7.1 What is an IoT Device

As described earlier, a "Thing" in Internet of Things (IoT) can be any object that has a unique identifier and which can send/receive data (including user data) over a network (e.g., smart phone, smart TV, computer, refrigerator, car, etc.). IoT devices are connected to the Internet and send information about themselves or about their surroundings (e.g. information sensed by the connected sensors) over a network (to other devices or servers/storage) or allow actuation upon the physical entities/environment around them remotely. Some examples of IoT devices are listed below:

- A home automation device that allows remotely monitoring the status of appliances and controlling the appliances.
- An industrial machine which sends information abouts its operation and health monitoring data to a server.
- A car which sends information about its location to a cloud-based service.
- A wireless-enabled wearable device that measures data about a person such as the number of steps walked and sends the data to a cloud-based service.

7.1.1 Basic building blocks of an IoT Device

An IoT device can consist of a number of modules based on functional attributes, such as:

- Sensing: Sensors can be either on-board the IoT device or attached to the device. IoT device can collect various types of information from the on-board or attached sensors such as temperature, humidity, light intensity, etc. The sensed information can be communicated either to other devices or cloud-based servers/storage.
- Actuation: IoT devices can have various types of actuators attached that allow taking actions upon the physical entities in the vicinity of the device. For example, a relay switch connected to an IoT device can turn an appliance on/off based on the commands sent to the device.
- Communication: Communication modules are responsible for sending collected data to other devices or cloud-based servers/storage and receiving data from other devices and commands from remote applications.
- Analysis & Processing: Analysis and processing modules are responsible for making sense of the collected data.

The representative IoT device used for the examples in this book is the widely used single-board mini computer called Raspberry Pi (explained in later sections). The use of Raspberry Pi is intentional since these devices are widely accessible, inexpensive, and available from multiple vendors. Furthermore, extensive information is available on their programming and use both on the Internet and in other textbooks. The principles we teach in this book are just as applicable to other (including proprietary) IoT endpoints, in addition to Raspberry Pi. Before we look at the specifics of Raspberry Pi, let us first look at the building blocks of a generic single-board computer (SBC) based IoT device.

Figure 7.1 shows a generic block diagram of a single-board computer (SBC) based IoT device that includes CPU, GPU, RAM, storage and various types of interfaces and peripherals.

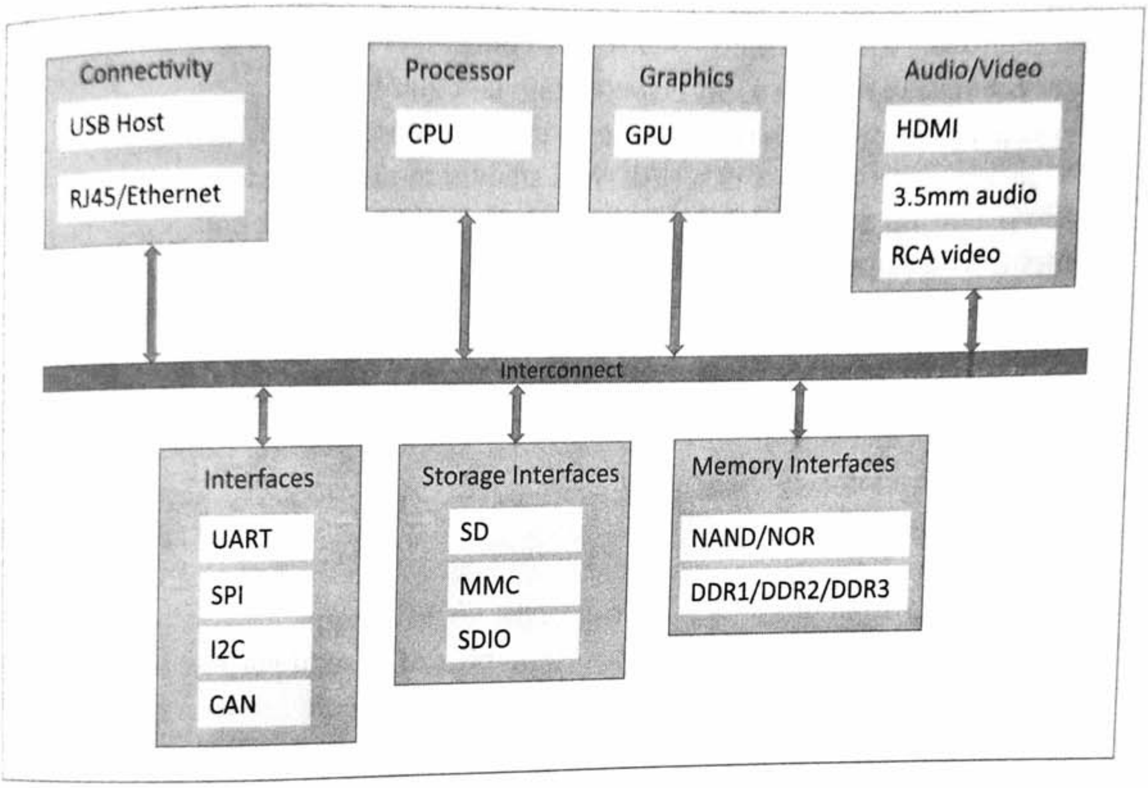


Figure 7.1: Block diagram of an IoT Device

7.2 Exemplary Device: Raspberry Pi

Raspberry Pi [104] is a low-cost mini-computer with the physical size of a credit card. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do. In addition to this, Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins. Since Raspberry Pi runs Linux operating system, it supports Python "out of the box".

7.3 About the Board

Figure 7.2 shows the Raspberry Pi board with the various components/peripherals labeled.

- **Processor & RAM** : Raspberry Pi is based on an ARM processor. The latest version of Raspberry Pi (Model B, Revision 2) comes with 700 MHz Low Power ARM1176JZ-F processor and 512 MB SDRAM.
- **USB Ports** : Raspberry Pi comes with two USB 2.0 ports. The USB ports on Raspberry Pi can provide a current upto 100mA. For connecting devices that draw current more than 100mA, an external USB powered hub is required.
- **Ethernet Ports** : Raspberry Pi comes with a standard RJ45 Ethernet port. You can connect an Ethernet cable or a USB Wifi adapter to provide Internet connectivity.
- **HDMI Output** : The HDMI port on Raspberry Pi provides both video and audio output. You can connect the Raspberry Pi to a monitor using an HDMI cable. For monitors that have a DVI port but no HDMI port, you can use an HDMI to DVI adapter/cable.
- **Composite Video Output** : Raspberry Pi comes with a composite video output with an RCA jack that supports both PAL and NTSC video output. The RCA jack can be used to connect old televisions that have an RCA input only.
- **Audio Output** : Raspberry Pi has a 3.5mm audio output jack. This audio jack is used for providing audio output to old televisions along with the RCA jack for video. The audio quality from this jack is inferior to the HDMI output.
- **GPIO Pins** : Raspberry Pi comes with a number of general purpose input/ouput pins. Figure 7.3 shows the Raspberry Pi GPIO headers. There are four types of pins on Raspberry Pi - true GPIO pins, I2C interface pins, SPI interface pins and serial Rx and Tx pins.
- **Display Serial Interface (DSI)** : The DSI interface can be used to connect an LCD panel to Raspberry Pi.
- **Camera Serial Interface (CSI)** : The CSI interface can be used to connect a camera module to Raspberry Pi.
- **Status LEDs** : Raspberry Pi has five status LEDs. Table 7.1 lists Raspberry Pi status LEDs and their functions.
- **SD Card Slot** : Raspberry Pi does not have a built in operating system and storage. You can plug-in an SD card loaded with a Linux image to the SD card slot. Appendix-A provides instructions on setting up New Out-of-the-Box Software (NOOBS) on Raspberry Pi. You will require atleast an 8GB SD card for setting up NOOBS.
- **Power Input** : Raspberry Pi has a micro-USB connector for power input.

Status LED	Function
ACT	SD card access
PWR	3.3V Power is present
FDX	Full duplex LAN connected
LNK	Link/Network activity
100	100 Mbit LAN connected

Table 7.1: Raspberry Pi Status LEDs

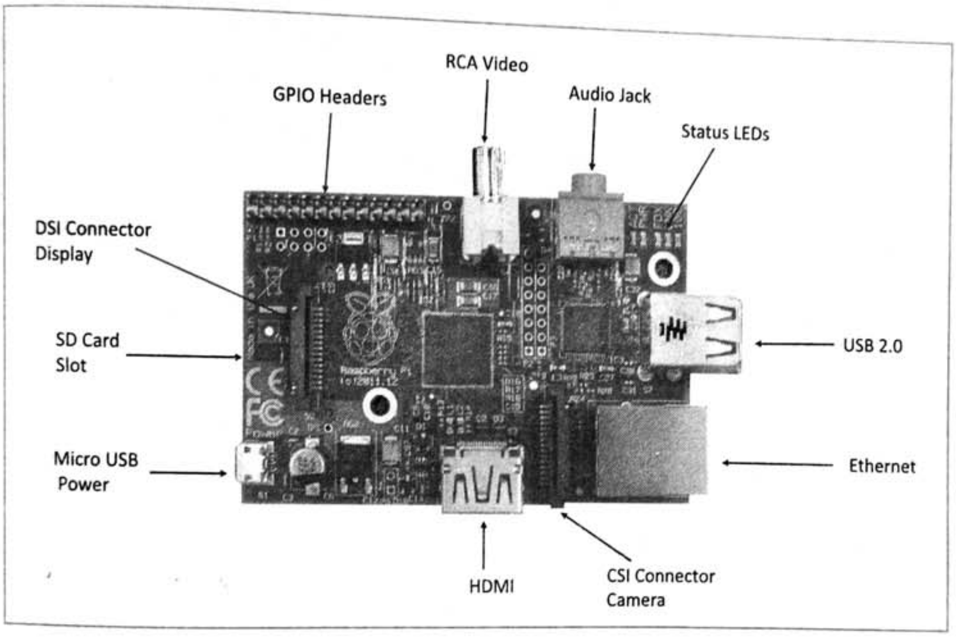


Figure 7.2: Raspberry Pi board

7.4 Linux on Raspberry Pi

Raspberry Pi supports various flavors of Linux including:

- **Raspbian** Raspbian Linux is a Debian Wheezy port optimized for Raspberry Pi. This is the recommended Linux for Raspberry Pi. Appendix-1 provides instructions on setting up Raspbian on Raspberry Pi.
- **Arch** : Arch is an Arch Linux port for AMD devices.
- **Pidora** : Pidora Linux is a Fedora Linux optimized for Raspberry Pi.
- **RaspBMC** : RaspBMC is an XBMC media-center distribution for Raspberry Pi.
- **OpenELEC** : OpenELEC is a fast and user-friendly XBMC media-center distribution.
- **RISC OS** : RISC OS is a very fast and compact operating system.

