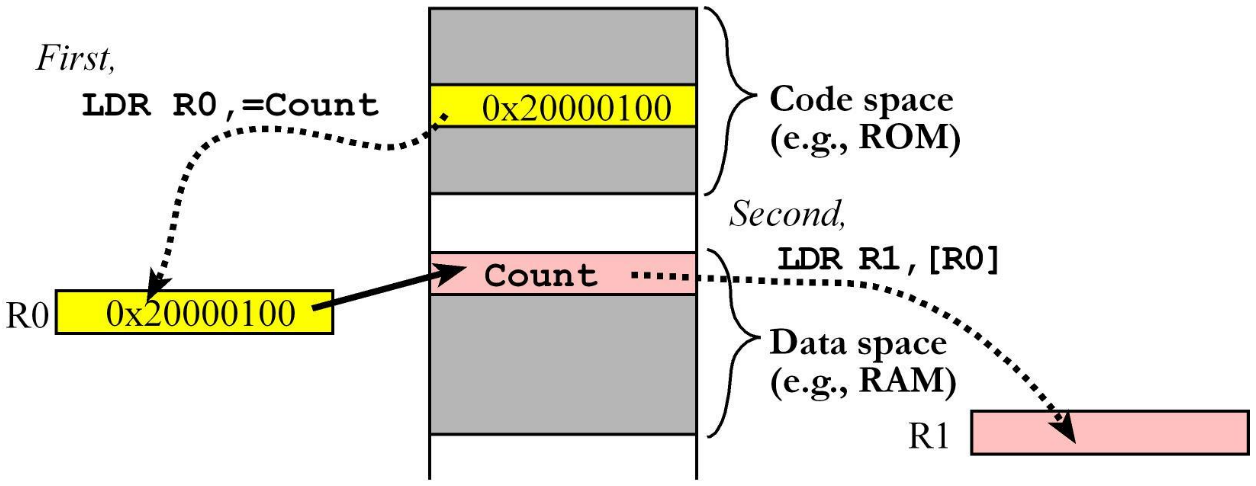


Figure 1.27. Indexed addressing using R0 as a register pointer to access memory. Data is moved into R1. Code space is where we place programs, and data space is where we place variables. The dotted arrows in this figure represent the motion of information, and the solid arrow is a pointer.

Let's work through code similar to what we will use in Chapter 2 as part of our operating system. The above example used indexed addressing with an implicit offset of 0. However, you will also need to understand indexed addressing with an explicit offset. In this example, assume **RunPt** points to a **linked list** as shown in Figure 1.28. A node of the **list** is a structure (struct in C) with multiple entries of different types. A linked list is a set of nodes where one of the entries of the node is a pointer or link to another node of the same type. In this example, the second entry of the list is a pointer to the next node in the list. Figure 1.28 shows three of many nodes that are strung together in a sequence defined by their pointers.

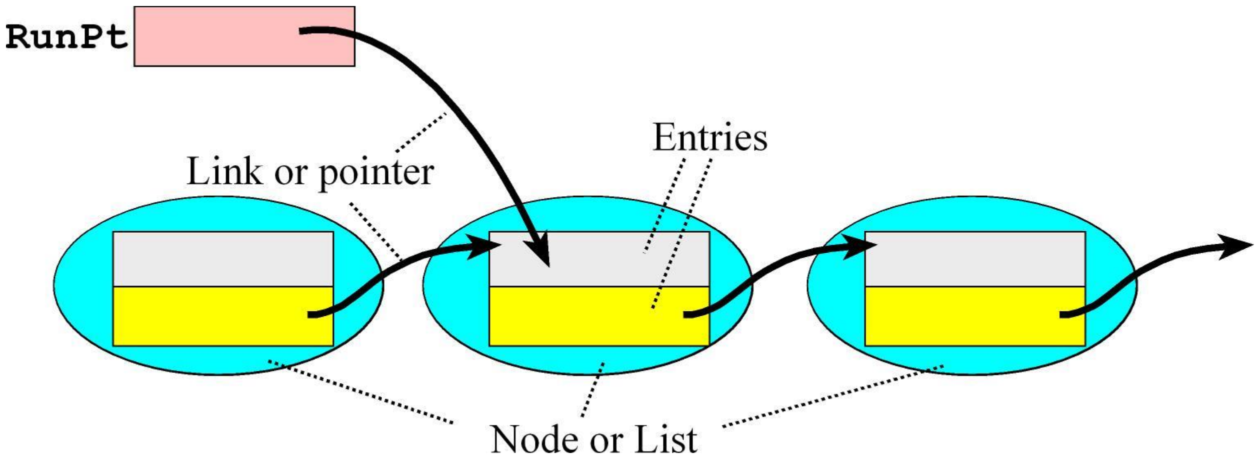


Figure 1.28. A linked list where the second entry is a pointer to the next node. Arrows are pointers or links, and dotted lines are used to label components in the figure.

As our operating system runs it will need to traverse the list. RunPt will always points to a node in the list. However, we may wish to change it to point to the next node in the list. In C, we would execute RunPt=RunPt->next; However, in assembly this translates to

```
    LDR   R1,=RunPt    ; R1 points to variable RunPt, using PC-relative
    LDR   R0,[R1]      ; R0= value of variable RunPt
    LDR   R2,[R0,#4]   ; next entry
    STR   R2,[R1]      ; update RunPt
```

Figure 1.29 draws the action caused by above the four instructions. Assume initially **RunPt** points to the middle node of the list. Each entry of the node is 32 bits or four bytes of memory. The first two instructions read the value of **RunPt** into **R0**. Since **RunPt** points to the middle node in the linked list in this figure, **R0** will also point to this node. Since each entry is 4 bytes, **R0+4** points to the second entry, which is the next pointer. The instruction **LDR R2,[R0,#4]** will read the 32-bit value pointed to by **R0+4** and place it in R2. Even though the memory address is calculated as **R0+4**, the Register **R0** itself is not modified by this instruction. **R2** now points to the right-most node in the list. The last instruction updates **RunPt** so it now points to the right-most node shown in the Figure 1.29.

