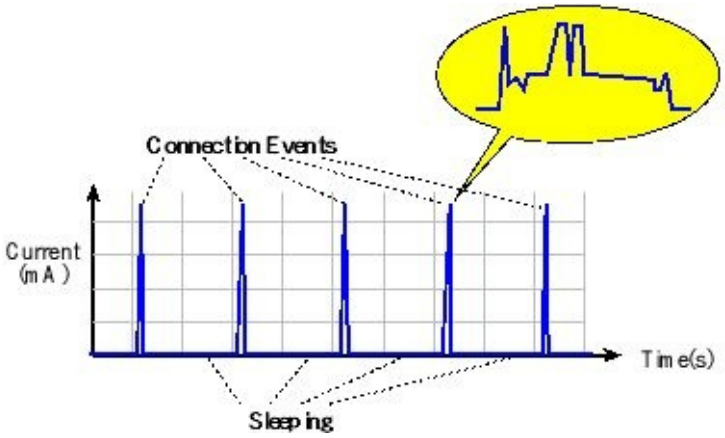


Figure 9.30. CC2650 current verses time, showing the connection events.

For example, you will see the advertising interval settings as parametersin the **NPI_StartAdvertisement** message. In particular, the example projects set the advertising interval to 62.5ms.

9.4.2. Client-server Paradigm

The **client-server paradigm** is the dominant communication pattern for network protocols, see Figure 9.31. In general, the embedded system will be the server, and the smart phone will be the client. The client can request information from the server, or the client can send data to the server. With Bluetooth this exchange of data is managed by the services and profiles, discussed in the next section. There are four main profile types.

A **peripheral device** has sensors and actuators. On startup it advertises as connectable, and once connected it acts as a slave. In general, the embedded device will be a peripheral.

A **central device** has intelligence to manage the system. On startup it scans for advertisements and initiates connections. Once connected it acts as the master. In general, the smart phone in will be a central device.

A **broadcaster** has sensors collecting information that is generally relevant. On startup it advertises but is not connectable. Other devices in the vicinity can read this information even though they cannot connect to the broadcaster. An example is a thermometer.

An **observer** can scan for advertisements but cannot initiate a connection. An example is a temperature display device that shows temperatures measured by broadcasters.

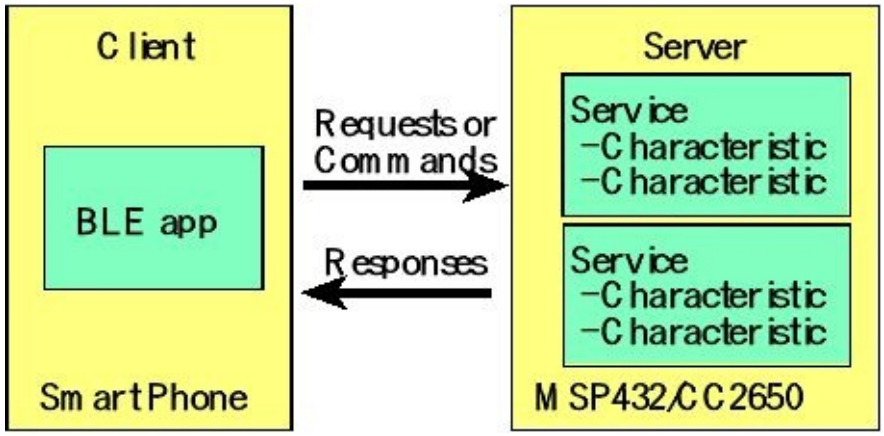


Figure 9.31. Client-server Paradigm.

Read indication. When the client wishes to know the value of a characteristic, it will issue a read indication. Inside the request will be a **universally unique identifier** (UUID) that specifies which characteristic is desired. The server will respond with the value by returning a **read confirmation**. The data may be one or more bytes. For large amounts of data, the response could be broken into multiple messages. In the example projects, the data will be 1, 2 or 4 bytes long. The size of the data is determined during initialization as the characteristic is configured.