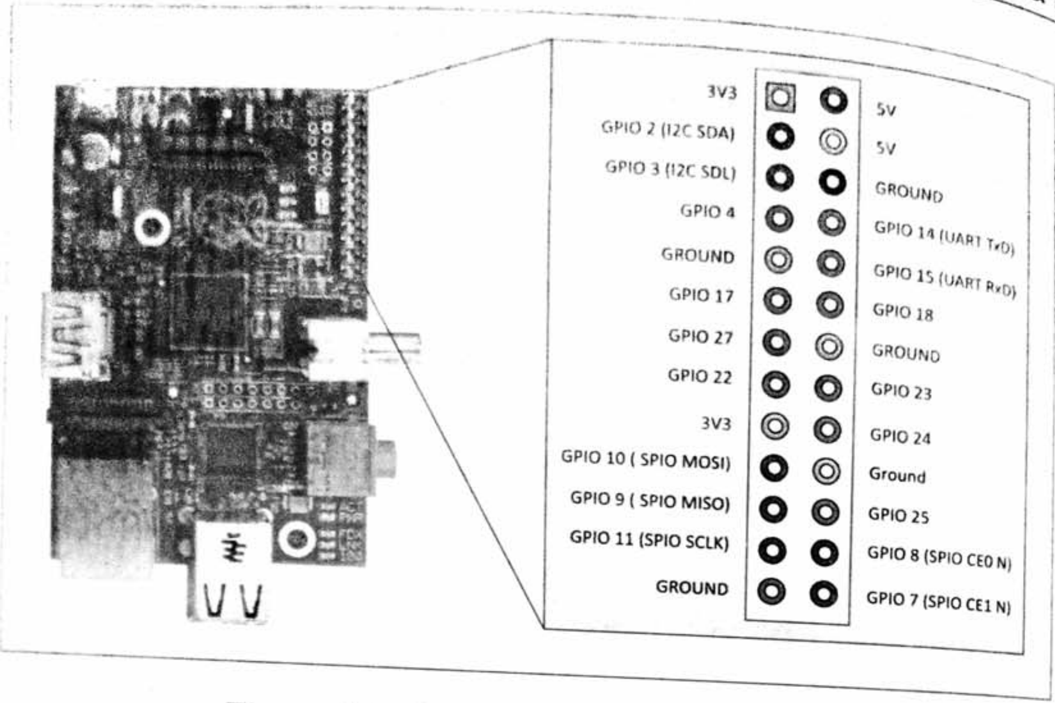


Figure 7.3: Raspberry Pi GPIO headers

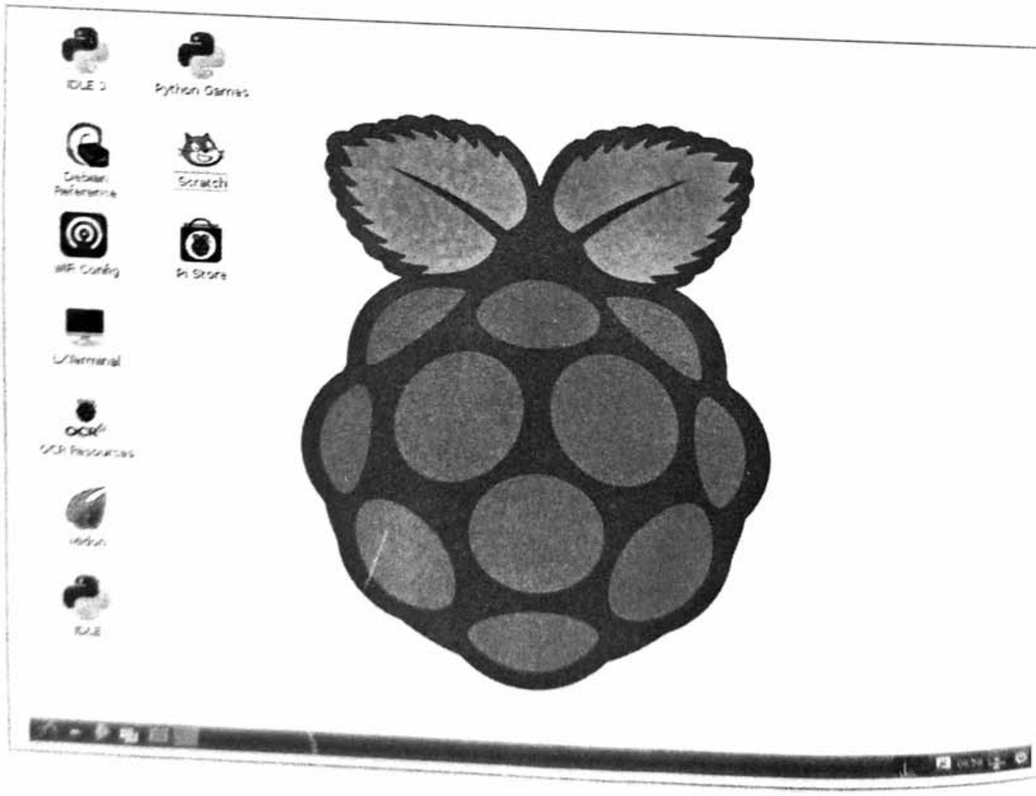


Figure 7.4: Rasbian Linux desktop

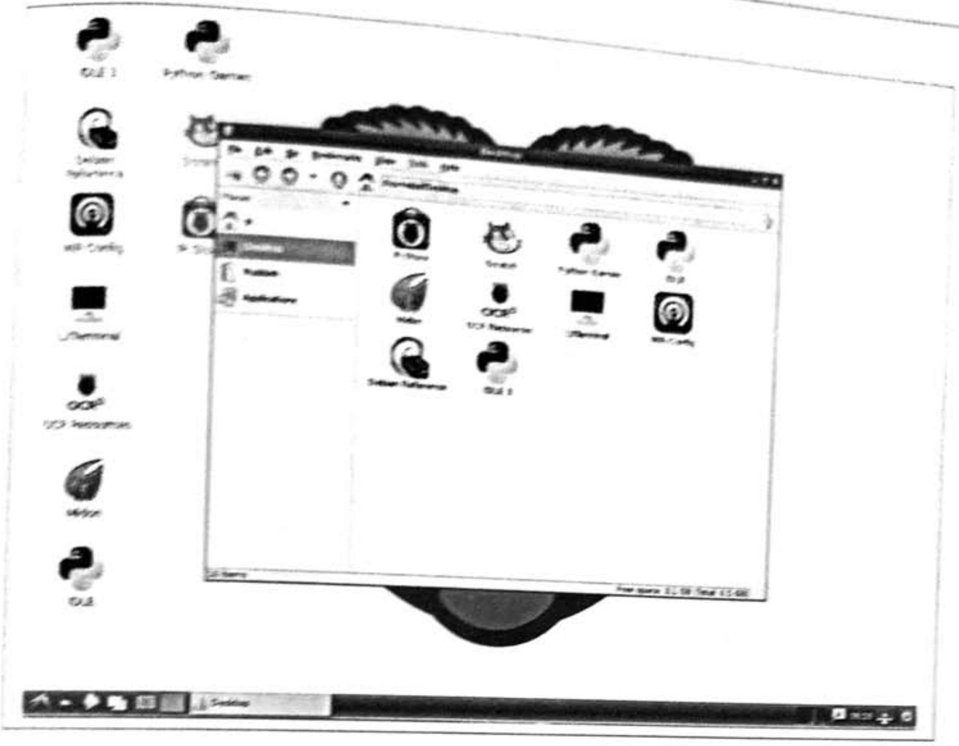
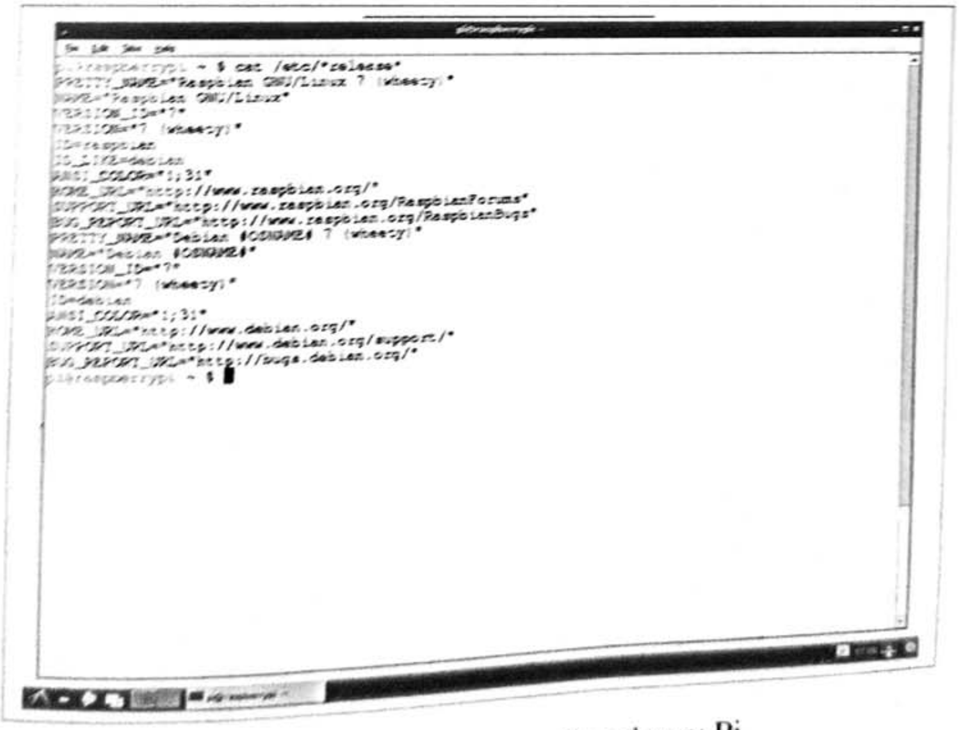
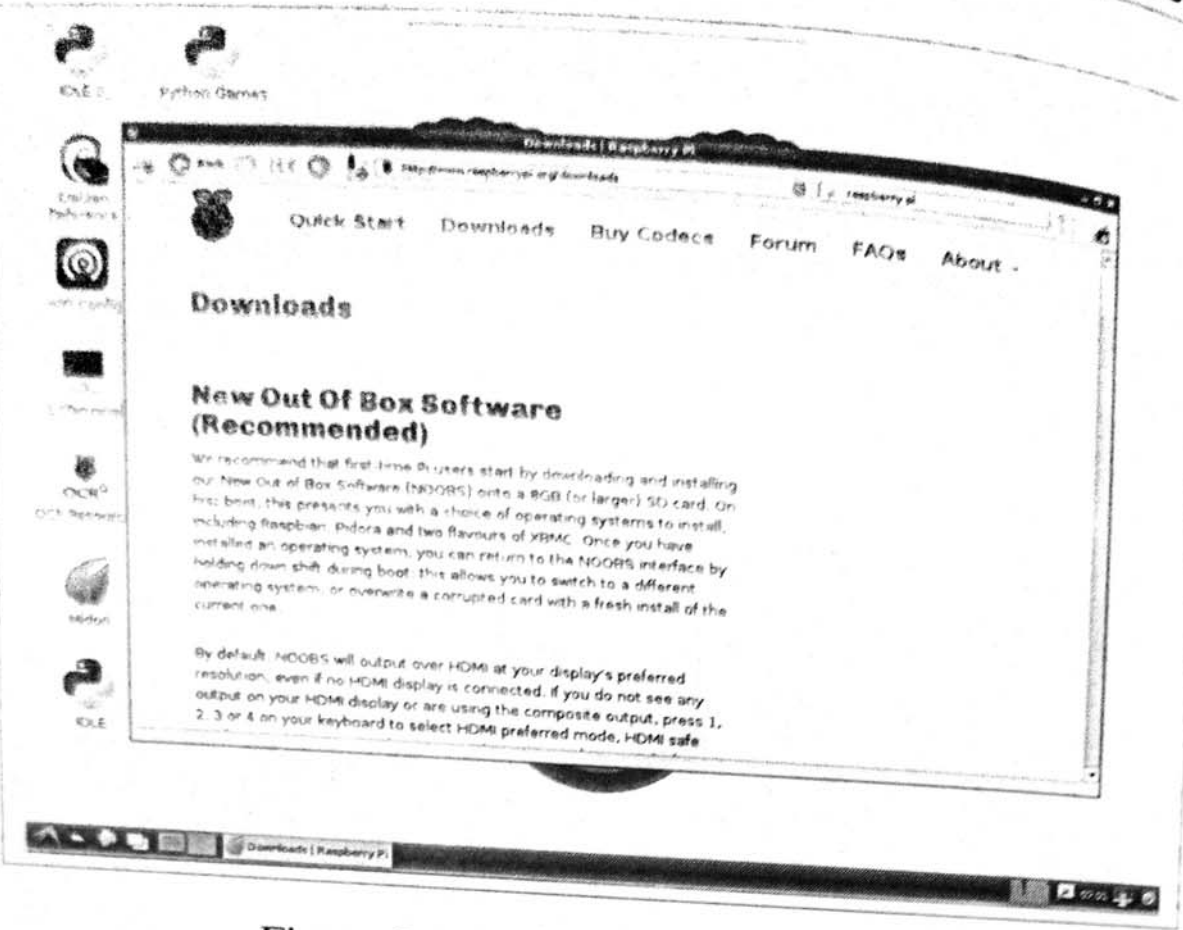


Figure 7.5: File explorer on Raspberry Pi



```
pi@raspberrypi ~ $ cat /etc/*release*
PRETTY_NAME="Raspbian GNU/Linux 7 (wheezy)"
NAME="Raspbian GNU/Linux"
VERSION_ID="7"
VERSION="7 (wheezy)"
ID=raspbian
ID_LIKE=debian
ANSI_COLOR="1;31"
HOME_URL="http://www.raspbian.org/"
SUPPORT_URL="http://www.raspbian.org/RaspbianForums"
BUG_REPORT_URL="http://www.raspbian.org/RaspbianBugs"
PRETTY_NAME="Debian $OSNAME$ 7 (wheezy)"
NAME="Debian $OSNAME$"
VERSION_ID="7"
VERSION="7 (wheezy)"
ID=debian
ANSI_COLOR="1;31"
HOME_URL="http://www.debian.org/"
SUPPORT_URL="http://www.debian.org/support/"
BUG_REPORT_URL="http://bugs.debian.org/"
pi@raspberrypi ~ $
```

Figure 7.6: Console on Raspberry Pi



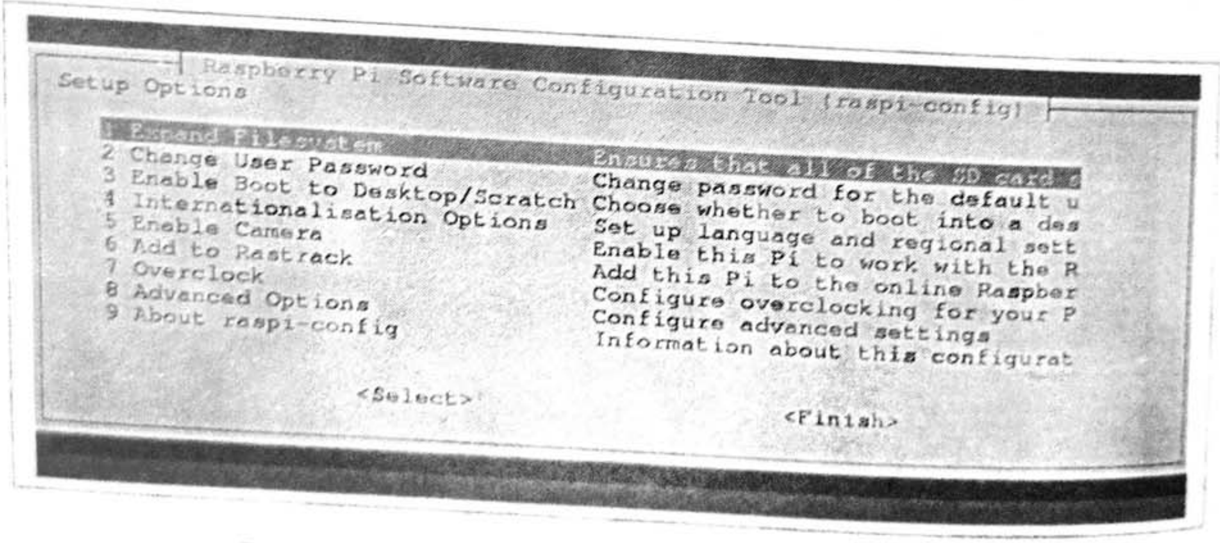
Downloads

New Out Of Box Software (Recommended)

We recommend that first-time Pi users start by downloading and installing our New Out of Box Software (NOOBS) onto a 8GB (or larger) SD card. On first boot, this presents you with a choice of operating systems to install, including Raspbian, Pidora and two flavours of XBMC. Once you have installed an operating system, you can return to the NOOBS interface by holding down shift during boot: this allows you to switch to a different operating system, or overwrite a corrupted card with a fresh install of the current one.

By default, NOOBS will output over HDMI at your display's preferred resolution, even if no HDMI display is connected. If you do not see any output on your HDMI display or are using the composite output, press 1, 2, 3 or 4 on your keyboard to select HDMI preferred mode, HDMI safe

Figure 7.7: Browser on Raspberry Pi



```
Setup Options     Raspberry Pi Software Configuration Tool (raspi-config)

  1 Expand Filesystem                  Ensures that all of the SD card s
  2 Change User Password               Change password for the default u
  3 Enable Boot to Desktop/Scratch     Choose whether to boot into a des
  4 Internationalisation Options       Set up language and regional sett
  5 Enable Camera                      Enable this Pi to work with the R
  6 Add to Rastrack                    Add this Pi to the online Raspber
  7 Overclock                          Configure overclocking for your P
  8 Advanced Options                   Configure advanced settings
  9 About raspi-config                 Information about this configurat

                  <Select>                        <Finish>
```

Figure 7.8: Raspberry Pi configuration tool

Figure 7.4 shows the Raspbian Linux desktop on Raspberry Pi. Figure 7.5 sh the default file explorer on Raspbian. Figure 7.6 shows the default console on Rasp Figure 7.7 shows the default browser on Raspbian. To configure Raspberry Pi, the raspi-co tool is used which can be launched from command line as ($raspi-config) as shown Figure 7.8. Using the configuration tool you can expand root partition to fill SD card, s keyboard layout, change password, set locale and timezone, change memory split, enab or disable SSH server and change boot behavior. It is recommended to expand the root file-system so that you can use the entire space on the SD card.

Though Raspberry Pi comes with an HDMI output, it is more convenient to access the device with a VNC connection or SSH. This does away with the need for a separate display for Raspberry Pi and you can use Raspberry Pi from your desktop or laptop computer. Appendix-A provides instructions on setting up VNC server on Raspberry Pi and the instructions to connect to Raspberry Pi with SSH. Table 7.2 lists the frequently used commands on Raspberry Pi.

Command	Function	Example
cd	Change directory	cd /home/pi
cat	Show file contents	cat file.txt
ls	List files and folders	ls /home/pi
locate	Search for a file	locate file.txt
lsusb	List USB devices	lsusb
pwd	Print name of present working directory	pwd
mkdir	Make directory	mkdir /home/pi/new
mv	Move (rename) file	mv sourceFile.txt destinationFile.txt
rm	Remove file	rm file.txt
reboot	Reboot device	sudo reboot
shutdown	Shutdown device	sudo shutdown -h now
grep	Print lines matching a pattern	grep -r "pi" /home/
df	Report file system disk space usage	df -Th
ifconfig	Configure a network interface	ifconfig
netstat	Print network connections, routing tables, interface statistics	netstat -lntp
tar	Extract/create archive	tar -xzf foo.tar.gz
wget	Non-interactive network downloader	wget http://example.com/file.tar.gz

Table 7.2: Raspberry Pi frequently used commands

7.5 Raspberry Pi Interfaces

Raspberry Pi has serial, SPI and I2C interfaces for data transfer as shown in Figure 7.3.

7.5.1 Serial

The serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serial peripherals.

7.5.2 SPI

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheral devices. In an SPI connection, there is one master device and one or more peripheral devices. There are five pins on Raspberry Pi for SPI interface:

- **MISO (Master In Slave Out)** : Master line for sending data to the peripherals.
- **MOSI (Master Out Slave In)** : Slave line for sending data to the master.
- **SCK (Serial Clock)** : Clock generated by master to synchronize data transmission
- **CE0 (Chip Enable 0)** : To enable or disable devices.
- **CE0 (Chip Enable 1)** : To enable or disable devices.

7.5.3 I2C

The I2C interface pins on Raspberry Pi allow you to connect hardware modules. I2C interface allows synchronous data transfer with just two pins - SDA (data line) and SCL (clock line).

7.6 Programming Raspberry Pi with Python

In this section you will learn how to get started with developing Python programs on Raspberry Pi. Raspberry Pi runs Linux and supports Python out of the box. Therefore, you can run any Python program that runs on a normal computer. However, it is the general purpose input/output capability provided by the GPIO pins on Raspberry Pi that makes it useful device for Internet of Things. You can interface a wide variety of sensor and actuators with Raspberry Pi using the GPIO pins and the SPI, I2C and serial interfaces. Input from the sensors connected to Raspberry Pi can be processed and various actions can be taken, for instance, sending data to a server, sending an email, triggering a relay switch.

7.6.1 Controlling LED with Raspberry Pi

Let us start with a basic example of controlling an LED from Raspberry Pi. Figure 7.9 shows the schematic diagram of connecting an LED to Raspberry Pi. Box 7.1 shows how to turn the LED on/off from command line. In this example the LED is connected to GPIO pin 18. You can connect the LED to any other GPIO pin as well.

Box 7.2 shows a Python program for blinking an LED connected to Raspberry Pi every second. The program uses the RPi. GPIO module to control the GPIO on Raspberry Pi. In this program we set pin 18 direction to output and then write *True/False* alternatively after a delay of one second.