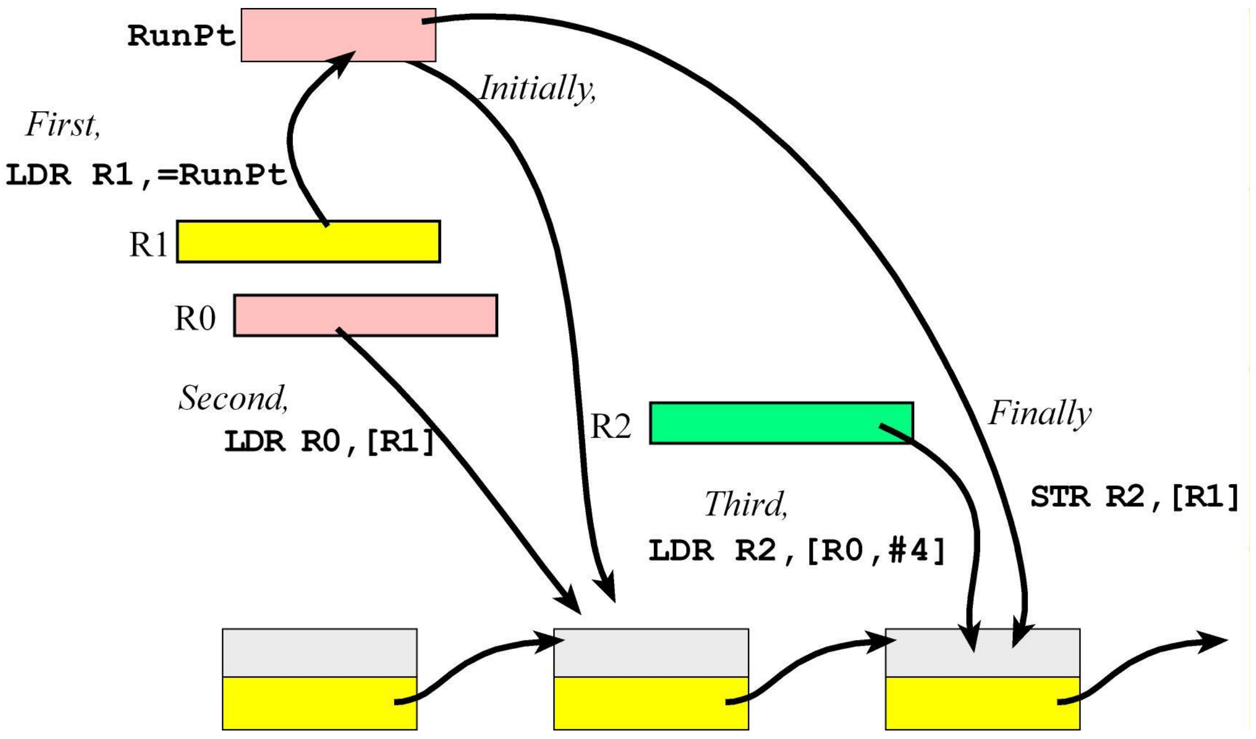


Figure 1.29. An example of indexed addressing mode with offset, data is in memory. Arrows in this figure represent pointers (not the motion of information).

Play Video

A really important concept. We use the **LDR** instruction to load data from RAM to a register and the **STR** instruction to store data from a register to RAM. In real life, when we *move* a box to the basement, *push* a broom across the floor, *load* bags into the trunk, *store* spoons in a drawer, *pop* a candy into your mouth, or *transfer* employees to a new location, there is a physical object and the action changes the location of that object. Assembly language uses these same verbs, but the action will be different. In most cases, the processor creates a copy of the data and places the copy at the new location. In other words, since the original data still exists in the previous location, there are now two copies of the information. The exception to this memory-access-creates-two-copies-rule is a stack pop. When we pop data from the stack, it no longer exists on the stack leaving us just one copy. Having the information in two places will create a very tricky problem that our operating system must handle.

Let's revisit the simple example of incrementing a global variable. In C, the code would be **count=count+1;** In assembly, the compiler creates code like this:

```
   LDR R0,=count
   LDR R1,[R0]    ;value of count
;two copies of count: in memory and in R1
   ADD R1,#1
;two copies of count with different values
   STR R1,[R0]    ;store new value
```

The instruction **LDR R1,[R0]** loads the contents of the variable count into **R1**. At this point, there are two copies of the data, the original in RAM and the copy in **R1**. After the **ADD** instruction, the two copies have different values. When designing an operating system, we will take special care to handle shared information stored in global RAM, making sure we access the proper copy. In Chapter 2, we will discuss in detail the concept of **race conditions**and **critical sections**. These very important problems arise from the problem generated by this concept of having multiple copies of information.

1.5.5. Functions[edit]

Subroutines, procedures, and functions are programs that can be called to perform specific tasks. They are important conceptual tools because they allow us to develop modular software. The programming languages Pascal, Fortran, and Ada distinguish between functions, which return values, and procedures, which do not. On the other hand, the programming languages C, C++, Java, and Lisp do not make this distinction and treat functions and procedures as synonymous. Object-oriented programming languages use the term method to describe functions that are part of classes; Objects being instantiation of classes. In assembly language, we use the term subroutine for all subprograms whether or not they return a value. Modular programming allows us to build complex systems using simple components. In this section we present a short introduction on the syntax for defining assembly subroutines. We define a subroutine by giving it a name in the label field, followed by instructions, which when executed, perform the desired effect. The last instruction in a subroutine will be BX LR, which we use to return from the subroutine.

The function in Program 1.1 and Figure 1.30 will increment the global variable count. The AREA DATA directive specifies the following lines are placed in data space (typically RAM). The SPACE 4 pseudo-op allocates 4 uninitialized bytes. The AREA CODE directive specifies the following lines are placed in code space (typically ROM). The |.text| connects this program to the C code generated by the compiler. ALIGN=2 will force the machine code to be halfword-aligned as required.

In assembly language, we will use the BL instruction to call this subroutine. At run time, the BL instruction will save the return address in the LR register. The return address is the location of the instruction immediately after the BL instruction. At the end of the subroutine, the BX LR instruction will get the return address from the LR register, returning the program to the place from which the subroutine was called. More precisely, it returns to the instruction immediately after the instruction that performed the subroutine call. The comments specify the order of execution. The while-loop causes instructions 4–10 to be repeated over and over.

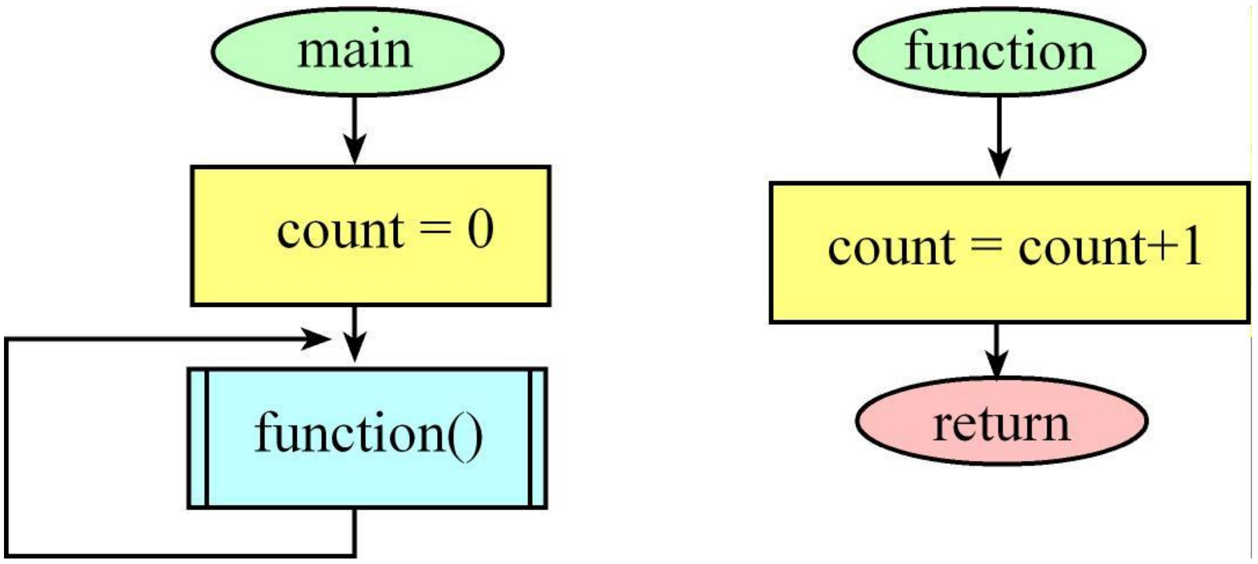


Figure 1.30. A flowchart of a simple function that adds 1 to a global variable.

```
    AREA DATA
count SPACE 4       ; 32-bit data
    AREA |.text|,CODE,READONLY,ALIGN=2
function
    LDR R0,=count  ;5
    LDR R1,[R0]    ;6 value of count
    ADD R1,#1      ;7
    STR R1,[R0]    ;8 store new value
    BX LR ;9
Start LDR R0,=count  ;1
    MOV R1,#0      ;2
    STR R1,[R0]    ;3 store new value
```

```
ui

vo
 c
}

int
 c
 w
```

```
loop  BL function    ;4                                                              }
      B loop         ;10                                                           }
```

Program 1.1. Assembly and C versions that initialize a global array of ten elements. The numbers illustrate the execution sequence.