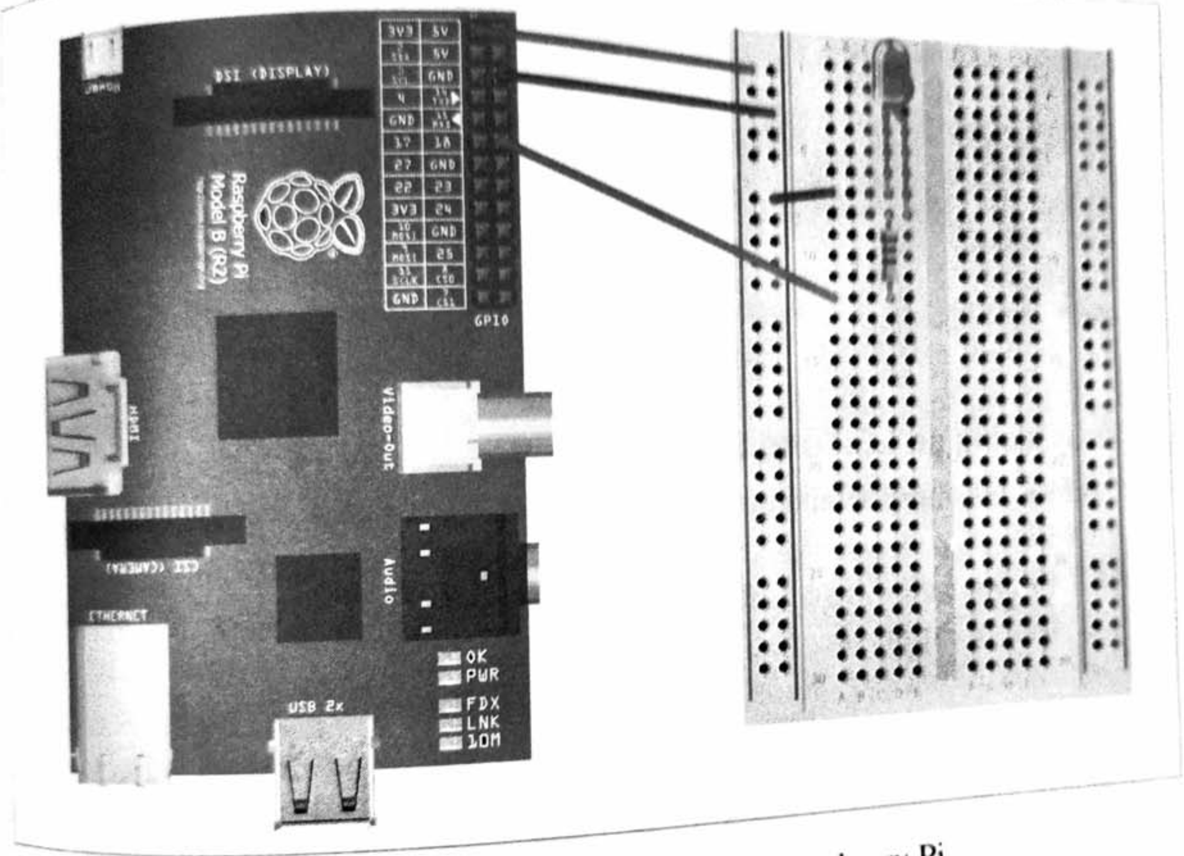


Figure 7.9: Controlling LED with Raspberry Pi

■ Box 7.1: Switching LED on/off from Raspberry Pi console

```
$echo 18 > /sys/class/gpio/export
$cd /sys/class/gpio/gpio18

#Set pin 18 direction to out
$echo out > direction

#Turn LED on
$echo 1 > value
```

```
#Turn LED off
$echo 0 > value
```

■ **Box 7.2: Python program for blinking LED**

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT)

while True:
   GPIO.output(18, True)
   time.sleep(1)
   GPIO.output(18, False)
   time.sleep(1)
```

7.6.2 Interfacing an LED and Switch with Raspberry Pi

Now let us look at a more detailed example involving an LED and a switch that is used to control the LED.

Figure 7.10 shows the schematic diagram of connecting an LED and switch to Raspberry Pi. Box 7.3 shows a Python program for controlling an LED with a switch. In this example the LED is connected to GPIO pin 18 and switch is connected to pin 25. In the infinite while loop the value of pin 25 is checked and the state of LED is toggled if the switch is pressed. This example shows how to get input from GPIO pins and process the input and take some action. The action in this example is toggling the state of an LED. Let us look at another example, in which the action is an email alert. Box 7.4 shows a Python program for sending an email on switch press. Note that the structure of this program is similar to the program in Box 7.3. This program uses the Python SMTP library for sending an email when the switch connected to Raspberry Pi is pressed.

■ **Box 7.3: Python program for controlling an LED with a switch**

```
from time import sleep
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
```

```
#Switch Pin
GPIO.setup(25, GPIO.IN)

#LED Pin
GPIO.setup(18, GPIO.OUT)

state=false

def toggleLED(pin):
  state = not state
  GPIO.output(pin, state)

while True:
  try:
    if (GPIO.input(25) == True):
      toggleLED(pin)
    sleep(.01)
  except KeyboardInterrupt:
    exit()
```

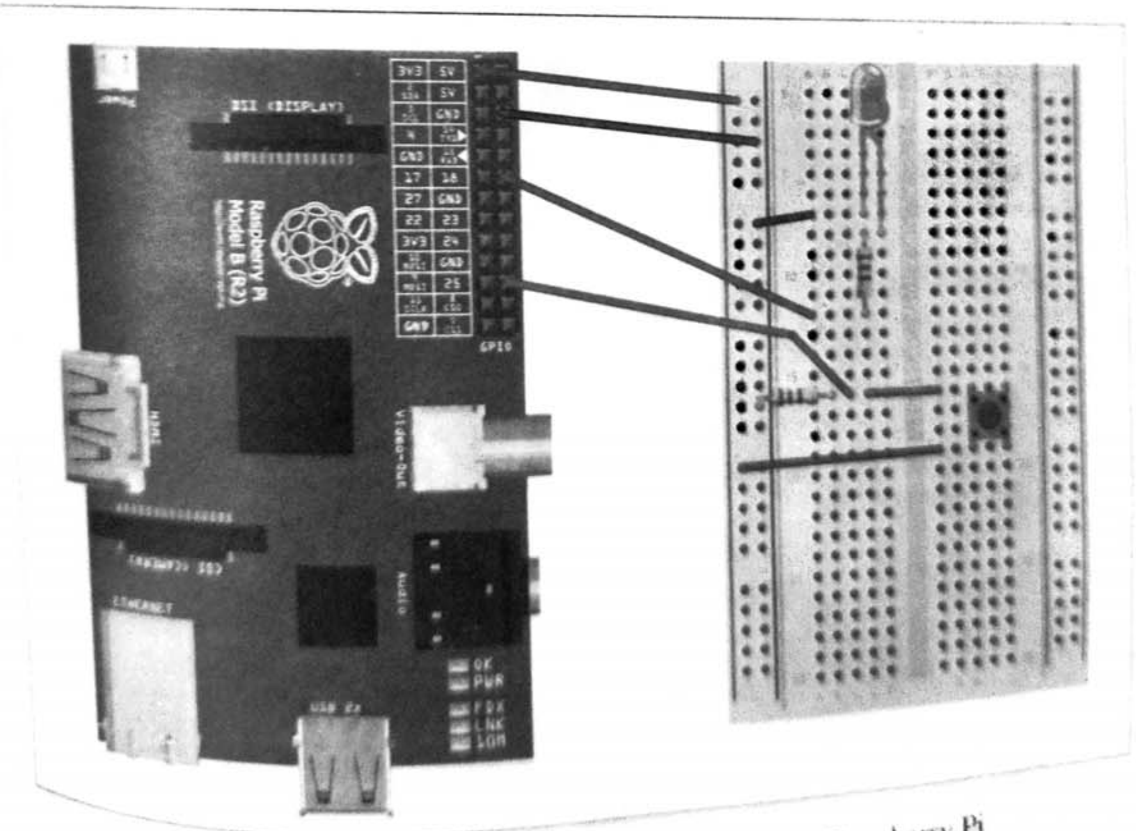


Figure 7.10: Interfacing LED and switch with Raspberry Pi

■ Box 7.4: Python program for sending an email on switch press

```
import smtplib
from time import sleep
import RPi.GPIO as GPIO
from sys import exit

from_email = '<my-email>'
receipients_list = ['<receipient-email>']
cc_list = []
subject = 'Hello'
message = 'Switch pressed on Raspberry Pi'
username = '<Gmail-username>'
password = '<password>'
server = 'smtp.gmail.com:587'

GPIO.setmode(GPIO.BCM)
GPIO.setup(25, GPIO.IN)


def sendemail(from_addr, to_addr_list, cc_addr_list,
      subject, message,
      login, password,
      smtpserver):

   header = 'From:  %s \n' % from_addr
   header += 'To:  %s \n' % ','.join(to_addr_list)
   header += 'Cc:  %s \n' % ','.join(cc_addr_list)
   header += 'Subject:  %s \n \n' % subject
   message = header + message

   server = smtplib.SMTP(smtpserver)
   server.starttls()
   server.login(login,password)
   problems = server.sendmail(from_addr, to_addr_list, message)
   server.quit()

while True:
   try:
      if (GPIO.input(25) == True):
         sendemail(from_email, receipients_list,
            cc_list, subject, message,
            username, password, server)
      sleep(.01)
```

```
    except KeyboardInterrupt:
        exit()
```

7.6.3 Interfacing a Light Sensor (LDR) with Raspberry Pi

So far you have learned how to interface LED and switch with Raspberry Pi. Now let us look at an example of interfacing a Light Dependent Resistor (LDR) with Raspberry Pi and turning an LED on/off based on the light-level sensed.

Figure 7.11 shows the schematic diagram of connecting an LDR to Raspberry Pi. Connect one side of LDR to 3.3V and other side to a 1μF capacitor and also to a GPIO pin (pin 18 in this example). An LED is connected to pin 18 which is controlled based on the light-level sensed.

Box 7.5 shows the Python program for the LDR example. The *readLDR*() function returns a count which is proportional to the light level. In this function the LDR pin is set to output and low and then to input. At this point the capacitor starts charging through the resistor (and a counter is started) until the input pin reads high (this happens when capacitor voltage becomes greater than 1.4V). The counter is stopped when the input reads high. The final count is proportional to the light level as greater the amount of light, smaller is the LDR resistance and greater is the time taken to charge the capacitor.

■ Box 7.5: Python program for switching LED/Light based on reading LDR reading

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
ldr_threshold = 1000
LDR_PIN = 18
LIGHT_PIN = 25

def readLDR(PIN):
  reading=0
  GPIO.setup(LIGHT_PIN, GPIO.OUT)
  GPIO.output(PIN, False)
  time.sleep(0.1)
  GPIO.setup(PIN, GPIO.IN)
  while (GPIO.input(PIN)==False):
    reading=reading+1
  return reading
```

3omr.

```
def switchOnLight(PIN):
    GPIO.setup(PIN, GPIO.OUT)
    GPIO.output(PIN, True)

def switchOffLight(PIN):
    GPIO.setup(PIN, GPIO.OUT)
    GPIO.output(PIN, False)



while True:
    ldr_reading = readLDR(LDR_PIN)
    if ldr_reading < ldr_threshold:
        switchOnLight(LIGHT_PIN)
    else:
        switchOffLight(LIGHT_PIN)

    time.sleep(1)
```

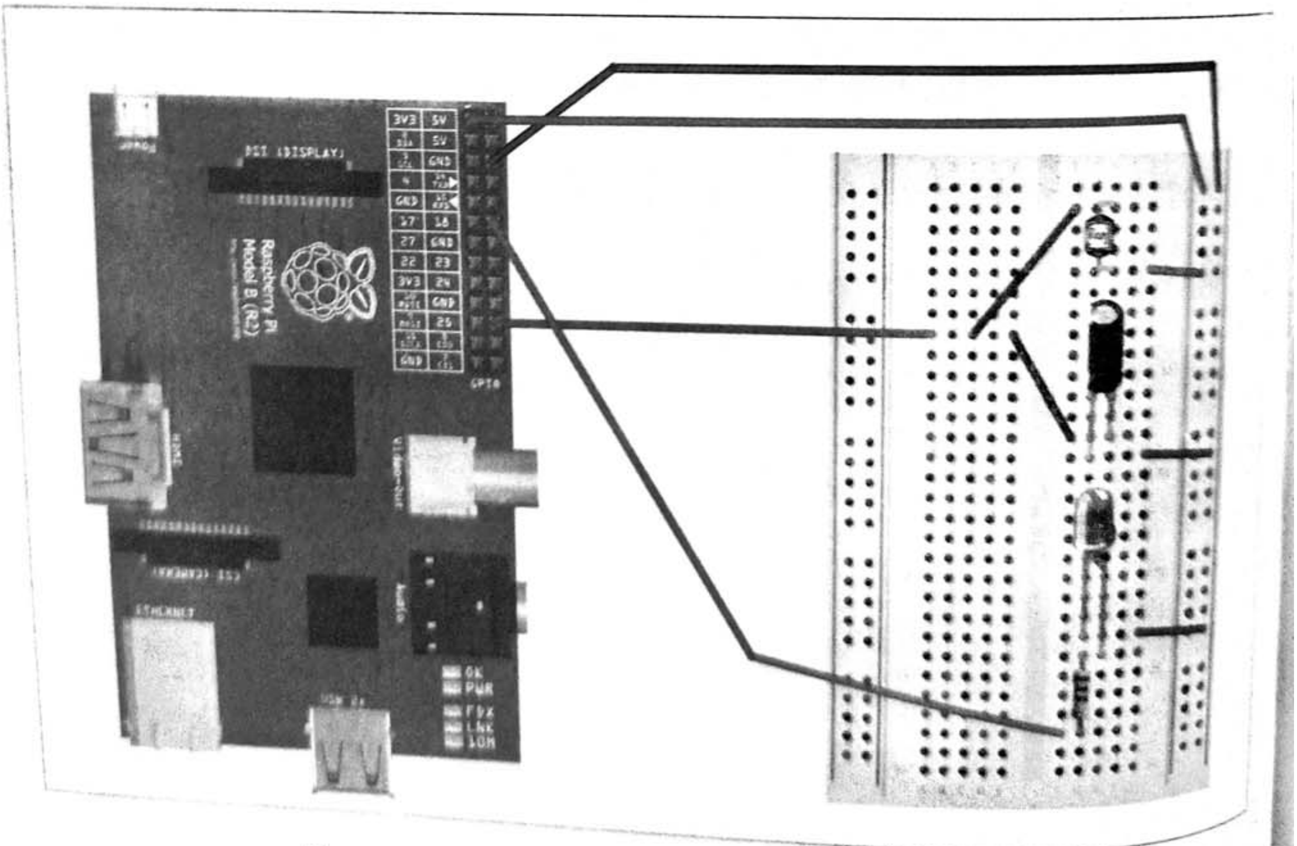


Figure 7.11: Interfacing LDR with Raspberry Pi

[6].

Transport Layer

The transport layer protocols provide end-to-end message transfer capability independent of the underlying network. The message transfer capability can be set up on connections, either using handshakes (as in TCP) or without handshakes/acknowledgements (as in UDP). The transport layer provides functions such as error control, segmentation, flow control and congestion control.

- ***TCP* :** Transmission Control Protocol (TCP) is the most widely used transport layer protocol, that is used by web browsers (along with HTTP, HTTPS application layer protocols), email programs (SMTP application layer protocol) and file transfer (FTP). TCP is a connection oriented and stateful protocol. While IP protocol deals with sending packets, TCP ensures reliable transmission of packets in-order. TCP also provides error detection capability so that duplicate packets can be discarded and lost packets are retransmitted. The flow control capability of TCP ensures that rate at which the sender sends the data is not too high for the receiver to process. The congestion control capability of TCP helps in avoiding network congestion and congestion collapse which can lead to degradation of network performance. TCP is described in RFC 793 [9].

- **UDP :** Unlike TCP, which requires carrying out an initial setup procedure, UDP is a connectionless protocol. UDP is useful for time-sensitive applications that have very small data units to exchange and do not want the overhead of connection setup. UDP is a transaction oriented and stateless protocol. UDP does not provide guaranteed delivery, ordering of messages and duplicate elimination. Higher levels of protocols can ensure reliable delivery or ensuring connections created are reliable. UDP is described in RFC 768 [10].

Application Layer

Application layer protocols define how the applications interface with the lower layer protocols to send the data over the network. The application data, typically in files, is encoded by the application layer protocol and encapsulated in the transport layer protocol which provides connection or transaction oriented communication over the network. Port numbers are used for application addressing (for example port 80 for HTTP, port 22 for SSH, etc.). Application layer protocols enable process-to-process connections using ports.

- **HTTP :** Hypertext Transfer Protocol (HTTP) is the application layer protocol that forms the foundation of the World Wide Web (WWW). HTTP includes commands such as GET, PUT, POST, DELETE, HEAD, TRACE, OPTIONS, etc. The protocol follows a request-response model where a client sends requests to a server using the HTTP commands. HTTP is a stateless protocol and each HTTP request is independent of the other requests. An HTTP client can be a browser or an application running on the client (e.g., an application running on an IoT device, a mobile application or other software). HTTP protocol uses Universal Resource Identifiers (URIs) to identify HTTP resources. HTTP is described in RFC 2616 [11].
- **CoAP :** Constrained Application Protocol (CoAP) is an application layer protocol for machine-to-machine (M2M) applications, meant for constrained environments with constrained devices and constrained networks. Like HTTP, CoAP is a web transfer protocol and uses a request-response model, however it runs on top of UDP instead of TCP. CoAP uses a client-server architecture where clients communicate with servers using connectionless datagrams. CoAP is designed to easily interface with HTTP. Like HTTP, CoAP supports methods such as GET, PUT, POST, and DELETE. CoAP draft specifications are available on IEFT Constrained environments (CoRE) Working Group website [12].