While using a register (**LR**) to store the return address is very effective, it does pose a problem if one function were to call a second function. In Program 1.2 **someother** calls **function**. Because the return address is saved in the LR, if one function calls another function it must save the **LR** before calling and restore the **LR** after the call. In Program 1.2, the saving and restoring is performed by the **PUSH** and **POP** instructions.

```
function                                     void function(void){
; .......                                      // .......
; .......                                      // .......
    BX  LR                                   }
someother                                    void someother(void){
; .......                                      // .......
    PUSH {R4,LR}                               function();
    BL   function                              // .......
    POP  {R4,LR}                             }
; .......
    BX  LR
```

Program 1.2. Assembly and C versions that define a simple function.

1.5.6. ARM Cortex Microcontroller Software Interface Standard[edit]

Play Video

The **ARM Architecture Procedure Call Standard**, AAPCS, part of the ARM **Application Binary Interface** (ABI), uses registers R0, R1, R2, and R3 to pass input parameters into a C function. Functions must preserve the values of registers R4–R11. Also according to AAPCS we place the return parameter in Register R0. AAPCS requires we push and pop an even number of registers to maintain an 8-byte alignment on the stack. In this book, the SP will always be the main stack pointer (MSP), not the Process Stack Pointer (PSP). Recall that all object code is halfword aligned, meaning bit 0 of the PC is always clear. When the **BL** instruction is executed, bits 31–1 of register LR are loaded with the address of the instruction after the **BL**, and bit 0 is set to one. When the **BX LR** instruction is executed, bits 31–1 of register LR are put back into the PC, and bit 0 of LR goes into the T bit. On the ARM Cortex-M processor, the T bit should always be 1, meaning the processor is always in the Thumb state. Normally, the proper value of bit 0 is assigned automatically.

ARM’s Cortex Microcontroller Software Interface Standard (CMSIS) is a standardized hardware abstraction layer for the Cortex-M processor series. The purpose of the CMSIS initiative is to standardize a fragmented industry on one superior hardware and software microcontroller architecture.

The CMSIS enables consistent and simple software interfaces to the processor and core MCU peripherals for silicon vendors and middleware providers, simplifying software re-use, reducing the learning curve for new microcontroller developers, and reducing the time to market for new devices. Learn more about CMSIS directly from ARM at www.onarm.com.

The CMSIS is defined in close cooperation with various silicon and software vendors and provides a common approach to interface to peripherals, real-time operating systems, and middleware components. The CMSIS is intended to enable the combination of software components from multiple middleware vendors. The CMSIS components are:

CMSIS-CORE: API for the Cortex-M processor core and peripherals. It provides at standardized interface for Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300. Included are also SIMD intrinsic functions for Cortex-M4 SIMD instructions.

CMSIS-DSP: DSP Library Collection with over 60 Functions for various data types: fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit). The library is available for Cortex-M0, Cortex-M3, and Cortex-M4. The Cortex-M4 implementation is optimized for the SIMD instruction set.

CMSIS-RTOS API: Common API for Real-Time operating systems. It provides a standardized programming interface that is portable to many RTOS and enables software templates, middleware, libraries, and other components that can work across supported RTOS systems.

CMSIS-SVD: System View Description for Peripherals. Describes the peripherals of a device in an XML file and can be used to create peripheral awareness in debuggers or header files with peripheral register and interrupt definitions.

1.7.1. Pointers[edit]

Play Video

At the assembly level, we implement **pointers** using indexed addressing mode. For example, a register contains an address, and the instruction reads or writes memory specified by that address. Basically, we place the address into a register, then use indexed addressing mode to access the data. In this case, the register holds the pointer. Figure 1.32 illustrates three examples that utilize pointers. In this figure, **Pt SP GetPt PutPt** are pointers, where the arrows show to where they point, and the shaded boxes represent data. An **array** or **string** is a simple structure containing multiple equal-sized elements. We set a pointer to the address of the first element, then use indexed addressing mode to access the elements inside. We have introduced the stack previously, and it is an important component of an operating system. The stack pointer (SP) points to the top element on the stack. A **linked list**contains some elements that are pointers themselves. The pointers are used to traverse the data structure. Linked lists will be used in Chapter 2 to maintain the states of threads in our RTOS. The first in first out (FIFO) queue is an important data structure for I/O programming because it allows us to pass data from one module to another. One module puts data into the FIFO and another module gets data out of the FIFO. There is a **GetPt** that points to the oldest data (to be removed next) and a **PutPt** that points to an empty space (location to be stored into next). The FIFO queue will be presented in detail in Chapter 3.

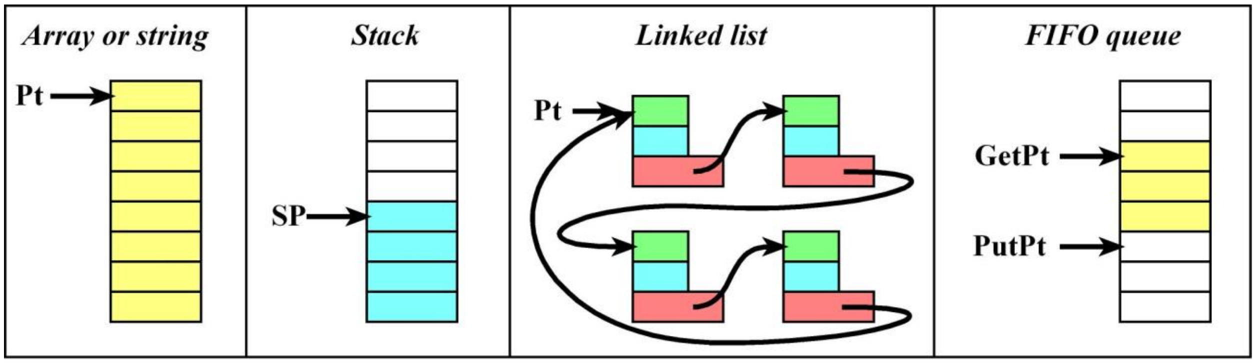


Figure 1.32. Examples of data structures that utilize pointers.

We will illustrate the use of pointers with some simple examples. Consider that we have a global variable called **Count**. This creates a 32-bit space in memory to contain the value of this variable. The int declaration means “is a signed 32-bit integer”.

```
int Count;
```

There are three phases to using pointers: creation, initialization, usage. To create a pointer, we define a variable placing the ***** before its name. As a convention, we will use "p", "pt", or "ptr" in the variable name to signify it is a pointer variable. The * means "is a pointer to". Therefore, **int *** means “is a pointer to a signed 32-bit integer”.

```
int *cPt;
```

To initialize a pointer, we must set it to point to something. Whenever we make an assignment in C, the type of the value must match the type of the variable. The following executable code, makes **cPt** point to **Count**. We see the type of **Count** is signed 32-bit integer, so the type of **&Count** is a pointer to a signed 32-bit integer.

```
cPt = &Count;
```

Assume we have another variable called x, and assume the value of Count is 42. Using the pointer is called dereferencing. If we place a *cPt inside an expression, then ***cPt** is replaced with the value at that address. So this operation will set x equal to 42.

```
x = *cPt;
```