- **WebSocket :** WebSocket protocol allows full-duplex communication over a single socket connection for sending messages between client and server. WebSocket is based on TCP and allows streams of messages to be sent back and forth between the client and server while keeping the TCP connection open. The client can be a browser, a mobile application or an IoT device. WebSocket is described in RFC 6455 [13].
- **MQTT :** Message Queue Telemetry Transport (MQTT) is a light-weight messaging protocol based on the publish-subscribe model. MQTT uses a client-server architecture where the client (such as an IoT device) connects to the server (also called MQTT Broker) and publishes messages to topics on the server. The broker forwards the messages to the clients subscribed to topics. MQTT is well suited for constrained environments where the devices have limited processing and memory resources and the network bandwidth is low. MQTT specifications are available on IBM developerWorks [14].
- **XMPP :** Extensible Messaging and Presence Protocol (XMPP) is a protocol for real-time communication and streaming XML data between network entities. XMPP powers wide range of applications including messaging, presence, data syndication, gaming, multi-party chat and voice/video calls. XMPP allows sending small chunks of XML data from one network entity to another in near real-time. XMPP s a decentralized protocol and uses a client-server architecture. XMPP supports both client-to-server and server-to-server communication paths. In the context of IoT, XMPP allows real-time communication between IoT devices. XMPP is described in RFC 6120 [15].
- **DDS :** Data Distribution Service (DDS) is a data-centric middleware standard for device-to-device or machine-to-machine communication. DDS uses a publish-subscribe model where publishers (e.g. devices that generate data) create topics to which subscribers (e.g., devices that want to consume data) can subscribe. Publisher is an object responsible for data distribution and the subscriber is responsible for receiving published data. DDS provides quality-of-service (QoS) control and configurable reliability. DDS is described in Object Management Group (OMG) DDS specification [16].
- **AMQP :** Advanced Message Queuing Protocol (AMQP) is an open application layer protocol for business messaging. AMQP supports both point-to-point and publisher/subscriber models, routing and queuing. AMQP brokers receive messages from publishers (e.g., devices or applications that generate data) and route them over connections to consumers (applications that process data). Publishers publish the messages to exchanges which then distribute message copies to queues. Messages are either delivered by the broker to the consumers which have subscribed to the queues or the consumers can pull the messages from the queues. AMQP specification is

2016 [11].

- **CoAP :** Constrained Application Protocol (CoAP) is an application layer protocol for machine-to-machine (M2M) applications, meant for constrained environments with constrained devices and constrained networks. Like HTTP, CoAP is a web transfer protocol and uses a request-response model, however it runs on top of UDP instead of TCP. CoAP uses a client-server architecture where clients communicate with servers using connectionless datagrams. CoAP is designed to easily interface with HTTP. Like HTTP, CoAP supports methods such as GET, PUT, POST, and DELETE. CoAP draft specifications are available on IEFT Constrained environments (CoRE) Working Group website [12].

a mobile application of an IoT device. WebSocket is described in RFC 6455 [13].

- **MQTT :** Message Queue Telemetry Transport (MQTT) is a light-weight messaging protocol based on the publish-subscribe model. MQTT uses a client-server architecture where the client (such as an IoT device) connects to the server (also called MQTT Broker) and publishes messages to topics on the server. The broker forwards the messages to the clients subscribed to topics. MQTT is well suited for constrained environments where the devices have limited processing and memory resources and the network bandwidth is low. MQTT specifications are available on IBM developerWorks [14].

CHAPTER 1

Introduction

1.1 Welcome to the World of Embedded Processors

1.1.1 Where Are the Processors Used?

If you are new to microcontrollers or ARM® processors, first I would like to give you a very warm welcome.

Processors are used in majority of electronic products. For example, your mobile phones, televisions, washing machines, cars, bank card (smartcards), and even simple devices like the remote control for your radio can have processors inside. In most cases, these processors are placed inside in chips called **microcontrollers**. In modern microcontrollers, the chip also contains the essential elements like memory systems and interface hardware (often called peripherals). There are many different types of microcontrollers; they can be available with different processors, memory sizes, and peripherals inside, and can be available in different packages (Figure 1.1).

Large numbers of microcontrollers are designed for general purpose, which means they can be used in wide range of applications. Sometimes processors are used in chips that are

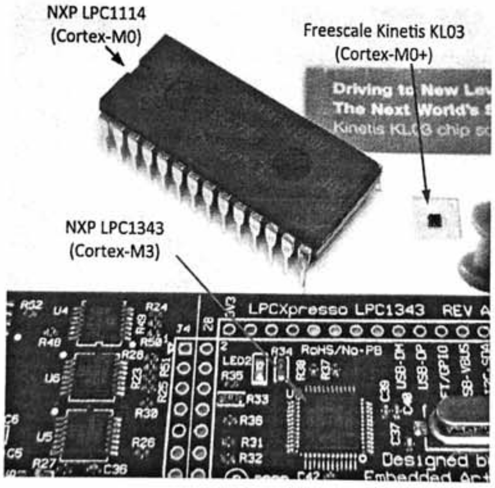


Figure 1.1
Microcontrollers are available in wide range of physical packages.

The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors. http://dx.doi.org/10.1016/B978-0-12-803277-0.00001-1

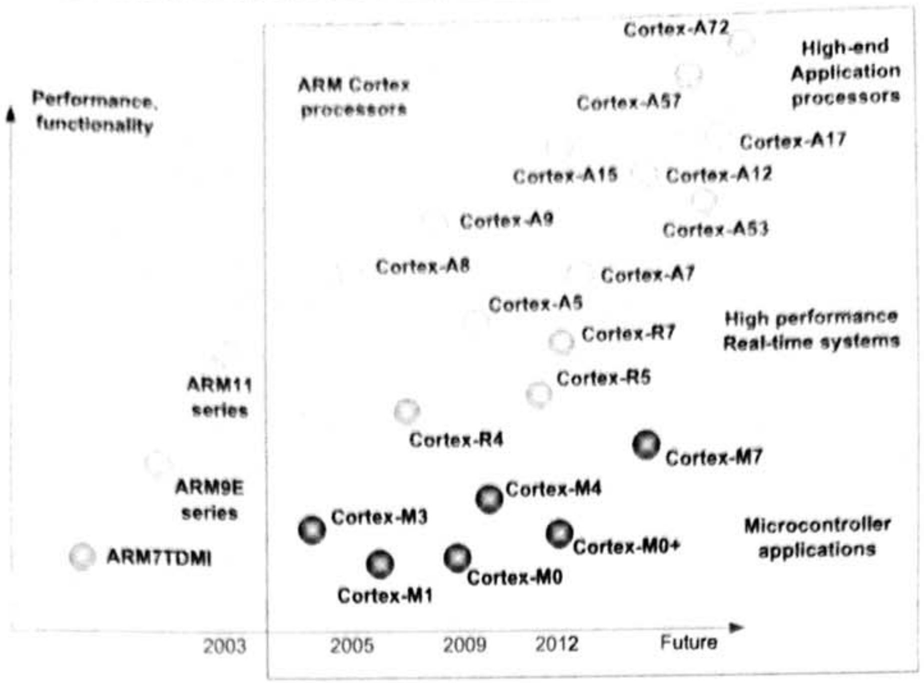


Figure 1.3
Overview of the ARM processor family.

In around 2003, ARM realized that it needs to diversify the processor products to address different technical requirements in different markets. As a result, three product profiles are defined, and the Cortex® processor brand name is created for the naming of these new processors:

Cortex-A processors—These are Application processors, which are designed to provide high performance and include features to support advanced operation systems (e.g., Android, Linux, Windows, iOS). These processors typically have longer processor pipeline and can run at relatively high clock frequency (e.g., over 1 GHz). In terms of features, these processors have Memory Management Unit (MMU) to support virtual memory addressing required by advanced OS, optional enhanced Java support, and a secure program execution environment called TrustZone®.

The Cortex-A processors are typically used in mobile phone, mobile computing devices (e.g., tablets), television, and some of the energy efficient servers.

While the Cortex-A processors have high performance, the processor is not designed to provide rapid response time to hardware events (i.e., real-time requirements). As a result, a different profile of high-performance processors is needed, and they are the Cortex-R processors.

Cortex-R processors—These are Real-Time, high performance processors that are very good at data crunching, can run at fairly high clock speed (e.g., 500 MHz to 1 GHz range), and at the same time can be very responsive to hardware events. They have cache memories as well as Tightly Coupled Memories, which enable deterministic behavior for interrupt handling. The Cortex-R processors are also designed with additional features to enable much higher system reliability such as Error Correction Code (ECC) support for memory systems and dual-core lock-step feature (i.e., redundant core logic for error detection).

The Cortex-R processors can be found in hard disk drive controllers, wireless baseband controllers/modem, specialized microcontrollers such as automotive and industrial controllers.

While the Cortex-R processors can be very good at high-performance microcontroller applications, they are quite complex designs and can consume fair amount of power. Therefore, another group of processors are need for the very low-power embedded products, and they are the Cortex-M processors.

Cortex-M Processors—The Cortex-M Processors are designed for main stream microcontroller market where the processing requirement is less critical, but need to be very low power. Most of the Cortex-M Processors are designed with a fairly short pipeline, for example, two stage in the Cortex-M0+ processor and three stages in Cortex-M0, Cortex-M3, and the Cortex-M4 Processors. The Cortex-M7 processor has a longer pipeline (six stages) due to higher performance requirement, but still the pipeline is a lot shorter than the designs of high-end application processors. As a result of the shorter pipeline and low power optimizations in the design, the maximum clock frequencies for these processors are slower than Cortex-R and Cortex-A processors, but this is rarely a problem because even a 100 MHz Cortex-M-based microcontroller can do a lot of work.

The Cortex-M processors are designed to provide very quick and deterministic interrupt responses. To achieve this, the processor's execution control part is closely coupled with a built-in interrupt controller called Nested Vectored Interrupt Controller (NVIC). The NVIC provides powerful and yet easy-to-use interrupt's management. In general, the Cortex-M processors are very easy to use, with almost everything can be programmed in C.

Due to their low power, fairly high performance, and ease of use benefits, the Cortex-M processors are selected by most major microcontroller vendors in their flagship microcontroller products. The Cortex-M processors are also used in some of the sensors, wireless communication chipsets, mixed signal ASICs/ASSPs, and even used as controller in some of the subsystems in complex application processors/SoC products.

In addition to the Cortex processor families, ARM also has processors specially designed for security-sensitive products, which included temper-resistance features. These processors are the SecurCore® series. For example, the SC000™, one of the SecurCore is designed based on the Cortex-M0 processor (same instruction set, and uses NVIC for interrupt management). The SecurCore products can be found in SIM cards, banking/payment systems, and even some electronic ID cards.

1.2.3 Blurring the Boundaries

In some ways, the term microcontroller can be a bit vague. Some of the microcontrollers are based on application processors such as ARM926EJ-S, one of the processor in the ARM9E processor family. In last few years, some of the microcontroller vendors starting to produce microcontroller products based on the ARM Cortex-A processors (e.g., Freescale Vybrid, Atmel SAMA5D3), and ARM Cortex-R processors (e.g., Texas Instruments TMS570, Spansion Traveo Family).

At the same time, the Cortex-M processors are also being used in many complex SoC devices as power management controller, I/O subsystem controller, etc.

In the next generation of Cortex-R processor based on the ARMv8-R architecture, the architecture definition also allows the processor to incorporate a MMU so that it can be used with a full feature OS like Linux or Android, and at the same time handle real-time tasks based on a virtualization mechanism.

1.2.4 ARM Cortex-M Processor Series

There are a number of processors in the Cortex-M processor family, as shown in Table 1.1.

If we look at the instruction set in a bit more details (Figure 1.4), we can see that the Cortex-M0, Cortex-M0+, and Cortex-M1 processors only support a small instruction set (56 instructions). Most of these instructions are 16 bit, thus provide a very good code density—which means it need a smaller program memory require for the same task compared to many architecture.

The instruction set of the Cortex-M0 and Cortex-M0+ processors are fairly simple. But if an application task involves complex data processing, then potentially a long sequence of instructions is needed to accomplish the operations in the Cortex-M0/M0+ processor because of the simple instruction set. In those cases, it might be better to use the Cortex-M3 processor because the Cortex-M3 processor supports a number of extra instructions (mostly 32 bit) that supports the following:

- More memory addressing modes
- Larger immediate data in the 32-bit instructions

Table 1.1: The Cortex®-M Processor family

Processor	Descriptions
Cortex-M0	The smallest ARM® processor—only approximately 12000[a] logic gates at minimum configuration. It is very low power and energy efficient.
Cortex-M0+	The most energy efficient ARM processor—it has a similar size as the Cortex-M0 processor, but with additional system level and debug features (all optional), and have higher energy efficiency than the Cortex-M0 processor design. It supports the same instruction set as the Cortex-M0 processor.
Cortex-M1	It is a small processor design optimized for field programmable Gate Array (FPGA) applications. It has the same instruction set and architecture as in the Cortex-M0 processor, but has FPGA specific memory system features.
Cortex-M3	When compared to the Cortex-M0 and Cortex-M0+ processors, the Cortex-M3 has a much more powerful instruction set, and its memory system is designed to provide higher processing throughput (e.g., use of Harvard bus architecture). It also has more system level and debug features, but at a cost of larger silicon area (minimum gate count is about 40000 gates) and slightly lower energy efficiency. In general, the energy efficiency of the Cortex-M3 processor is still a lot better than many traditional 8-bit and 16-bit microcontroller devices because the performance is substantially higher. The Cortex-M3 processor is very popular in the 32-bit microcontroller market.
Cortex-M4	The Cortex-M4 processor contains all the features of the Cortex-M3 processor, but with additional instructions to support DSP applications and have an option to include a floating point unit (FPU). It has the same system level and debug features as the Cortex-M3 processor.
Cortex-M7	It is a high performance processor designed to cover application spaces where the existing Cortex-M3 and Cortex-M4 processors cannot reach. Its instruction set is a superset of the Cortex-M4 processor, for example, supporting both single and double precision floating point calculations. It also has many advanced features, which are usually find in high-end processors such as caches and branch predictions.

[a] The exact gate count of a processor depends on many factors such as the semiconductor process library used, the chip design tool used, the design optimization options, signal routing constraints, etc.

- Longer branch and conditional branch ranges
- Additional branch instructions
- Hardware divide instructions
- Multiply accumulate (MAC) instructions
- Bit field processing instructions
- Saturation adjustment instructions

As a result, the Cortex-M3 processor can handle complicate data processing quicker. The code size might be similar to Cortex-M0 or Cortex-M0+ processor because although fewer number of instructions are required to perform the same operations, and these powerful instructions are mostly 32 bit instead of 16 bit. These 32-bit instructions also enable the Cortex-M3 processor to utilize the registers in the register bank better.

In some applications, however, you might need to perform some DSP operations such as filtering, signal transformations (e.g., Fast Fourier Transform), etc. In these applications,

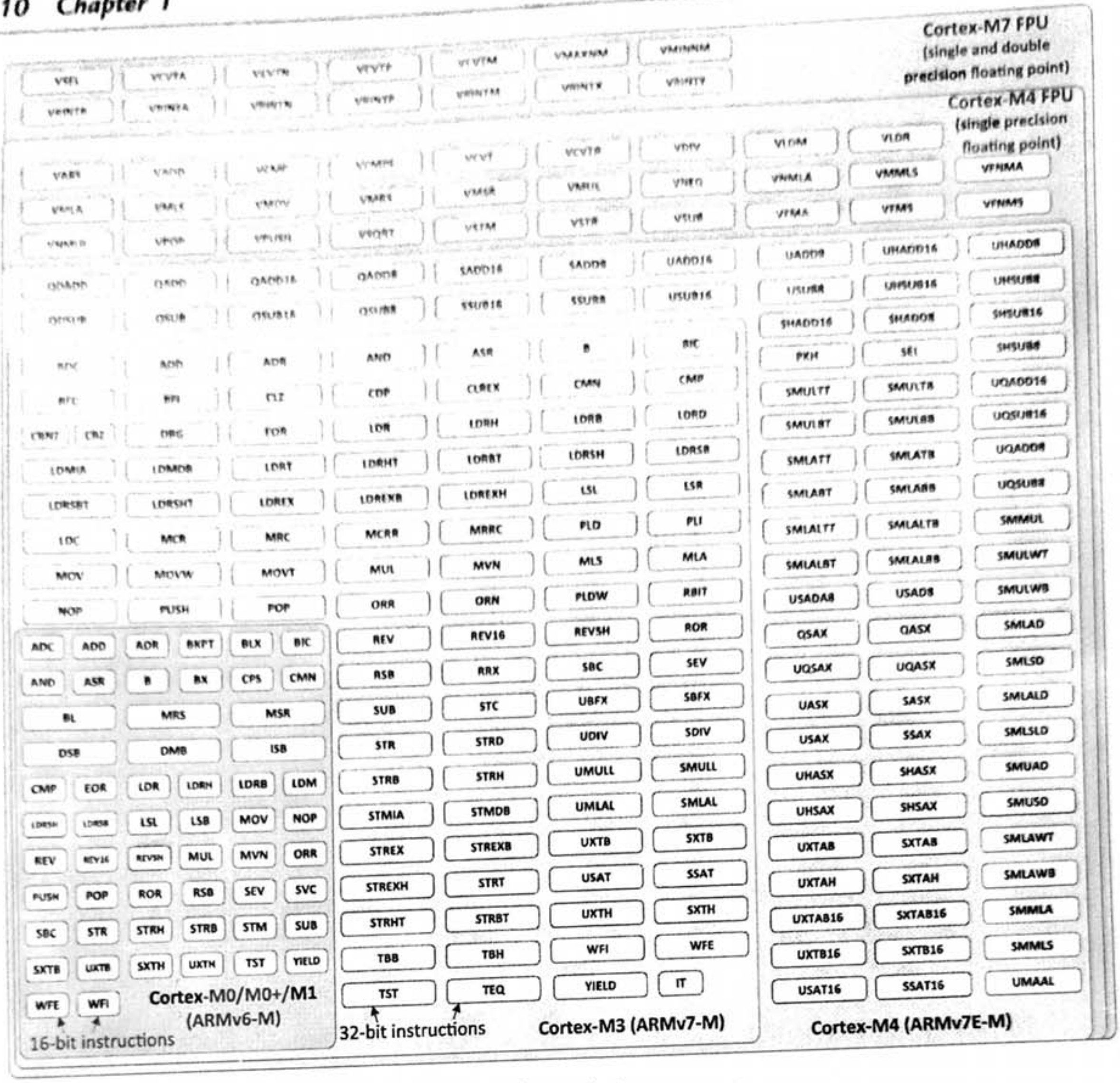


Figure 1.4
Instruction set of the Cortex®-M processor family.

you might want to use the Cortex-M4 processor because the Cortex-M4 processor added another group of instructions targeted for these applications—these included Single Instruction Multiple Data (SIMD) operations and saturated arithmetic instructions. The internal data path of the processor is also redesigned to enable single cycle MAC operations.

The Cortex-M4 processor also has an optional floating point unit that support IEEE-754 single precision floating point calculations. It does not mean that you cannot perform floating point processing in the Cortex-M0, Cortex-M0+, or other processors without the floating point unit. If you are using these processors for floating point operations, the compiler will insert runtime library functions to handle the floating point calculation using software, which can take much longer to do and need additional code size overhead.

For applications that demand very high data-processing requirements, or if double precision floating point calculation is needed, then the Cortex-M7 processor might be the best choice. It is designed to provide very high data-processing performance, but use the same programmer's model and a superset of the instruction set as Cortex-M4 processor.

To decide which processor to use in a project, you need to understand the processing requirements of the application. Some general guideline is shown in Table 1.2.

Please note that you might also need to consider the differences of the system-level features and performance when selecting the right Cortex-M processor. An overview of the comparison is shown in Table 1.3 and a comparison of the performance is shown in Table 1.4. Please note that the Cortex-M processors are very configurable and the exact features can be customized by the chip designers and vary among different devices.

In general, the ARM Cortex-M0 and Cortex-M0+ processors are both very suitable for ultra-low power applications, and because the instruction set and programmer's model are relatively simple, and the architecture is very C-friendly, they are also very suitable for beginners. For example, there is no need to learn a lot of tool chain-specific keywords or data types to get the application to work on a Cortex-M microcontroller, unlike many 8-bit or 16-bit architectures.

Table 1.2: The applications for various Cortex®-M Processors

Processor	Applications
Cortex-M0, Cortex-M0+ processors	General data processing and I/O control tasks. Ultra low power applications. Upgrade/replacement for 8-bit/16-bit microcontrollers. Low-cost ASICs, ASSPs
Cortex-M1	Field Programmable Gate Array(FPGA) applications with small to medium data processing complexity. (For high-complexity data processing there are FPGAs with built-in Cortex-A processors such as Xilinx Zynq-7000 and some of the Altera Arria V SoCs and Cyclone V SoCs).
Cortex-M3	Feature-rich/high-performance/low-power microcontrollers. Light-weight DSP applications.
Cortex-M4	Feature-rich/high-performance/low-power microcontrollers. DSP applications. Applications with frequent single precision floating point operations.
Cortex-M7	Feature-rich/very high performance power microcontrollers. DSP applications. Applications with frequent single or double precision floating point operations.

Table 1.3: An overview of the system level and debug features for various Cortex®-M Processors

Features	Cortex-M0	Cortex-M0+	Cortex-M1	Cortex-M3	Cortex-M4	Cortex-M7
Number of interrupts	1–32	1–32	1, 8, 16, 32	1–240	1–240	1–240
Interrupt priority levels	4	4	4	8–256	8–256	8–256
FPU	-	-	-	-	Optional (single precision)	Optional (single precision/single + double precision)
OS support	Y	Y	Optional	Y	Y	Y
Memory Protection unit	-	Optional	-	Optional	Optional	Optional
Cache	-	-	-	-	-	Optional
Debug	Optional	Optional	Optional	Optional	Optional	Yes
Instruction trace	-	Optional MTB	-	Optional ETM	Optional ETM	Optional ETM
Other trace	-	-	-	Optional	Optional	Optional

Table 1.4: Performance of various Cortex®-M Processors with commonly used benchmarks

Features	Cortex-M0	Cortex-M0+	Cortex-M3	Cortex-M4	Cortex-M7
Dhrystone 2.1 (per MHz)	0.9	0.95	1.25	1.25	2.14
CoreMark 1.0 (per MHz)	2.33	2.46	3.34	3.40	5.01

1.2.5 Quick Glance on the ARM Cortex-M0 and Cortex-M0+ Processor

The Cortex-M0 and Cortex-M0+ Processors:

- Are 32-bit Reduced Instruction Set Computing (RISC) processor, based on an architecture specification called ARMv6-M Architecture. The bus interface and internal data paths are 32-bit width.
- Have 16 32-bit registers in the register bank (r0 to r15). However, some of these registers have special purposes (e.g., R15 is the Program Counter, R14 is a register called Link Register, and R13 is the Stack Pointer).
- The instruction set is a subset of the Thumb Instruction Set Architecture. Most of the instructions are 16 bit to provide very high code density.
- Support up to 4 GB of address space. The address space is architecturally divided into a number of regions.
- Based on Von Neumann bus architecture (although arguably the Cortex-M0+ processor have a hybrid bus architecture because of an optional separate bus interface for fast peripheral register accesses, see section 4.3.2 Single Cycle I/O Interface in Chapter 4).

- Designed for low-power applications, including architectural support for sleep modes and have various low power features at the design/implementation level.
- Includes an interrupt controller called NVIC. The NVIC provides very flexible and powerful interrupt management.
- The system bus interface is pipelined, based on a bus protocol called Advanced High-performance Bus (AHB™) Lite. The bus interface supports transfers of 8-bit, 16-bit, and 32-bit data, and also allows wait states to be inserted. The Cortex-M0+ processor also have an optional bus interface (Single Cycle I/O interface, see section 4.3.2) for high-speed peripheral registers, which is separated from the main system bus.
- Support various features for the OS (Operating System) implementation such as a system tick timer, shadowed stack pointer, and dedicated exceptions for OS operations.
- Includes various debug features to enable software developers to create applications efficiently.
- Designed to be very easy to use. Almost everything can be programmed in C and in most cases no need for special C language extension for data types or interrupt handling support.
- Provide good performance in most general data processing and I/O control applications.

The Cortex-M0 and Cortex-M0+ processors do not include any memory and have only got one built-in timer which is primarily for OS operations. Therefore a chip designer needs to add additional components in the chip design themselves.

1.2.6 From Cortex-M0 Processor to Cortex-M0+ Processor

The ARM Cortex-M0 processor was released in 2009. It was a ground-breaking product because it is the first product that demonstrated it is possible to cramp a 32-bit processor into the silicon footprint similar to an 8-bit or 16-bit processors, while still able to make the design usable and provide excellent energy efficiency and a decent performance for a 32-bit processor.

Although the Cortex-M0 processor is a lot smaller than the Cortex-M3 processor (which was released in 2005), it maintains a number of key advantages as in Cortex-M3 processor:

- Flexible interrupt management using a built-in interrupt controller called NVIC
- OS support features including a timer hardware called SysTick (System Tick timer) and exception types dedicated to OS operations
- High code density
- Low power support such as sleep modes
- Integrated debug support
- Easy to use (almost everything programmable in plain C language)

The Cortex-M0 processor has been a very successful product, and was the fastest licensed ARM processor in 2009.[1] After the Cortex-M0 processor is released, the designers in ARM have received additional feedback from customers, microcontroller users and chip designers, and ARM decided that there is an opportunity for an enhanced version for the Cortex-M0 processor, which was subsequently called the Cortex-M0+ processor.

The Cortex-M0+ processor supports all the features available in the Cortex-M0 processor, but additional features were added to make it more powerful (these are all configurable by the chip designers):

- Unprivileged execution level and Memory Protection Unit (MPU)—this feature is available in other ARM processors such as the Cortex-M3 processor. It allows an OS to execute some of the application tasks with an unprivileged level so that the OS can impose memory access restrictions. For example, the unprivileged software cannot access critical system registers in the processors like NVIC registers, and memory access permissions can be managed by the MPU. In this way, a system can be made more robust because a misbehaving unprivileged task cannot corrupt critical data used by the OS kernel and other tasks.
- Vector Table relocation—again, this is a feature already existing in the Cortex-M3 processor. By default, the vector table is defined as the start of the memory (address 0x00000000). The Vector Table Offset Register allows the vector table to be defined in other memory locations such as a different program memory location or in SRAM. This is very useful for microcontroller devices, which might have separated vector table for boot process and user applications.
- Single Cycle I/O interface—this is a separate bus interface specifically added to allow frequently accessed I/O registers to be read/write in a single cycle. Without this feature, a load/store operation needs to go through the pipelined system bus, which needs two clock cycles per access. This feature enables microcontrollers or embedded system to have higher I/O performance, as well as higher energy efficiency in I/O intensive operations.

Internally to the processor design, there are also some significant changes. Instead of using a three-stage pipeline as in the Cortex-M0 and Cortex-M3 processors, the Cortex-M0+ processor is designed with a two-stage pipeline. This reduces the number of flip-flops in the processor, and hence reduces the dynamic power, and provides slightly higher performance at the same time because the branch penalty is reduced by one clock cycle.

In the Cortex-M0+ processor pipeline, as shown in Figure 1.5, a small part of the instruction decoding operations is carried out as soon as the instruction enters the

[1] Cortex-M0 Processor—Fastest Licensing ARM Processor (http://www.arm.com/about/newsroom/26419.php).

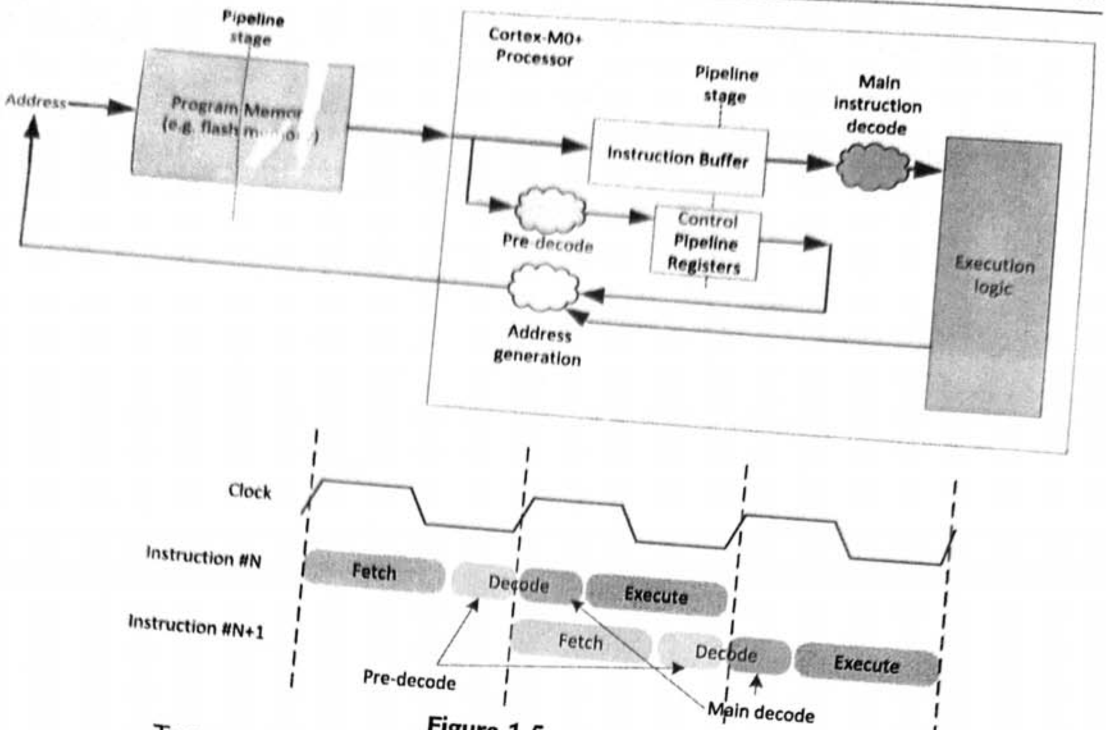


Figure 1.5
Two-stage Pipeline in the ARM® Cortex®-M0+ Processor.

processor bus interface. The rest of the instruction decoding is combined with the execution stage.

Adding decode logic to the instruction fetch stage do have some impact to the timing of the design. However, the balance between predecode and main decode logic was selected carefully to minimize the impact to the achievable maximum clock frequency. In addition, most of the low-power microcontrollers run at fairly low clock frequency in comparison to the maximum processor speed. Therefore this is not a problem to most of the silicon designs.

In some cases, the power consumption of the processor is reduced by 30% when comparing between Cortex-M0 processor and the Cortex-M0+ Processor. However, at the system level, the difference would be much smaller because most of the power could be consumed by the memory system.

In order to reduce system-level power, additional optimizations have been implemented to reduce the program memory accesses:

First, by shortening the processor to a two-stage pipeline design, the branch shadow of the processor is reduced. In a pipeline processor, when a branch instruction is executed, the

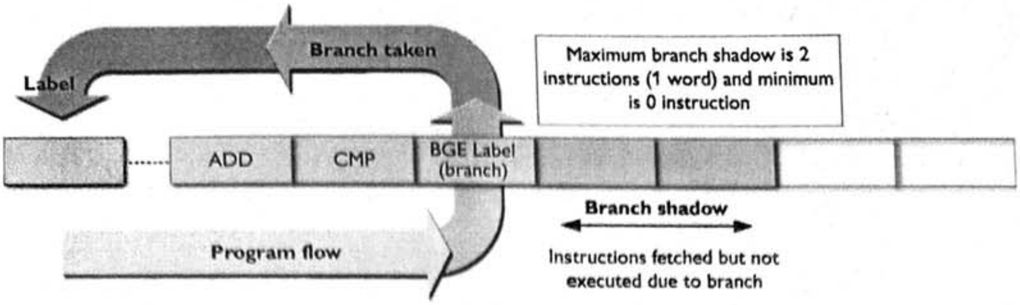

Branch taken
Label
Maximum branch shadow is 2 instructions (1 word) and minimum is 0 instruction
ADD
CMP
BGE Label (branch)
Branch shadow
Program flow
Instructions fetched but not executed due to branch

Figure 1.6
Power wastage reduction by reducing branch shadow. *Image courtesy of ARM®*.

instructions following the branch instruction would have been fetched by the processor. These instructions fetched are called branch shadow (Figure 1.6), and they are discarded by the processor and hence a long branch-shadow means wasting more energy.

Secondly, when a branch operation takes place and if the branch target instruction occupies only the second half of a 32-bit memory space (as shown in Figure 1.7), the instruction fetch is carried out as a 16-bit transfer. In this way, the program memory can switch off half of the byte lanes to reduce power.

The amount of power reduction by these techniques depends on how often branch operations are carried out in the application code.

Finally, in linear code execution, the program fetches are handled as 32-bit accesses. Since most of the instructions are 16-bit, each instruction fetch can provide up to two instructions. This means that the processor bus can be in idle state half of the time if there

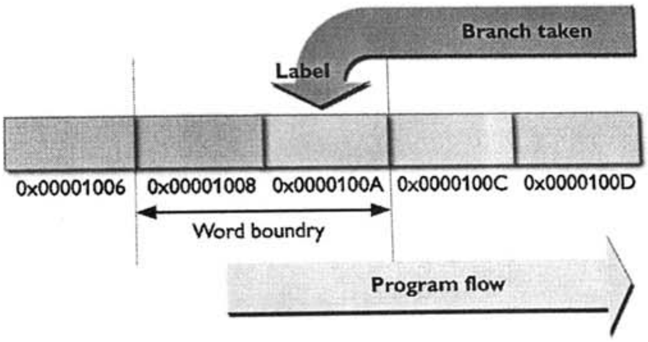


Figure 1.7
Power wastage reduction by fetching branch target with minimum transfer size. *Image courtesy of ARM®*.

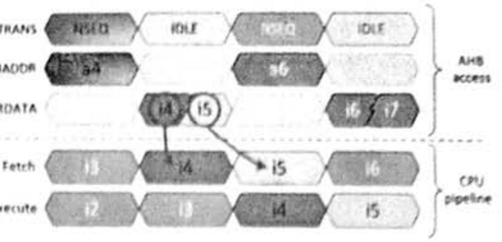


Figure 1.8
Program fetch power reduction by fetching up to two instructions at a time.
Image courtesy of ARM®.

is no data access instruction executed (Figure 1.8). Chip designers can utilize this characteristic to reduce the power consumption in the program memory (e.g., flash memory).

Another important enhancement in the Cortex-M0+ processor is the adding of a feature called Micro Trace Buffer (MTB). This unit enables low-cost instruction trace, which is very useful during software development, for example, helping to investigate the reason for a software failure. The details of the MTB are covered in Chapter 13 and appendix E.

The Cortex-M0+ processor have additional enhancements when compared to the Cortex-M0 processor in terms of chip design aspects (most of these are invisible to microcontroller users). For example, a hardware interface was added to allow the startup sequence of the processor to be delayed, which is useful for many SoC designs with multiple processors.

Today, many microcontroller vendors already started offering microcontroller products based on the Cortex-M0+ processors.

1.2.7 Applications of the Cortex-M0 and Cortex-M0+ Processor

The Cortex-M0 and Cortex-M0+ processors are used in a wide range of products.