If we place a ***cPt** as the assignment, then the value of the expression is stored into the memory at the address of the pointer. So, this operation will set Count equal to 5;

```
   *cPt = 5;
```

We can use the dereferencing operator in both the expression and as the assignment. These operations will increment Count.

```
   *cPt = *cPt + 1;
   *cPt += 1;
   (*cPt)++;
```

This operation will not increment Count. Rather, it fetches Count and increments the pointer.

```
   *cPt++;
```

Functions that require data to passed by the value they hold are said to use call-by-value parameter passing. With an input parameter using call by value, the data itself is passed into the function. For an output parameter using return by value, the result of the function is a value, and the value itself is returned. According to AAPCS, the first four input parameters are passed in R0 to R3 and the output parameter is returned in R0. Alternatively, if you pass a pointer to the data, rather than the data itself, we will be able to pass large amounts of data. Passing a pointer to data is classified as call-by-reference. For large amounts of data, call by reference is faster, because the data need not be copied from calling program to the called subroutine. In call by reference, the one copy of the data exists in the calling program, and a pointer to it is passed to the subroutine. In this way, the subroutine actually performs read/write access to the original data. Call by reference is also a convenient mechanism to return data as well. Passing a pointer to an object allows this object (a primitive data type like char, int, or a collection like an array, or a composite struct data type) to be an input parameter and an output parameter. Our real-time operating system will make heavy use of pointers. In this example, the function is allowed to read and write the original data:

```
void Increment(int *cpt){
   (*cpt)= (*cpt)+1; // read, modify, write to original data
}
```

We will also use pointers for arrays, linked-lists, stacks, and first-in-first-out queues. If your facility with pointers is weak, we suggest you review pointers. Chapter 7 of the following ebook teaches pointers and their usage in Valvano Embedded Systems Ebook.

1.7.2. Arrays[edit]

Play Video

Figure 1.33 shows an array of the first ten prime numbers stored as 32-bit integers, we could allocate the structure in ROM using

```
  int const Primes[10]={1,2,3,5,7,11,13,17,19,23};
```

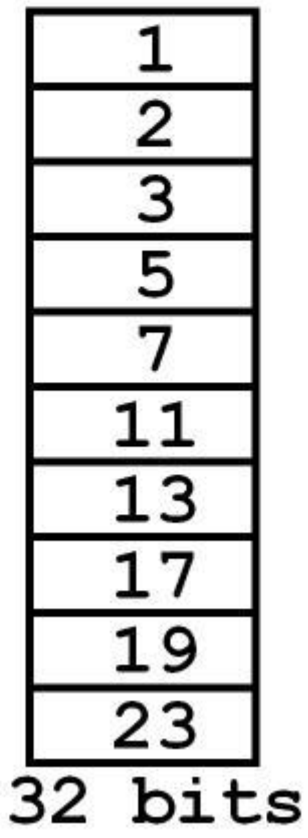


Figure 1.33. Array of ten 32-bit values.

By convention, we define Primes[0] as the first element, Primes[1] as the second element, etc. The address of the first element can be written as **&Primes[0]** or just **Primes**. In C, if we want the 5th element, we use the expression **Primes[4]** to fetch the 7 out of the structure. In C the following two expressions are equivalent, both of which will fetch the contents from the 5th element.

```
  Primes[4]
  *(Primes+4)
```

In C, we define a pointer to a signed 32-bit constant as

```
int const *Cpt;
```

In this case, the const does not indicate the pointer is fixed. Rather, the pointer refers to constant 16-bit data in ROM. We initialize the pointer at run time using

```
  Cpt = Primes;    // Cpt points to Primes
```
or
```
 Cpt = &Primes[0]; // Cpt points to Primes
```

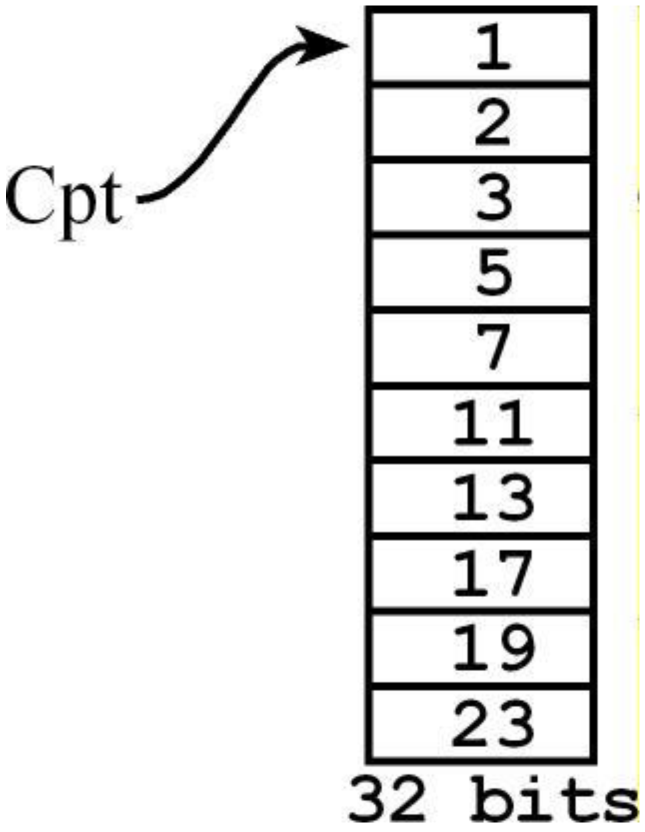


Figure 1.34. Cpt is a pointer to an array of ten 32-bit values.

When traversing an array, we often wish to increment the pointer to the next element. To move the pointer to the next element, we use the expression**Cpt++**. In C, **Cpt++**, which is the same thing as **Cpt = Cpt+1;** actually adds four to the pointer because it points to 32-bit words. If the array contained 8-bit data, incrementing the pointer would add 1. If the array contained 16-bit data, incrementing the pointer adds 2. The pointers themselves are always 32-bits on the ARM, but the data could be 1, 2, 4, 8 ... bytes.

As an example, consider the situation where we wish to pass a large amount of data into the function **BubbleSort**. In this case, we have one or more buffers, defined in RAM, which initially contains data in an unsorted fashion. The buffers shown here are uninitialized, but assume previously executed software has filled these buffers with corresponding voltage and pressure data. In C, we could have

```
uint8_t VBuffer[100]; // voltage data
uint8_t PBuffer[200]; // pressure data
```

Since the size of these buffers is more than will fit in the registers, we will use call by reference. In C, to declare a parameter call by reference we use the *.

```
void BubbleSort(uint8_t *pt, uint32_t size){
   uint32_t i,j; uint8_t data,*p1,*p2;
   for(i=1; i<size; i++){
     p1 = pt; // pointer to beginning
     for(j=0; j<size-i; j++){
       p2 = p1+1; // p2 points to the element after p1
       if(*p1 > *p2){
         data = *p1; // swap
         *p1 = *p2;
         *p2 = data;
       }
      p1++;
     }
   }
}
```

To invoke a function using call by reference we pass a pointer to the object. These two calling sequences are identical, because in C the array name is equivalent to a pointer to its first element (**VBuffer** ≡ **VBuffer[0]**). Recall that the & operator is used to get the address of a variable.

```
void main(void){
BubbleSort(Vbuffer,100);
BubbleSort(Pbuffer,200);
}
```

```
void main(void){
BubbleSort(&VBuffer[0],100);
BubbleSort(&PBuffer[0],200);
}
```

One advantage of call by reference in this example is the same buffer can be used also as the return parameter. In particular, this sort routine re-arranges the data in the same original buffer.