Microcontrollers

The most common usage is microcontrollers. Many Cortex-M0 and Cortex-M0+ microcontrollers are low-cost devices and are designed for low-power applications. They can be used in applications including computer peripherals and accessories, toys, white goods, industrial and HVAC (heating, ventilating, and air conditioning) controls, home automation, etc.

When comparing the microcontrollers based on the Cortex-M0 and Cortex-M0+ processors to traditional 8-bit and 16-bit microcontroller products, the Cortex-M microcontrollers allow embedded products to be built with more features, more sophisticated user interface, due to support of larger address space, powerful interrupt control, and higher performance.

The better performance and small size also bring the benefit of higher energy efficiency. For example, for the same processing task, you can finish the processing quicker and allow the system to stay in sleep modes longer.

Another advantage of using ARM Cortex-M processors for microcontroller applications is that they are very easy to use. Therefore it is very appealing to many microcontroller vendors as product support and educating the users can be challenging for some other processor architectures.

ASICs and ASSPs

Another important group of applications for the Cortex-M0 and Cortex-M0+ processors are ASICs and ASSPs. For example, there are a number of touch screen controllers, sensors, wireless controllers, Power Management ICs (PMIC), and smart battery controllers designed based on the Cortex-M0 or Cortex-M0+ processors.

In these applications, the low gate count advantage of the Cortex-M0 and Cortex-M0+ processors allow high performance processing capability to be included in chip designs that traditionally only allow 8-bit or simple 16-bit processors to be used.

System on Chips

For complex SoC, the designs are often divided into a main application processor system and a number of subsystems for: I/O controls, communication protocol processing, and system management. In some cases, the Cortex-M0 and Cortex-M0+ processor can be used in part of the subsystems to off-load some activities from the main application processor, and to allow small amount of processing be carried out while the main processor is in standby mode (e.g., in battery powered products). It might also be used as a System Control Processor (SCP) for boot sequence management and power management.

1.3 What Is Inside a Microcontroller

1.3.1 Typical Elements Inside a Microcontroller

There can be many components inside a basic microcontroller. For example, a simplified block diagram is shown in Figure 1.9:

In the diagram there are a lot of acronyms. They are explained in Table 1.5.

As shown in Figure 1.9, there can be a lot of components in a microcontroller (not to mention other complex interfaces like Ethernet, USB, etc.). In some microcontrollers you

Technical Overview

2.1 What are the Cortex®-M0 and Cortex-M0+ Processors?

The ARM® Cortex-M0 processor and Cortex-M0+ processors are both 32-bit processors. Their internal registers in the register banks, data paths, and the bus interfaces are all 32 bit. Both of them have a single main system bus interface, therefore they are considered as Von Neumann bus architecture.

The Cortex-M0+ processor has an optional single cycle I/O interface that is primarily for faster peripheral I/O register accesses. Therefore, it is possible to say the Cortex-M0+ processor has limited Harvard bus architecture capability as instruction access and I/O register accesses could be carried out at the same time, but it is important to understand that although there can be two bus interfaces, the memory space is shared (unified) and therefore the extra bus interface does not bring additional addressable memory space.

The key characteristics of the Cortex-M0 and Cortex-M0+ processors are as follows:

Processor pipeline

- The Cortex-M0 processor has a three-stage pipeline (fetch, decode, and execute)
- The Cortex-M0+ processor has a two-stage pipeline (fetch + predecode, decode + execute)

Instruction set

- The instruction set is based on Thumb® Instruction Set Architecture (ISA). Only a subset of the Thumb ISA is used (56 of them). Most of the instructions are 16 bit in size, only a few of them are 32 bit.
- In general, the Cortex-M processors are classified as Reduced Instruction Set Computing although they have instructions of different sizes.
- Support optional single cycle 32 bit × 32 bit multiply, or a smaller multicycle multiplier for designs that need small silicon area.

Memory addressing

- 32-bit addressing supporting up to 4 GB of memory space
- The system bus interface is based on an on-chip bus protocol called AHB-Lite, supporting 8-bit, 16-bit, and 32-bit data transfers
- The AHB-Lite protocol is pipelined, support high operation frequency for the system. Peripherals can be connected to a simpler bus based on APB protocol (Advanced Peripheral Bus) via an AHB to APB bus bridge.

The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors. http://dx.doi.org/10.1016/B978-0-12-803277-0.00002-3

Interrupt Handling

- The processors include a built-in interrupt controller called the Nested Vectored Interrupt Controller (NVIC). This unit handles interrupt prioritization and masking functions. It supports up to 32 interrupt requests from various peripherals (chip design dependent), an additional Non-Maskable Interrupt (NMI) input, and also support a number of system exceptions.
- Each of the interrupts can be set to one of the four programmable priority levels. NMI has a fixed priority level.

Operating Systems (OS) support

- Two system exception types (SVCall and PendSV) are included to support OS operations.
- An optional 24-bit hardware timer called SysTick (System Tick Timer) is also included for periodic OS time keeping.
- The Cortex-M0+ processor support privileged and unprivileged execution level (optional to chip designers). This allows OS to run some of the application tasks with unprivileged execution level and impose memory access restrictions to these tasks.
- The Cortex-M0+ processor has an optional Memory Protection Unit (MPU) to allow OS to define memory access permission for application tasks during run time.

Low Power support

- Architecturally two sleep modes are defined as normal sleep and deep sleep. The exact behaviors in these sleep modes are device specific (depends on which chip you are using). Chip designers can also add device specific power saving mode control registers to expand the number of sleep modes or to allow the sleep mode behavior for each part of the chip to be defined.
- Sleep mode can be entered using WFI (Wait for Interrupt) or WFE (Wait for Event) instructions, or using a feature called Sleep-on-Exit to allow the processor to enter sleep automatically.
- Additional hardware level supports to enable chip designers to create better power reductions based on the sleep mode features, for example, the Wake-up Interrupt Controller (WIC).

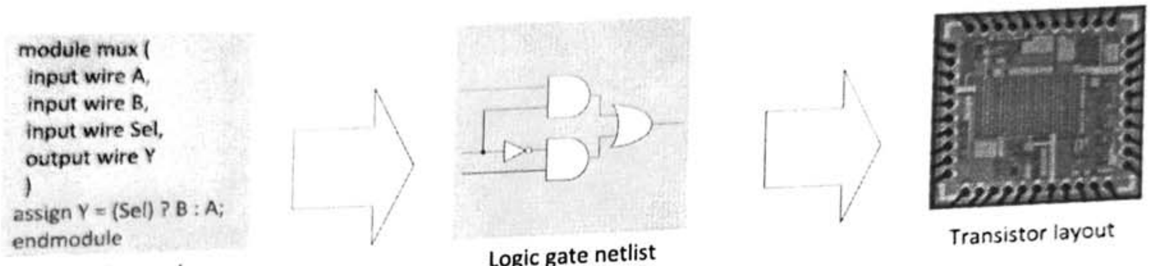
Debug

- The debug system is based on the ARM CoreSight™ Debug Architecture. It is a scalable debug architecture that can support simple-single processor designs to complex multi-processor designs.
- A debug interface that can either be based on JTAG protocol (4 or five pins), or Serial Wire Debug protocol (2 pins). The debug interface allows software developers to access debug features of the processors.
- Support up to four hardware breakpoints, two data watchpoints, and unlimited software breakpoint using BKPT (breakpoint) instruction.
- Support basic program execution profiling using a feature called Program Counter (PC) Sampling via the debug connection.

- The Cortex-M0+ Processor has an optional feature called Micro Trace Buffer (MTB), this provide instruction trace.

The Cortex-M Processors are configurable designs. They are delivered to chip designers in form of Verilog source code files with a number of parameters that chip designers can select. In this way, chip designers can omit some of the features that are unnecessary for their projects to save power and reduce silicon area. As a result, you can find microcontrollers based on the Cortex-M0 and Cortex-M0+ processor with different number of supported interrupts, and Cortex-M0+ processor with and without the optional MPU.

During the design process (Figure 2.1), the processor is integrated with the rest of the system and converted to a design composed of logic gates and then transistors layout using chip design tools. The timing characteristics like maximum clock frequency are defined at these stages based on the semiconductor process selected for the project and various design constraints. In addition, the exact maximum speed and power consumption of the Cortex-M0 or Cortex-M0+ processor on different products can also be different from each other.



```
module mux (
  input wire A,
  input wire B,
  input wire Sel,
  output wire Y
)
assign Y = (Sel) ? B : A;
endmodule
```

Verilog code

Logic gate netlist

Transistor layout

Figure 2.1
Simplified chip design flow.

2.2 Block Diagrams

A simplified block diagram of the Cortex®-M0 processor is shown in Figure 2.2.

The processor core contains the register banks, ALU, data path, and control logic. It is a three-stage pipeline design with fetch stage, decode stage, and execution stage. The register bank has sixteen 32-bit registers. A few of the registers in the register bank have special usages (e.g., PC). The rest are available for general data processing.

The NVIC accepts up to 32 interrupt request signals and a NMI input. It contains the functionality required for comparing priority between interrupt requests and current priority level so that nested interrupts can be handled automatically. If an interrupt is accepted, the NVIC communicates with the processor so that the processor can execute the correct interrupt handler.

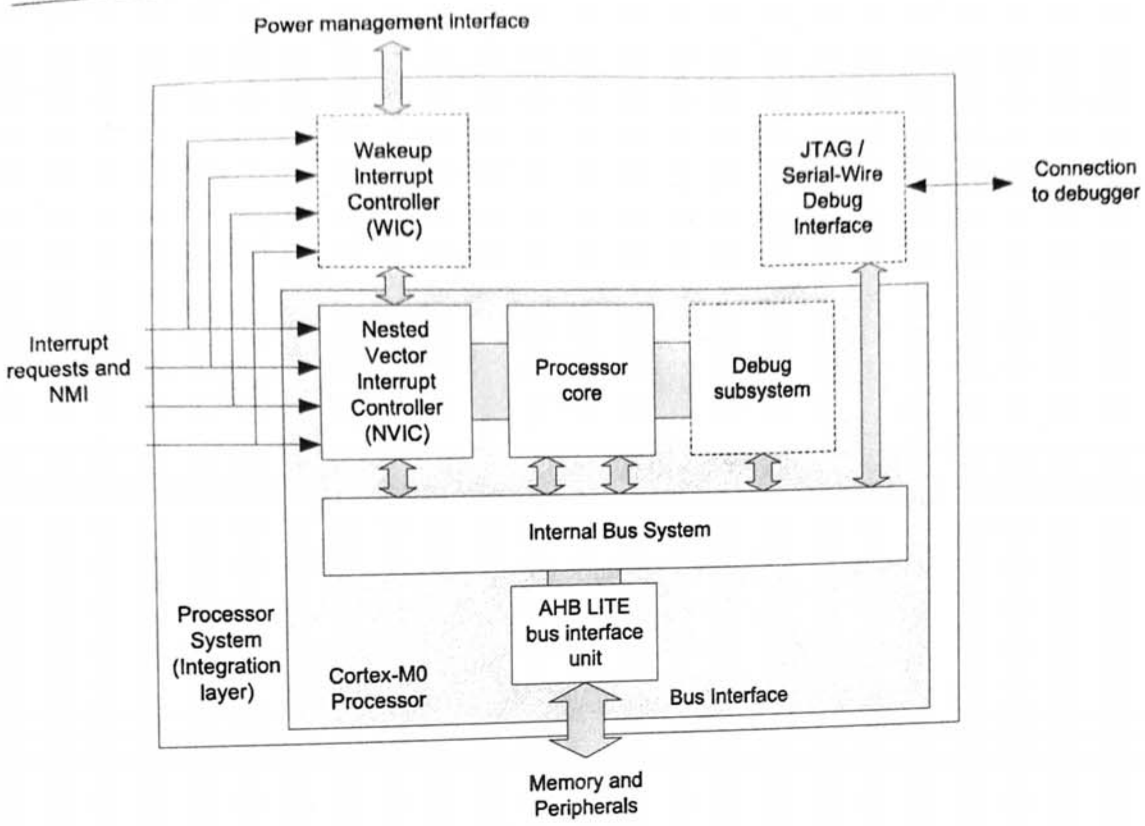

Power management interface
Wakeup Interrupt Controller (WIC)
JTAG / Serial-Wire Debug Interface
Connection to debugger
Interrupt requests and NMI
Nested Vector Interrupt Controller (NVIC)
Processor core
Debug subsystem
Internal Bus System
AHB LITE bus interface unit
Processor System (Integration layer)
Cortex-M0 Processor
Bus Interface
Memory and Peripherals

Figure 2.2
A simplified block diagram of the Cortex®-M0 Processor.

The WIC is an optional unit. In low-power applications, the microcontroller can enter standby state with most parts of the processor powered down. Under this situation, the WIC can perform the function of interrupt masking while the NVIC and the processor core are inactive. When an interrupt request is detected, the WIC informs the power management to power up the system so that the NVIC and the processor core can then handle the rest of the interrupt processing.

The debug subsystem contains various functional blocks to handle debug control, program breakpoints, and data watchpoints. When a debug event occurs, it can put the processor core in a halted state so that embedded developers can examine the status of the processor at that point.

The internal bus system, data path in the processor core, and the AHB-Lite bus interface are all 32-bit wide. AHB-Lite is an on-chip bus protocol used in many ARM® processors. This bus protocol is part of the AMBA® (Advanced Microcontroller Bus Architecture) specification, which is a bus architecture developed by ARM and widely used in the IC design industry.

The JTAG or Serial Wire interface units provide access to the bus system and debugging functionalities. The JTAG protocol is a popular 4-pin (5-pin if including a reset signal) communication protocol commonly used for IC and PCB testing. The Serial Wire protocol is a newer communication protocol that only requires two wires, but it can handle the same debug functionalities as JTAG. As illustrated in the block diagrams (Figures 2.2 and 2.3), the debug interface module is separated from the processor design. This is required in the CoreSight™ Debug Architecture where multiple processors can share the same debug connections. There are a number of additional signals for multiprocessor debug support not shown in the diagrams.

The Cortex-M0+ processor is very similar (as shown in Figure 2.3) to Cortex-M0 processor. The only addition is the adding of the optional MPU, single cycle I/O interface bus and the interface for the MTB. The processor core internal design is also changed to a two-stage pipeline arrangement.

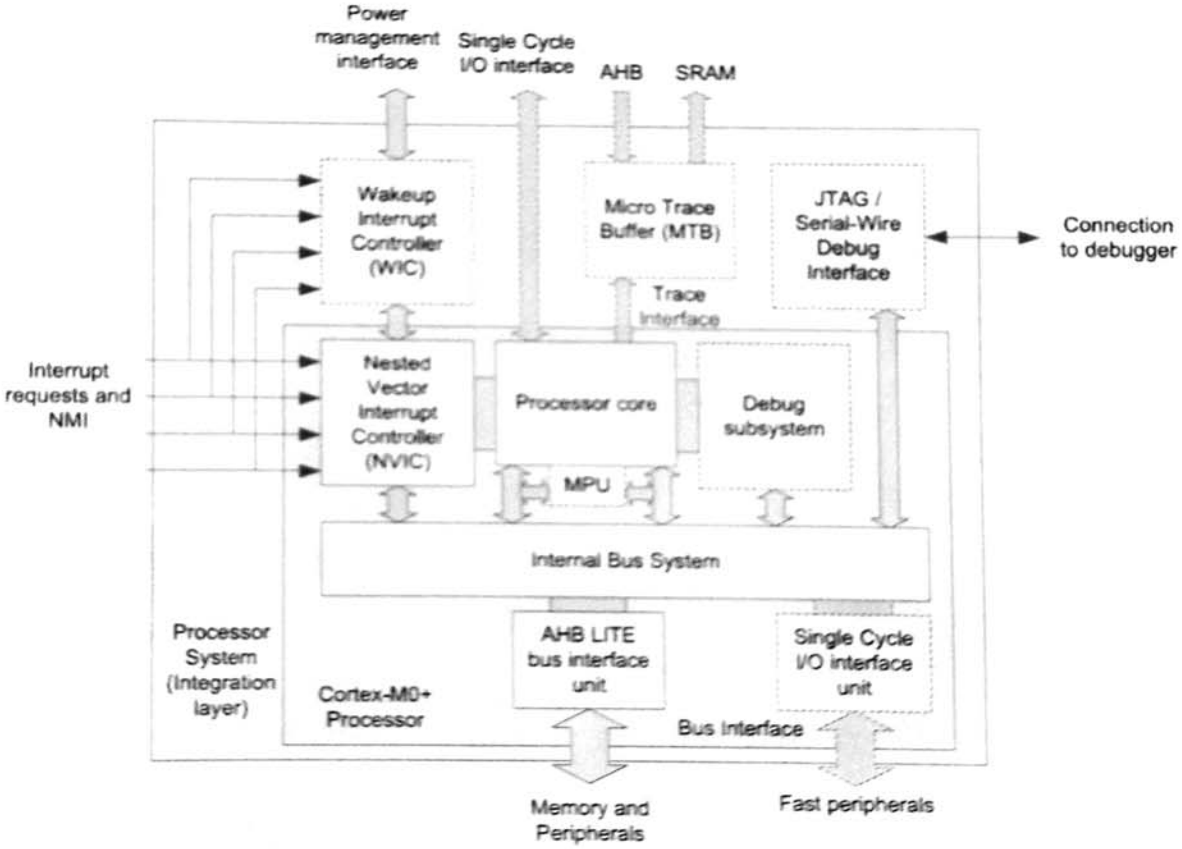

Power management interface
Single Cycle I/O interface
AHB
SRAM
Wakeup Interrupt Controller (WIC)
Micro Trace Buffer (MTB)
JTAG / Serial-Wire Debug Interface
Connection to debugger
Trace Interface
Interrupt requests and NMI
Nested Vector Interrupt Controller (NVIC)
Processor core
Debug subsystem
MPU
Internal Bus System
Processor System (Integration layer)
Cortex-M0+ Processor
AHB LITE bus interface unit
Single Cycle I/O interface unit
Bus Interface
Memory and Peripherals
Fast peripherals

Figure 2.3

A simplified block diagram of the Cortex®-M0+ processor.

The MPU is a programmable device used to define access permission of the memory map. In some of the applications where an OS is used, application tasks can be executed with an unprivileged execution level with restrict memory access defined by the MPU, which is programmed by the OS.

The single cycle I/O interface provides another bus interface with faster access compared to the AHB-Lite system bus (pipelined operation). The MTB is used to provide instruction trace.

In both Cortex-M0 and Cortex-M0+ processors, a number of components in the processors are optional. For example, the debug support, MPU and the WIC are all optional. Some other components like the NVIC are configurable: allowing chip designers to define the features available, for example, the number of interrupt requests (IRQ).

2.3 Typical Systems

As you can see from the block diagrams, the Cortex®-M0 and Cortex-M0+ processors do not contain memories and peripherals. Chip designers need to add these components to the designs. As a result, different Cortex-M processor-based microcontrollers can have different memory sizes, address map, peripherals, interrupt assignment, etc.

In a simple microcontroller design based on a Cortex-M processor, the design would consist of the following:

- A memory for program code storage, usually a Read-Only-Memory (ROM) component, or reprogrammable memory technologies such as flash memory.
- A read–write memory for data (including variables, stack, etc.), usually based on Static Random Access Memory (SRAM).
- Various types of peripherals.
- Bus infrastructure components for joining the processor to all the memories and peripherals.

In some cases, there can also be a separate ROM device with boot code to boot up the microcontroller before the program in the user flash is executed. This is typically called boot ROM or boot loader.

For a simple design with Cortex-M0 processor, the design could look like the one shown in Figure 2.4.

A typical design based on the Cortex-M0 processor might partition the bus system into two parts, which are as follows:

- System bus connected to the memories including ROM, flash memory (for user program storage), the SRAM, a few number of peripherals, and a bus bridge to the peripheral bus system.

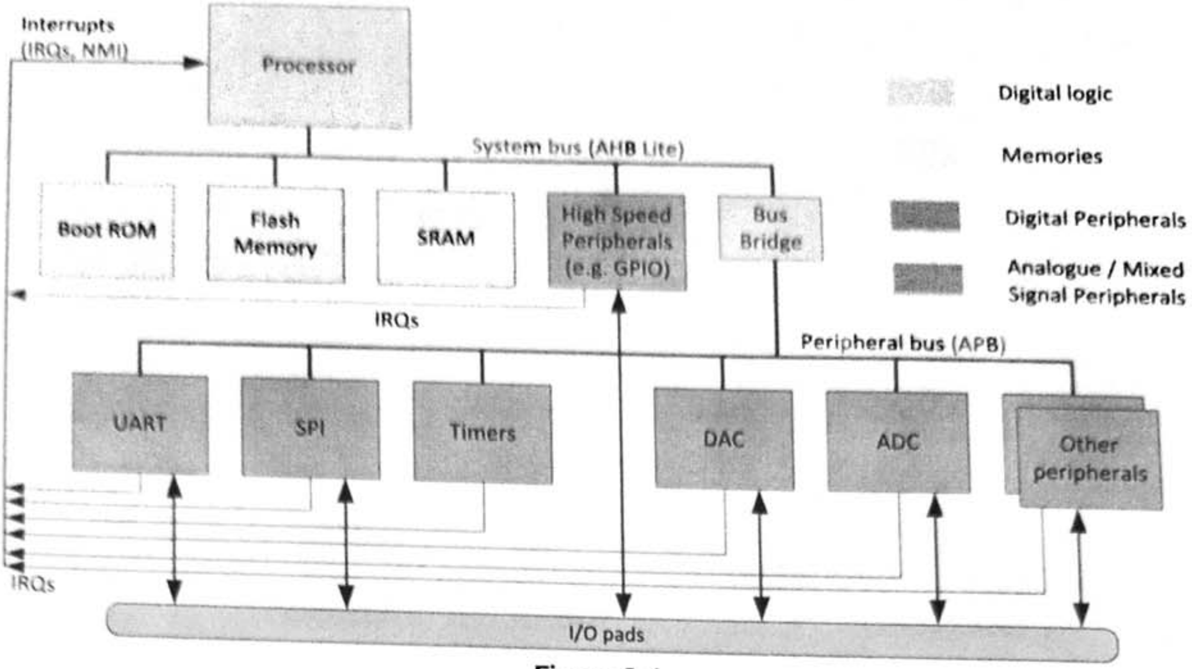


Figure 2.4
A simple system with the Cortex®-M0 Processor.

- The peripherals are connected to the peripheral bus, which might have a different operating frequency compared to the system bus.

It is quite common for some of the peripherals to be connected to a separated peripheral bus, which is linked to the main system bus via a bus bridge. This bus protocol for the peripheral bus is typically based on APB, which is a bus protocol defined in the AMBA®.

The uses of a separated APB peripheral bus are as follows:

- Allows lower hardware cost because the APB protocol (non-pipelined operations) is simpler than AHB-Lite (pipelined operations)
- Allows the peripheral bus to run at a different clock frequency than the main system bus
- Avoids large combinational logic in the bus infrastructure for the main system bus, which could become the bottle neck in terms of getting to get high operating frequency. Many peripherals might present in a microcontroller designs and the bus fabric for peripherals can become quite large.

Another group of important connections are the interrupts—A number of peripherals can generate interrupt requests, including the General Purpose Input/Output (GPIO) modules. In most microcontroller designs, external devices connected to certain GPIO pins can generate interrupt request to the processor via some additional conditioning and synchronization logic.

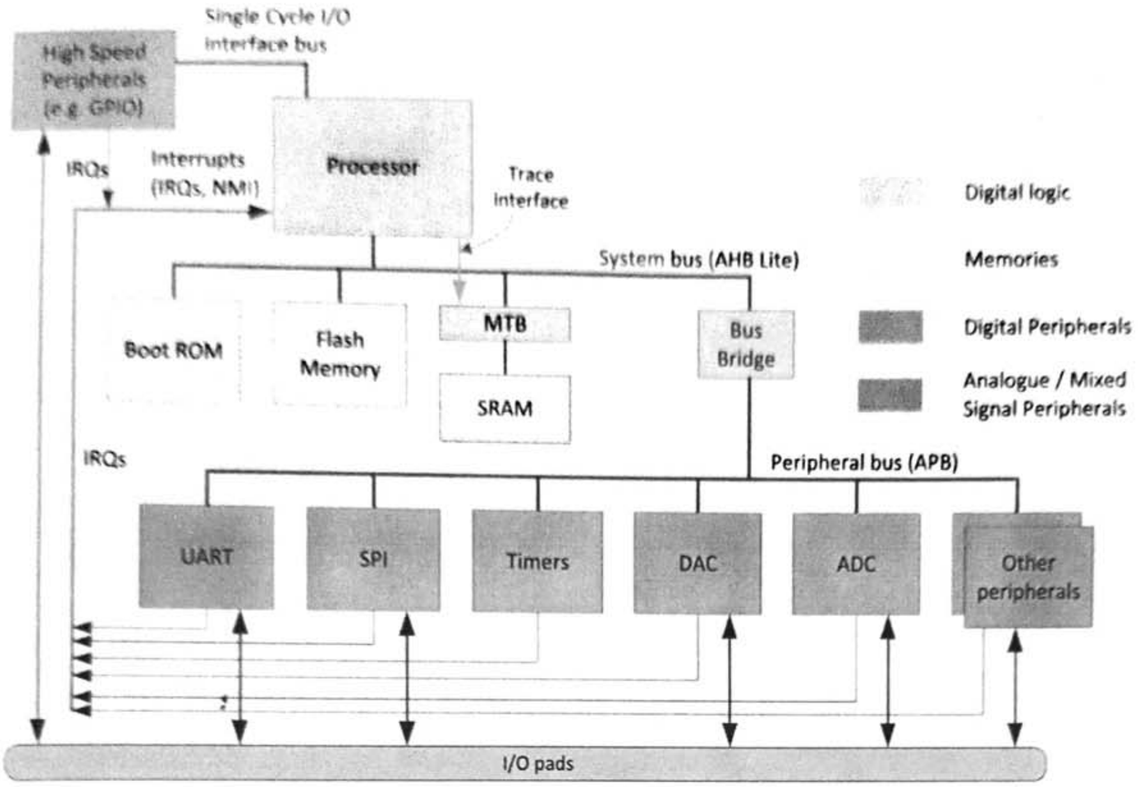


Figure 2.5
A simple system with the Cortex®-M0+ Processor.

For a system based on the Cortex-M0+ processor, the system design can be very similar, like the one shown in Figure 2.5.

In this design, the high-speed peripherals are moved to the single cycle I/O interface bus for faster I/O performance, and the MTB is added between the AHB-Lite system bus and the SRAM for support instruction trace capture.

Potentially the processor might not be the only component in the system that can generate bus transactions. In many microcontroller products, there is also a component called Direct Memory Access (DMA) controller. Once programmed, the DMA controller can carry out memory accesses on requests from peripherals without processor intervention (Figure 2.6)

The DMA controller can perform data transfers between memory and peripherals, or between memories (e.g., to accelerate memory copy). This is commonly needed for microcontrollers with high bandwidth communication interface like Ethernet or USB. However, it can also benefit some low-power applications, for example, by avoiding waking up the processor from sleep mode to collect small amount of data from peripherals.

mode, with privileged access level by default.

4.2.2 Registers and Special Registers

In order to perform data processing and controls, a number of registers are required inside the processor core. If data from memory is to be processed, it has to be loaded from the memory to a register in the register bank, processed inside the processor, and then written back to the memory if needed, or kept in the register bank for another operation. This is commonly called "load-store architecture." By having a sufficient number of registers in the register bank, this mechanism is easy to use, and is C-friendly. It is easy for C compilers to compile a C program into machine code with good performance.

The Cortex-M0 and Cortex-M0+ processor provides a register bank of 16 32-bit registers (most are general purposed, R13–R15 has special purposes), and a number of special registers (Figure 4.3).

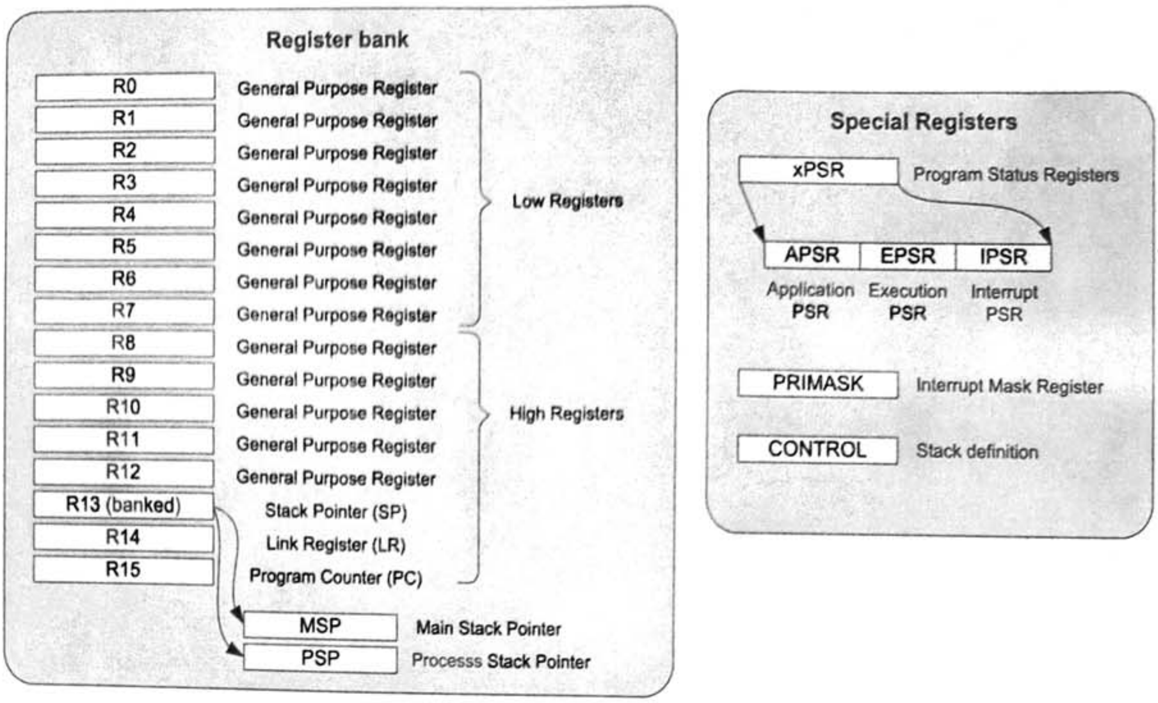


Figure 4.3
Registers in the Cortex®-M0 and Cortex-M0+ processors.

The detailed descriptions for these registers are as follows:

R0–R12

Registers R0–R12 are for general uses. Due to the limited space in the 16-bit Thumb® instructions, many of the Thumb instructions can only access R0–R7, which are also called the low registers. While some instructions, like MOV (move), can be used on all registers. When using these registers with ARM® development tools such as the ARM assembler, you can use either upper case (e.g., R0) or lower case (e.g., r0) to specify the register to be used. The initial values of R0–R12 at reset are undefined.

R13, Stack Pointer

R13 is the Stack Pointer. It is used for accessing the stack memory via PUSH and POP operations. There are physically two different stack pointers in Cortex-M0 and Cortex-M0+ Processors.

- The Main Stack Pointer (MSP, or SP_main in ARM documentation) is the default Stack Pointer after reset, and is used when running exception handlers.
- The Process Stack Pointer (PSP, or SP_process in ARM documentation) can only be used in Thread mode (when not handling exceptions).

The stack pointer selection is determined by the CONTROL register, one of the special registers which will be introduced later (*CONTROL—Special Register*).

When using ARM development tools, you can access the stack pointer using either "R13" or "SP." Both upper case and lower case (e.g., "r13" or "sp") can be used. Only one of the stack pointers is visible at a given time. However, you can access to the MSP or PSP directly when using the special register access instructions MRS and MSR. In such cases, the register names "MSP" or "PSP" should be used.

The lowest 2 bits of the stack pointers are always zero and writes to these 2 bits are ignored. In ARM processors, PUSH and POP are always 32-bit accesses because the registers are 32-bit, and the transfers in stack operations must be aligned to a 32-bit word boundary. The initial value of MSP is loaded from the first 32-bit word of the vector table from the program memory during the start-up sequence. The initial value of PSP is undefined.

It is not necessary to use the PSP. In many applications, the system can completely rely on the MSP. The PSP is normally used in designs with an OS, where the stack memory for OS Kernel and the thread-level application codes must be separated.

R14, Link Register

R14 is the Link Register (LR). The LR is used for storing the return address of a subroutine or function call. When BL or BLX is executed, the return address is stored in LR. At the end of the subroutine or function, the return address stored in LR is loaded into the program counter (PC) so that the execution of the calling program can be resumed. In the case where an exception occurs, the LR also provides a special code value which is used by the exception return mechanism. When using ARM development tools, you can access to the LR using either "R14" or "LR." Both upper and lower case (e.g., "r14" or "lr") can be used.

Although the return address in the Cortex-M0/M0+ processor is always an even address (bit[0] is zero because smallest instruction are 16-bit and must be half-word aligned), bit zero of LR is readable and writeable. In the ARMv6-M architecture, some instructions require bit zero of a function address set to 1 to indicate Thumb state.

R15, Program Counter

R15 is the PC. It is readable and writeable. A read returns the current instruction address plus four (this is caused by the pipeline nature of the design). Writing to R15 will cause a branch to take place (but unlike a function call, the LR does not get updated).