Since RAM is a scarce commodity on most microcontrollers, not having to allocate two buffers will reduce RAM requirements for the system.

From a security perspective, call by reference is more vulnerable than call by value. If we have important information, then a level of trust is required to pass a pointer to the original data to a subroutine. Since call by value creates a copy of the data at the time of the call, it is slower but more secure. With call by value, the original data is protected from subroutines that are called.

1.7.3. Linked lists[edit]

Play Video

Linked lists are an important data structure used in operating systems. Each element (node) contains data and a pointer to another element as shown in Figure 1.35. Given that a node in the list is a composite of data and a pointer, we use **struct** to declare a composite data type. A composite data type can be made up of primitive data type, pointers and also other composite data-types.

```
struct Node{
  struct Node *Next;
  int Data;
};
typedef struct Node NodeType;
```

In this simple example, the **Data** field is just a 32-bit number, we will expand our node to contain multiple data fields each storing a specific attribute of the node. There is a pointer to the first element, called the head pointer. The last element in the list has a **null** pointer in its next field to indicate the end of the list.

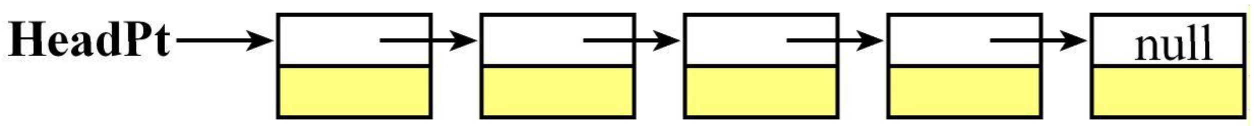


Figure 1.35. A linked list with 5 nodes.

We can create lists statically or dynamically. A statically created list is created at compile time and does not change during the execution of the program.

```
NodeType theList[8] ={
  {&theList[1], 1},
  {&theList[2], 10},
  {&theList[3], 100},
  {&theList[4], 1000},
  {&theList[5], 10000},
  {&theList[6], 100000},
  {&theList[7], 1000000},
  {0, 10000000}};
NodeType *HeadPt = theList; // points to first element
```

The following function searches the list to see if a data value exists in the list.

```
int Search(int x){ NodeType *pt;
  pt = HeadPt; // start at beginning
  while(pt){
    if(pt->Data == x) return 1; // found
    pt = pt->Next;
  }
  return 0; // not found
}
```

This example created the linked-list statically. The compiler will generate code prior to running main (called premain) that will initialize the eight nodes. To do this initialization, there will be two copies of the structure: the initial copy in ROM used during premain, and the RAM copy used by the program during execution. If the program needs to change this structure during execution then having two copies is fine. Lab 2 will be implemented in this manner. However, if the program does not change the structure, then you could put a single copy in ROM by adding **const** to the definition. In this case, **HeadPt** will be in RAM but the linked list will be in ROM.

```
const struct Node{
  const struct Node *Next;
  int Data;
};
typedef const struct Node NodeType;
NodeType theList[8] ={
  {&theList[1], 1},
  {&theList[2], 10},
  {&theList[3], 100},
  {&theList[4], 1000},
  {&theList[5], 10000},
  {&theList[6], 100000},
  {&theList[7], 1000000},
  {0, 10000000}};
NodeType *HeadPt = theList; // points to first element
```

It is possible to create a linked list dynamically and grow/shrink the list as a program executes. However, in keeping with our goal to design a simple RTOS, we will refrain from doing any dynamic allocation, which would require the management of a heap. Most real-time systems do not allow the heap (malloc and free) to be accessed by the application programmer, because the use of the heap could lead to **nondeterministic** behavior (the activity of one program affects the behavior of another completely unrelated program).

1.8.1. Introduction to debugging[edit]

Microcontroller-related problems often require the use of specialized equipment to debug the system hardware and software. Useful hardware tools include a logic probe, an oscilloscope, a logic analyzer, and a JTAG debugger. A **logic probe** is a handheld device with an LED or buzzer. You place the probe on your digital circuit and LED/buzzer will indicate whether the signal is high or low. An **oscilloscope**, or scope, graphically displays information about an electronic circuit, where the voltage amplitude versus time is displayed. A scope has one or two channels, with many ways to trigger or capture data. A scope is particularly useful when interfacing analog signals using an ADC or DAC. The PicoScope 2104 (from http://www.picotech.com/) is a low-cost but effective tool for debugging microcontroller circuits. A **logic analyzer** is essentially a multiple channel digital storage scope with many ways to trigger. As shown in Figure 1.36, we can connect the logic analyzer to digital signals that are part of the system, or we can connect the logic analyzer channels to unused microcontroller pins and add software to toggle those pins at strategic times/places. As a troubleshooting aid, it allows the experimenter to observe numerous digital signals at various points in time and thus make decisions based upon such observations. One problem with logic analyzers is the massive amount of information that it generates. To use an analyzer effectively one must learn proper triggering mechanisms to capture data at appropriate times eliminating the need to sift through volumes of output. The logic analyzer figures in this book were collected with a logic analyzer Digilent (from http://www.digilentinc.com/). The Analog Discovery combines a logic analyzer with an oscilloscope, creating an extremely effective debugging tool. Maintenance Tip: First, find the things that will break you. Second, break them. Common error: Sometimes the original system operates properly, and the debugging code has mistakes.

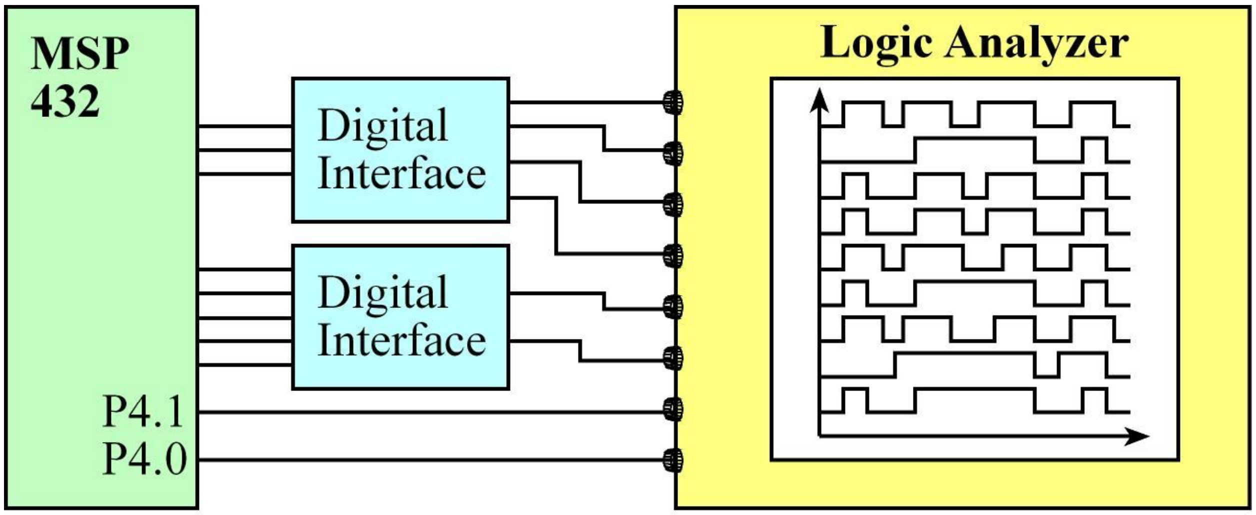
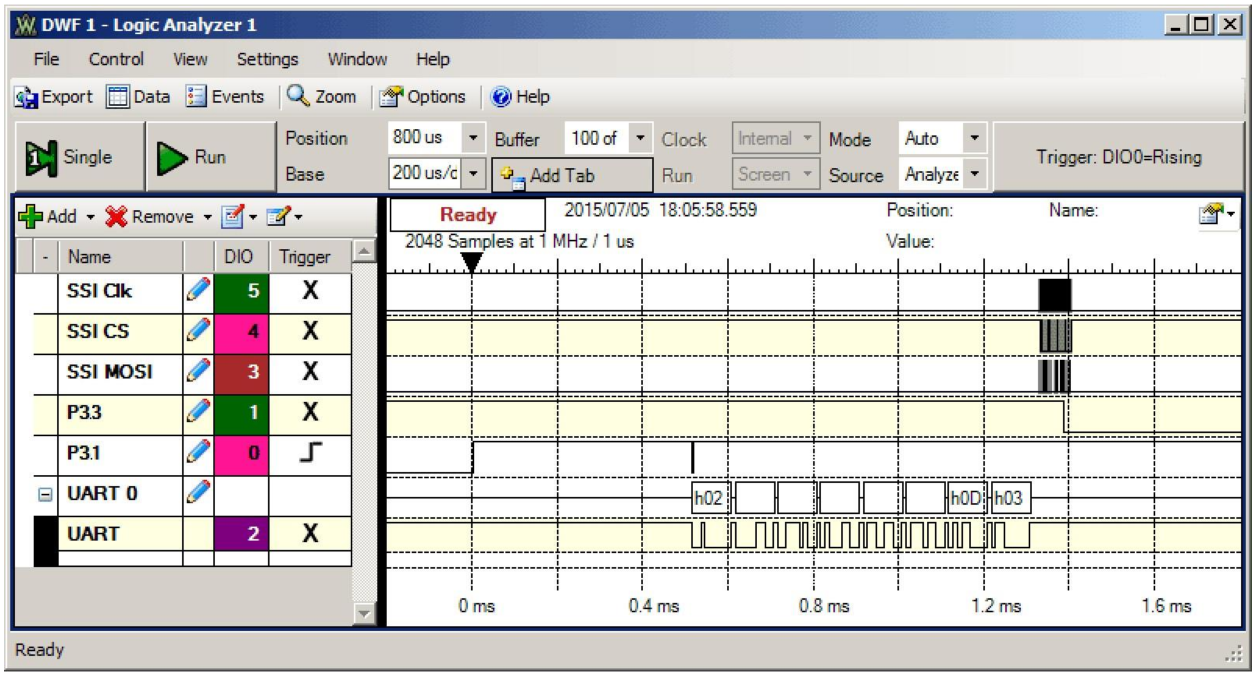


Figure 1.36. A logic analyzer and example output. P4.1 and P4.0 are extra pins just used for debugging.

Figure 1.37 shows a logic analyzer output, where signals SSI are outputs to the LCD, and UART is transmission between two microcontrollers. However P3.3 and P3.1 are debugging outputs to measuring timing relationships between software execution and digital I/O. The rising edge of P3.1 is used to trigger the data collection.



Position 800 us
Base 200 us/d
Buffer 100 of
Clock Internal
Run Screen
Mode Auto
Source Analyze
Trigger: DIO0=Rising

Ready 2015/07/05 18:05:58.559
2048 Samples at 1 MHz / 1 us
Position:
Value:
Name:

Name	DIO	Trigger
SSI Clk	5	X
SSI CS	4	X
SSI MOSI	3	X
P3.3	1	X
P3.1	0	⌠
UART 0		
UART	2	X

Ready

Figure 1.37. Analog Discovery logic analyzer output (www.digilentinc.com).

An **emulator** is a hardware debugging tool that recreates the input/output signals of the processor chip. To use an emulator, we remove the processor chip and insert the emulator cable into the chip socket. In most cases, the emulator/computer system operates at full speed. The emulator allows the programmer to observe and modify internal registers of the processor. Emulators are often integrated into a personal computer, so that its editor, hard drive, and printer are available for the debugging process.

The only disadvantage of the in-circuit emulator is its cost. To provide some of the benefits of this high-priced debugging equipment, many microcontrollers use a JTAG debugger. The JTAG hardware exists both on the microcontroller chip itself and as an external interface to a personal computer. Although not as flexible as an ICE, JTAG can provide the ability to observe software execution in real-time, the ability to set breakpoints, the ability to stop the computer, and the ability to read and write registers, I/O ports and memory.

Debugging is an essential component of embedded system design. We need to consider debugging during all phases of the design cycle. It is important to develop a structure or method when verifying system performance. This section will present a number of tools we can use when debugging. Terms such as program testing, diagnostics, performance debugging, functional debugging, tracing, profiling, instrumentation, visualization, optimization, verification, performance measurement, and execution measurement have specialized meanings, but they are also used interchangeably, and they often describe overlapping functions. For example, the terms profiling, tracing, performance measurement, or execution measurement may be used to describe the process of examining a program from a time viewpoint. But, tracing is also a term that may be used to describe the process of monitoring a program state or history for functional errors, or to describe the process of stepping through a program with a debugger. Usage of these terms among researchers and users vary.

Black-box testing is simply observing the inputs and outputs without looking inside. Black-box testing has an important place in debugging a module for its functionality. On the other hand, **white-box testing** allows you to control and observe the internal workings of a system. A common mistake made by new engineers is to just perform black box testing. Effective debugging uses both. One must always start with black-box testing by subjecting a hardware or software module to appropriate test-cases. Once we document the failed test-cases, we can use them to aid us in effectively performing the task of white-box testing.

We define a **debugging instrument** as software code that is added to the program for the purpose of debugging. A print statement is a common example of an instrument. Using the editor, we add print statements to our code that either verify proper operation or display run-time errors.

Nonintrusiveness is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by the debugging instrument itself. Let t be the time required to execute the instrument, and let Δt be the average time in between executions of the instrument. One quantitative measure of intrusiveness is t/Δt, which is the fraction of available processor time used by the debugger. For example, a print statement added to your source code may be very intrusive because it might significantly affect the real-time interaction of the hardware and software. Observing signals that already exist as part of the system with an oscilloscope or logic analyzer is **nonintrusive**, meaning the presence of the scope/analyzer has no effect on the system being measured.. A debugging instrument is classified as **minimally intrusive** if it has a negligible effect on the system being debugged. In a real microcontroller system, breakpoints and single-stepping are also intrusive, because the real hardware continues to change while the software has stopped. When a program interacts with real-time events, the performance can be significantly altered when using intrusive debugging tools. To be effective we must employ nonintrusive or minimally intrusive methods.

Although, a wide variety of program monitoring and debugging tools are available today, in practice it is found that an overwhelming majority of users either still prefer or rely mainly upon "rough and ready" manual methods for locating and correcting program errors. These methods include desk-checking, dumps, and print statements, with print statements being one of the most popular manual methods. Manual methods are useful because they are readily available, and they are relatively simple to use. But, the usefulness of manual methods is limited: they tend to be highly intrusive, and they do not provide adequate control over repeatability, event selection, or event isolation. A real-time system, where software execution timing is critical, usually cannot be debugged with simple print statements, because the print statement itself will require too much time to execute.

The first step of debugging is to **stabilize** the system. In the debugging context, we stabilize the problem by creating a test routine that fixes (or stabilizes) all the inputs. In this way, we can reproduce the exact inputs over and over again. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters.

Acceleration means we will speed up the testing process. When we are testing one module we can increase how fast the functions are called in an attempt to expose possible faults. Furthermore, since we can control the test environment, we will **vary** the test conditions over a wide range of possible conditions. **Stress testing** means we run the system beyond the requirements to see at what point it breaks down.