In the ARM assembler, you can access the PC using either "R15" or "PC," in either upper or lower case (e.g., "r15" or "pc"). Instruction addresses in the Cortex-M0/M0+ processor must be aligned to half-word address, which means the actual bit zero of the PC should be zero all the time. However, when attempting to carry out a branch using the branch instructions (BX or BLX), the LSB of the PC should be set to1.[1] This is to indicate that

[1] Not required when a move (MOV) or add (ADD) instruction is used to modify the PC.

the branch target is a Thumb program region. Otherwise, it can imply an attempt to switch the processor to ARM state (depending on the instruction used), which is not supported and will cause a fault exception.

xPSR, Combined Program Status Register

The combined Program Status Register (PSR) provides information about program execution and the ALU flags. It consists of the following three PSRs (Figure 4.4):

- Application PSR (APSR),
- Interrupt PSR (IPSR), and
- Execution PSR (EPSR)

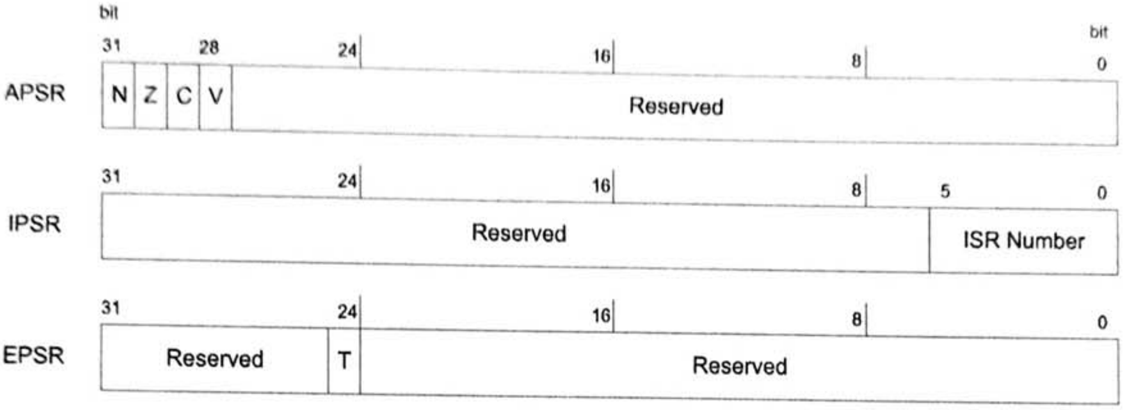


Figure 4.4
Application PSR (APSR), Interrupt PSR (IPSR), and Execution PSR (EPSR).

The APSR contains the ALU flags: N (negative flag), Z (zero flag), C (carry or borrow flag), and V (overflow flag). These bits are at the top 4 bits of the APSR. The common use of these flags is to control conditional branches.

The IPSR contains the current executing ISR (Interrupt Service Routine) number. Each exception on the Cortex-M0/M0+ processor has a unique associated ISR number (exception type). This is useful for identifying the current interrupt type during debugging and allows an exception handler that is shared by several exceptions to know which exception it is serving.

The EPSR on the Cortex-M0/M0+ processor contains the T bit which indicates that the processor is in the Thumb state. On the Cortex-M0/M0+ processor, this bit is normally set to 1 because the Cortex-M processors only support Thumb state. If this bit is cleared, a HardFault exception will be generated in the next instruction execution.

These three registers can be accessed as one register called xPSR. For example, when an interrupt takes place, the xPSR is one of the registers that is stored on to the stack memory automatically and restored automatically after returning from an exception. During the stack store and restore, the xPSR is treated as one register (Figure 4.5).

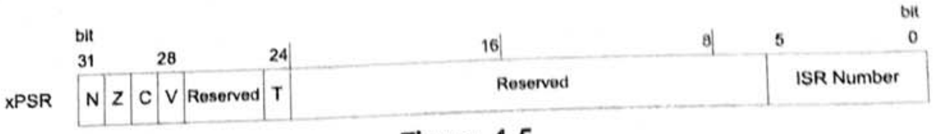


Figure 4.5
xPSR.

Direct access to the PSRs is only possible through special register access instructions. However, the value of the APSR can affect conditional branches and the carry flag in the APSR can also be used in some data processing instructions.

PRIMASK—Interrupt Mask Special Register

The PRIMASK register is a 1-bit wide interrupt mask register. When set, it blocks all interrupts apart from the Non-Maskable Interrupt (NMI) and the HardFault exception. Effectively it raises the current interrupt priority level to 0 which is the highest value for a programmable exception (Figure 4.6).

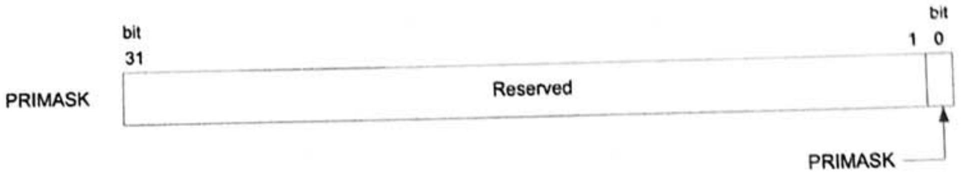


Figure 4.6
PRIMASK.

The PRIMASK register can be accessed using special register access instructions (MSR, MRS) as well as using an instruction called CPS. This is commonly used for handling time critical routines.

CONTROL—Special Register

As mentioned earlier, there are two stack pointers in the Cortex-M0 and Cortex-M0+ processors. The stack pointer selection is determined by the processor mode as well as the configuration of the CONTROL register (bit 1—SPSEL). The Thread mode of the Cortex-M0+ processor can either be privileged or unprivileged, and this is also controlled by CONTROL (bit 0—nPRIV) (Figure 4.7).

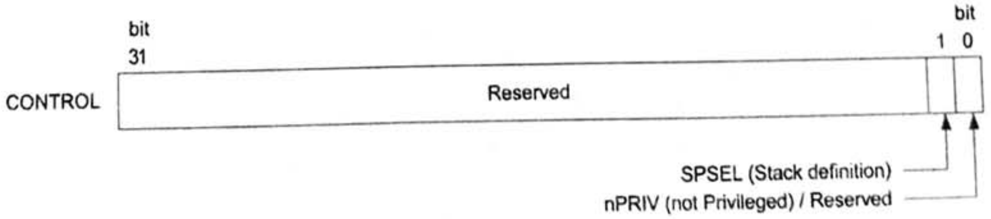


Figure 4.7
CONTROL.

After reset, the MSP is used, but can be switched to the PSP in Thread mode (when not running an exception handler) by setting bit[1] in the CONTROL register. During running of an exception handler (when the processor is in handler mode), only the MSP is used, and the CONTROL register reads as zero. The bit[1] of CONTROL register can only be changed in Thread mode, or via the exception entrance and return mechanism (Figure 4.8).

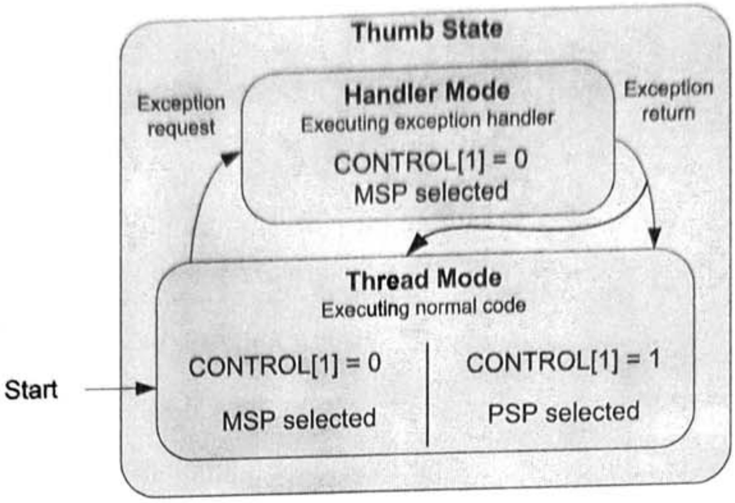


Figure 4.8
Stack pointer selection.

Bit[0] of the CONTROL register is for selecting between Privileged and Unprivileged states during Thread mode. Some of the Cortex-M0+ devices and all Cortex-M0 processor-based devices do not support unprivileged state and therefore this bit is always zero (Figure 4.9).

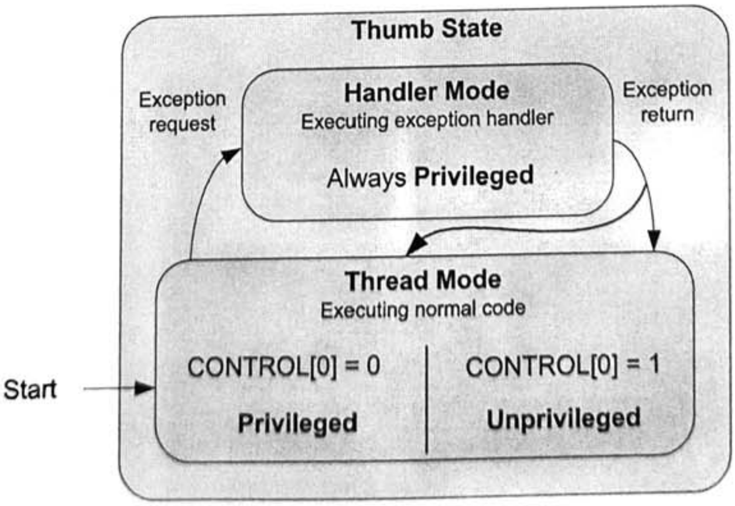


Figure 4.9
Privileged state selection.

Access of Registers and Special Registers

In C/C++ programming or any other high level languages, the registers in the register bank (R0–R12) can be utilized by the compiler automatically. In most cases, you do not need to worry about which registers being used, unless you are interfacing assembly code and C/C++ code (such mixed language development will be cover in Chapter 21).

The other special registers need to be accessed using some special instructions (MRS and MSR). The CMSIS-CORE provides a number of APIs for such usages. However, please note that some of these special registers cannot be accessed or changed by software (Table 4.1).

Table 4.1: Access limitations to special registers

	Privileged	Unprivileged
APSR	R/W	R/W
EPSR	No access (T bit read as zero)	No access (T bit read as zero)
IPSR	Read only	Read only
PRIMASK	R/W	Read only
CONTROL	R/W	Read only

4.2.3 Behaviors of the APSR

Data processing instructions can affect destination registers as well as the APSR which is commonly known as ALU status flags in other processor architectures. The APSR is essential for controlling conditional branches. In addition, one of the APSR flags, the C (Carry) bit, can also be used in add and subtract operations.

There are four APSR flags in the Cortex-M0 ad Cortex-M0+ processors (Table 4.2).

Table 4.2: ALU flags on the Cortex®-M0 and Cortex-M0+ processors

Flag	Descriptions
N (bit 31)	Set to bit[31] of the result of the executed instruction. When it is "1," the result has a negative value (when interpreted as a signed integer). When it is "0," the result has a positive value or equal zero.
Z (bit 30)	Set to "1" if the result of the executed instruction is zero. It can also be set to "1" after a compare instruction is executed if the two values are the same.
C (bit 29)	Carry flag of the result. For unsigned addition, this bit is set to "1" if an unsigned overflow occurred. For unsigned subtract operations, this bit is the inverse of the borrow output status.
V (bit 28)	Overflow of the result. For signed addition or subtraction, this bit is set to "1" if a signed overflow occurred.

A few examples of the ALU flag results are as given in Table 4.3.

Table 4.3: ALU flags operation examples

Operation	Results, flags
0x70000000 + 0x70000000	Result = 0xE0000000, N = 1, Z = 0, C = 0, V = 1
0x90000000 + 0x90000000	Result = 0x20000000, N = 0, Z = 0, C = 1, V = 1
0x80000000 + 0x80000000	Result = 0x00000000, N = 0, Z = 1, C = 1, V = 1
0x00001234 − 0x00001000	Result = 0x00000234, N = 0, Z = 0, C = 1, V = 0
0x00000004 − 0x00000005	Result = 0xFFFFFFFF, N = 1, Z = 0, C = 0, V = 0
0xFFFFFFFF − 0xFFFFFFFC	Result = 0x00000003, N = 0, Z = 0, C = 1, V = 0
0x80000005 − 0x80000004	Result = 0x00000001, N = 0, Z = 0, C = 1, V = 0
0x70000000 − 0xF0000000	Result = 0x80000000, N = 1, Z = 0, C = 0, V = 1
0xA0000000 − 0xA0000000	Result = 0x00000000, N = 0, Z = 1, C = 1, V = 0

In the Cortex-M0 and Cortex-M0+ processors, almost all of the data processing instructions modify the APSR; however, some of these instructions do not update the V flag or the C flag. For example, the MULS (multiply) instruction only changes the N flag and the Z flag.

The ALU flags can be used for handling data that is larger than 32-bits. For example, we can perform a 64-bit addition by splitting the operation into two 32-bit additions. The pseudo form of the operation can be written as follows:

```
// Calculating Z = X + Y, where X, Y and Z are all 64-bit
Z[31:0] = X[31:0] + Y[31:0];        // Calculate lower word addition,
                                    // carry flag get updated
Z[63:32] = X[63:32] + Y[63:32] + Carry;  // Calculate upper word addition.
```

An example of carry out such 64-bit add operation in assembly code can be found in Chapter 6 (Section 6.5.1).

The other common usage of APSR flag is to control branching. More on this will be covered in Chapter 5 (Section 5.4.8), where the details of the condition branch instruction will be covered.

4.3 Memory System

4.3.1 Overview

All ARM® Cortex®-M processors have a 4 GB of memory address space. The memory space is architecturally defined into a number of regions, with each region having a recommended usage to help software porting between different devices (Figure 4.10).

The Cortex-M0 and Cortex-M0+ processors contain a number of built-in components like the NVIC (the interrupt controller) and a number of debug components. These are located in fixed memory locations within the system region of the memory map. As a result, all the devices based on the Cortex-M processors have the same programming model for interrupt control and debug. This makes it convenient for software porting as well as helping debug

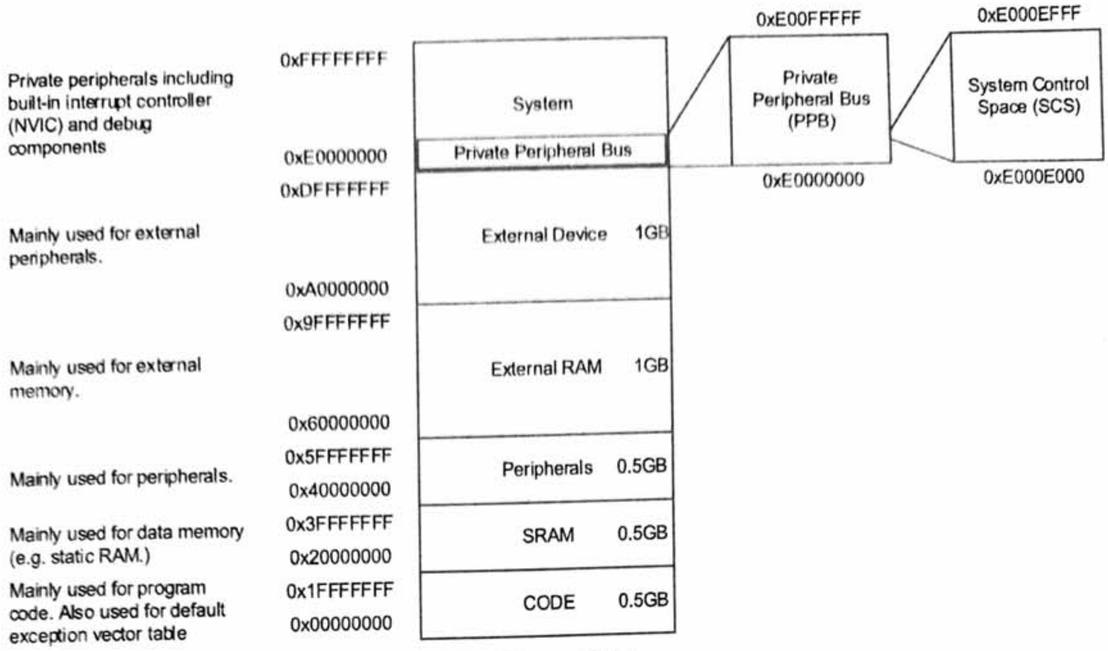

0xE00FFFFF
0xE000EFFF
Private peripherals including built-in interrupt controller (NVIC) and debug components
0xFFFFFFFF
System
Private Peripheral Bus (PPB)
System Control Space (SCS)
0xE0000000
Private Peripheral Bus
0xE0000000
0xE000E000
0xDFFFFFFF
Mainly used for external peripherals.
External Device 1GB
0xA0000000
0x9FFFFFFF
Mainly used for external memory.
External RAM 1GB
0x60000000
0x5FFFFFFF
Mainly used for peripherals.
Peripherals 0.5GB
0x40000000
0x3FFFFFFF
Mainly used for data memory (e.g. static RAM.)
SRAM 0.5GB
0x20000000
Mainly used for program code. Also used for default exception vector table
0x1FFFFFFF
CODE 0.5GB
0x00000000

Figure 4.10
Memory map.

tool vendors to develop debug solutions for the Cortex-M0-based microcontroller or System-on-Chip (SoC) products.

The memory space is shared between instruction memory, data memory, peripherals processor's built-in peripherals (e.g., the interrupt controller), and processor's debug components. However, the debug components are not visible to the software running on the processor (from architecture point of view this is implementation defined, and existing Cortex-M0 and Cortex-M0+ processors are designed to make the debug components to be visible only from debugger). This is different from Cortex-M3, Cortex-M4, and Cortex-M7 processors, where privileged codes can access the debug components.

In most cases, the memories connected to the Cortex-M processors are 32-bits, but it is also possible to connect memory of different data widths to a Cortex-M processor with suitable memory interface hardware. The memory system in Cortex-M processors supports memory transfers of different sizes such as byte (8-bit), half word (16-bit), and word (32-bit). The Cortex-M0 and Cortex-M0+ processor designs can be configured to support either little endian or big endian memory systems, but cannot switch from one to another in an implemented design.

Since the memory system and peripherals connected to the Cortex-M0 or Cortex-M0+ processors are developed by microcontroller vendors or SoC designers, different memory sizes and memory types can be found in different Cortex-M0/M0+-based products.