When a system has a small number of possible inputs (e.g., less than a million), it makes sense to test them all. When the number of possible inputs is large we need to choose a set of inputs. **Coverage** defines the subset of possible inputs selected for testing. A **corner case** is defined as a situation at the boundary where multiple inputs are at their maximum, like the corner of a 3-D cube. At the corner small changes in input may cause lots of internal and external changes. In particular, we need to test the cases we think might be difficult (e.g., the clock output increments one second from 11:59:59 PM December 31, 1999.) There are many ways to decide on the coverage. We can select values:

- Near the extremes and in the middle
- Most typical of how our clients will properly use the system
- Most typical of how our clients will improperly use the system
- That differ by one
- You know your system will find difficult
- Using a random number generator

To **stabilize** the system we define a fixed set of inputs to test, run the system on these inputs, and record the outputs. Debugging is a process of finding patterns in the differences between recorded behavior and expected results. The advantage of modular programming is that we can perform **modular debugging**. We make a list of modules that might be causing the bug. We can then create new test routines to stabilize these modules and debug them one at a time. Unfortunately, sometimes all the modules seem to work, but the combination of modules does not. In this case we study the interfaces between the modules, looking for intended and unintended (e.g., unfriendly code) interactions.

1.8.2. Functional Debugging[edit]

Debugging on TM4C123

Play Video

Functional debugging involves the verification of input/output parameters. It is a static process where inputs are supplied, the system is run, and the outputs are compared against the expected results. We will present seven methods of functional debugging.

1. Single Stepping or Trace.
2. Breakpoints without filtering.
3. Conditional breakpoints.
4. Instrumentation: print statements.
5. Instrumentation: dump into array without filtering.
6. Instrumentation: dump into array with filtering.
7. Monitor using the LED heartbeat.

We can add a debugger **instrument** that dumps strategic information into arrays at run time. Assume P1 is an input and P2 is an output port that are strategic to the system. The first step when instrumenting a dump is to define a buffer in RAM to save the debugging measurements. The **Debug_Cnt** will be used to index into the buffers. Debug_Cnt must be initialized to zero, before the debugging begins. The debugging instrument, shown in Program 1.3, saves the strategic data into the buffer. We can then observe the contents of the array at a later time. One of the advantages of dumping is that the JTAG debugging allows you to visualize memory even when the program is running.

```
#define SIZE 100
uint8_t Debug_Buffer[SIZE][2];
unsigned int Debug_Cnt=0;
void Debug_Dump(void){ // dump P1IN and P2OUT
  if(Debug_Cnt < SIZE){
    Debug_Buffer[Debug_Cnt][0] = P1IN;
    Debug_Buffer[Debug_Cnt][1] = P2OUT;
    Debug_Cnt++;
  }
}
```

Program 1.3. Instrumentation dump without filtering.

Next, you add **Debug_Dump();** statements at strategic places within the system. You can either use the debugger to display the results or add software that prints the results after the program has run and stopped. In this way, you can collect information in the exact same manner you would if you were using print statements.

One problem with dumps is that they can generate a tremendous amount of information. If you suspect a certain situation is causing the error, you can add a filter to the instrument. A filter is a software/hardware condition that must be true in order to place data into the array. In this situation, if we suspect the error occurs when the pointer nears the end of the buffer, we could add a filter that saves in the array only when data matches a certain condition. In the example shown in Program 1.4, the instrument saves the strategic variables into the buffer only when P1.7 is high.

```
#define SIZE 100
uint8_t Debug_Buffer[SIZE][2];
unsigned int Debug_Cnt=0;
void Debug_FilteredDump(void){ // dump P1IN and P2OUT
  if((P1IN&0x80)&&(Debug_Cnt < SIZE)){
    Debug_Buffer[Debug_Cnt][0] = P1IN;
    Debug_Buffer[Debug_Cnt][1] = P2OUT;
    Debug_Cnt ++;
  }
}
```

Program 1.4. Instrumentation dump with filter.

Another tool that works well for real-time applications is the monitor. A monitor is an independent output process, somewhat similar to the print statement, but one that executes much faster and thus is much less intrusive. An LCD can be an effective monitor for small amounts of information if the time between outputs is much larger than the time to output. Another popular monitor is the LED. You can place one or more LEDs on individual otherwise unused output bits. Software toggles these LEDs to let you know what parts of the program are running. An LED is an example of a Boolean monitor or heartbeat. Assume an LED is attached to Port 1 bit 0. Program 1.5 will toggle the LED.

```
#define LEDOUT (*((volatile uint8_t *)(0x42000000+32*0x4C02+4*0)))
#define Debug_HeartBeat() (LEDOUT ^= 0x01)
```

Program 1.5. An LED monitor.

Next, you add **Debug_HeartBeat();** statements at strategic places within the system. Port 1 must be initialized so that bit 0 is an output before the debugging begins. You can either observe the LED directly or look at the LED control signals with a high-speed oscilloscope or logic analyzer. When using LED monitors it is better to modify just the one bit, leaving the other 7 as is. In this way, you can have multiple monitors on one port.

Play Video

1.8.3. TExaS Logic analyzer[edit]

Play Video

Because time is an important aspect of real-time operating systems we have created means for you to observe the execution pattern of the user application. We have implemented a zero-cost logic analyzer that has three parts. When debugging a lab, you will enable logic analyzer mode by initializing Texas

```
TExaS_Init(LOGICANALYZER, 1000);
```

First, as part of the Texas.c/Texas.h component there are seven functions you will call from within the user application. Basically, one of these functions is called each time the user task performs a time-sensitive operation.

```
    TExaS_Task0
    TExaS_Task1
    TExaS_Task2
    TExaS_Task3
    TExaS_Task4
    TExaS_Task5
    TExaS_Task6
```

Inside Texas, each function performs two operations. When in grading mode, the function will record the time in microseconds in an array. These recordings will be used by the grader to verify the tasks are executed as desired. When in debugging or logic analyzer mode, the function toggles one bit in a shared global variable called LogicData. You could extern this variable and set the bottom 7 bits however you wish. Bits 6 – 0 contain data and bit 7 should remain 1.

The second part of the logic analyzer is a UART and a periodic interrupt, running along side of your code. When in debugging or logic analyzer mode, the periodic interrupt sends the 8-bit **LogicData** to the PC every 100us (10 kHz). Bit 7=1 signifies it is logic analyzer data. It is possible to use the UART to send ASCII text (with bit 7=0).

The third part of the logic analyzer is the **TExaSdisplay** application. To use the logic analyzer, you must enable the logic analyzer when calling **TExaS_Init**, and the microcontroller must be running with interrupts enabled. Within **TExaSdisplay**, you first configure the COM port settings and then you open the COM port. To run the logic analyzer, click the logic analyzer tool bar button or select logic analyzer in the view menu. Triggering can be configured in the Logic analyzer configuration dialog. Triggers can occur when a signal is high, low, rising edge, falling edge or either edge.

Some of the useful short cuts are

F1	about **TExaSdisplay**
F2	clear ASCII text display
F3	COM port settings
F4	open COM port
Shift+F4	open next COM port
F5	close COM port
F6	run slower (longer time scale)
F7	run faster (shorter time scale)
F8	pause
F9	single

Remember the timing parameters calculated by the logic analyzer are only accurate to 100 us. The grader calculates timing parameters to 1-us accuracy. The display will flicker because it collects a buffer of data and then displays it. For example, at 3.2-second window size, the application will collect 6.4 seconds of data and then update the display.

1.8.4. Profiling[edit]

Profiling is a type of performance debugging that collects the time history of program execution. Profiling measures where and when our software executes. It could also include what data is being processed. For example if we could collect the time-dependent behavior of the program counter, then we could see the execution patterns of our software.

Profiling using a software dump to study execution pattern. In this section, we will discuss software instruments that study the execution pattern of our software. In order to collect information concerning execution we will add debugging instruments that save the time and location in arrays (Program 1.6). By observing these data, we can determine both a time profile (when) and an execution profile (where) of the software execution. Running this profile revealed the sequence of places as 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, and 3. Each call to **Debug_Profile** requires 32 cycles to execute. Therefore, this instrument is a lot less intrusive than a print statement.

```
uint32_t Debug_time[20];
uint8_t Debug_place[20];
uint32_t n;
void Debug_Profile(uint8_t p){
  if(n < 20){
    Debug_time[n] = STCURRENT; // record current time
    Debug_place[n] = p;
  n++;
  }
}
uint32_t sqrt(uint32_t s){
uint32_t t; // t*t becomes s
int n; // loop counter
  Debug_Profile(0);
  t = s/10+1; // initial guess
  Debug_Profile(1);
  for(n = 16; n; --n){ // will finish
    Debug_Profile(2);
    t = ((t*t+s)/t)/2;
  }
  Debug_Profile(3);
  return t;
}
```

Program 1.6: A time/position profile dumping into a data array.

Arm Cortex M Instructions[edit]

Memory access instructions

```
LDR    Rd, [Rn]       ; load 32-bit number at [Rn] to Rd
LDR    Rd, [Rn,#off]  ; load 32-bit number at [Rn+off] to Rd
LDR    Rd, =value     ; set Rd equal to any 32-bit value (PC rel)
LDRH   Rd, [Rn]       ; load unsigned 16-bit at [Rn] to Rd
LDRH   Rd, [Rn,#off]  ; load unsigned 16-bit at [Rn+off] to Rd
LDRSH  Rd, [Rn]       ; load signed 16-bit at [Rn] to Rd
LDRSH  Rd, [Rn,#off]  ; load signed 16-bit at [Rn+off] to Rd
LDRB   Rd, [Rn]       ; load unsigned 8-bit at [Rn] to Rd
LDRB   Rd, [Rn,#off]  ; load unsigned 8-bit at [Rn+off] to Rd
LDRSB  Rd, [Rn]       ; load signed 8-bit at [Rn] to Rd
LDRSB  Rd, [Rn,#off]  ; load signed 8-bit at [Rn+off] to Rd
STR    Rt, [Rn]       ; store 32-bit Rt to [Rn]
STR    Rt, [Rn,#off]  ; store 32-bit Rt to [Rn+off]
STRH   Rt, [Rn]       ; store least sig. 16-bit Rt to [Rn]
STRH   Rt, [Rn,#off]  ; store least sig. 16-bit Rt to [Rn+off]
STRB   Rt, [Rn]       ; store least sig. 8-bit Rt to [Rn]
STRB   Rt, [Rn,#off]  ; store least sig. 8-bit Rt to [Rn+off]
PUSH   {Rt}           ; push 32-bit Rt onto stack
POP    {Rd}           ; pop 32-bit number from stack into Rd
ADR    Rd, label      ; set Rd equal to the address at label
MOV{S} Rd, <op2>      ; set Rd equal to op2
MOV    Rd, #im16      ; set Rd equal to im16, im16 is 0 to 65535
MVN{S} Rd, <op2>      ; set Rd equal to -op2
```

Branch instructions

```
B    label  ; branch to label    Always
BEQ  label  ; branch if Z == 1   Equal
BNE  label  ; branch if Z == 0   Not equal
BCS  label  ; branch if C == 1   Higher or same, unsigned ≥
BHS  label  ; branch if C == 1   Higher or same, unsigned ≥
BCC  label  ; branch if C == 0   Lower, unsigned
BLO  label  ; branch if C == 0   Lower, unsigned <
BMI  label  ; branch if N == 1   Negative
BPL  label  ; branch if N == 0   Positive or zero
BVS  label  ; branch if V == 1   Overflow
BVC  label  ; branch if V == 0   No overflow
BHI  label  ; branch if C==1 and Z==0  Higher, unsigned >
BLS  label  ; branch if C==0 or  Z==1  Lower or same, unsigned ≤
BGE  label  ; branch if N == V   Greater than or equal, signed ≥
BLT  label  ; branch if N != V   Less than, signed <
BGT  label  ; branch if Z==0 and N==V  Greater than, signed >
BLE  label  ; branch if Z==1 or N!=V  Less than or equal, signed ≤
BX   Rm     ; branch indirect to location specified by Rm
BL   label  ; branch to subroutine at label
BLX  Rm     ; branch to subroutine indirect specified by Rm
```

Interrupt instructions

```
CPSIE  I              ; enable interrupts  (I=0)
CPSID  I              ; disable interrupts (I=1)
```

Logical instructions

```
AND{S} {Rd,} Rn, <op2> ; Rd=Rn&op2    (op2 is 32 bits)
ORR{S} {Rd,} Rn, <op2> ; Rd=Rn|op2    (op2 is 32 bits)
EOR{S} {Rd,} Rn, <op2> ; Rd=Rn^op2    (op2 is 32 bits)
BIC{S} {Rd,} Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)
ORN{S} {Rd,} Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)
LSR{S} Rd, Rm, Rs      ; logical shift right Rd=Rm>>Rs  (unsigned)
LSR{S} Rd, Rm, #n      ; logical shift right Rd=Rm>>n   (unsigned)
ASR{S} Rd, Rm, Rs      ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S} Rd, Rm, #n      ; arithmetic shift right Rd=Rm>>n  (signed)
LSL{S} Rd, Rm, Rs      ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n      ; shift left Rd=Rm<<n  (signed, unsigned)
```

Arithmetic instructions

```
ADD{S} {Rd,} Rn, <op2> ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12 ; Rd = Rn + im12, im12 is 0 to 4095
SUB{S} {Rd,} Rn, <op2> ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12 ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2> ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12 ; Rd = im12 – Rn
CMP    Rn, <op2>       ; Rn – op2      sets the NZVC bits
CMN    Rn, <op2>       ; Rn - (-op2)   sets the NZVC bits
MUL{S} {Rd,} Rn, Rm    ; Rd = Rn * Rm       signed or unsigned
MLA    Rd, Rn, Rm, Ra  ; Rd = Ra + Rn*Rm    signed or unsigned
MLS    Rd, Rn, Rm, Ra  ; Rd = Ra - Rn*Rm    signed or unsigned
UDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         unsigned
SDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         signed
```

Notes Ra Rd Rm Rn Rt represent 32-bit registers

value	**any 32-bit value: signed, unsigned, or address**
{S}	**if S is present, instruction will set condition codes**
#im12	**any value from 0 to 4095**
#im16	**any value from 0 to 65535**
{Rd,}	**if Rd is present Rd is destination, otherwise Rn**
#n	**any value from 0 to 31**
#off	**any value from -255 to 4095**
label	**any address within the ROM of the microcontroller**
op2	**the value generated by <op2>**

Examples of flexible operand <op2> creating the 32-bit number. E.g., Rd = Rn+op2

ADD Rd, Rn, Rm ; op2 = Rm
ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned
ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned
ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed
ADD Rd, Rn, #constant ; op2 = constant,

where X and Y are hexadecimal digits: produced by shifting an 8-bit unsigned value left by any number of bits

in the form 0x00XY00XY
in the form 0xXY00XY00
in the form 0xXYXYXYXY