Write indication. When the client wishes to set the value of a characteristic, it will issue a write indication. This request will include data. The request will also include a UUID that specifies to which characteristic the data should be written. The server will respond with an acknowledgement, called a **write confirmation**.

Notify request. When the client wishes to keep up to data on a certain value in the server, it will issue a notify request. The request includes a UUID. The server will respond with an acknowledgement, and then the server will stream data. This streaming could occur periodically, or it could occur whenever the value changes. In the example projects, **notify indication** messages are sent from server to client periodically. The client can start notification (listen command on the phone) or stop notifications.

9.5. CC2650 Solutions

9.5.1. CC2650 Microcontroller

There are three controllers on the CC2650: a main CPU, an RF core, and a sensor controller. Together, these combine to create a one-chip solution for Bluetooth applications. The **main CPU** includes 128kB of flash, 20kB of SRAM, and a full range of peripherals. Typically, the ARM Cortex-M3 processor handles the application layer and BLE protocol stack. However, in this chapter, we will place the application layer on another processor and use the CC2560 just to implement Bluetooth.

The **RF Core** contains an ARM Cortex-M0 processor that interfaces the analog RF and base-band circuitries, handles data to and from the system side, and assembles the information bits in a given packet structure. The RF core offers a high level, command-based API to the main CPU. The RF core is capable of autonomously handling the time-critical aspects of the radio protocols (802.15.4 RF4CE and ZigBee, Bluetooth Low Energy) thus offloading the main CPU and leaving more resources for the user application. The RF core has its own RAM and ROM. The ARM Cortex-M0 ROM is not programmable by customers. The basic circuit implementing the 2.4 GHz antenna is shown in Figure 9.32.

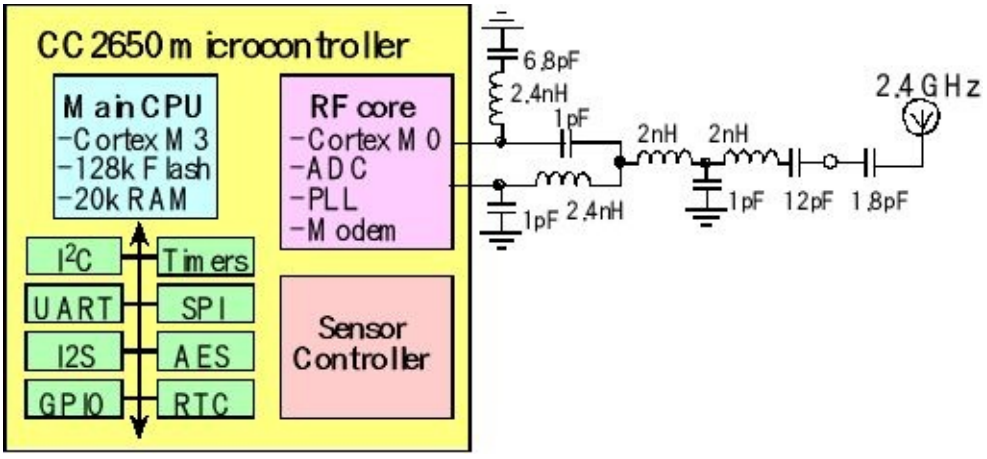


Figure 9.32. The CC2650 includes a main CPU, a suite of I/O devices, an RF core, and a sensor controller.

The **Sensor Controller** block provides additional flexibility by allowing autonomous data acquisition and control independent of the main CPU, further extending the low-power capabilities of the CC2650. The Sensor Controller is set up using a PC-based configuration tool, called Sensor Controller Studio, and example interfaces include:

- Analog sensors using integrated ADC

- Digital sensors using GPIOs, bit-banged I2C, and SPI
- UART communication for sensor reading or debugging
- Capacitive sensing
- Waveform generation
- Pulse counting
- Keyboard scan
- Quadrature decoder for polling rotation sensors
- Oscillator calibration

The CC2650 uses a radio-frequency (RF) link to implement Bluetooth Low Energy (BLE). As illustrated in Figure 9.33, the CC2650 can be used as a bridge between any microcontroller and Bluetooth. It is a transceiver, meaning data can flow across the link in both directions.

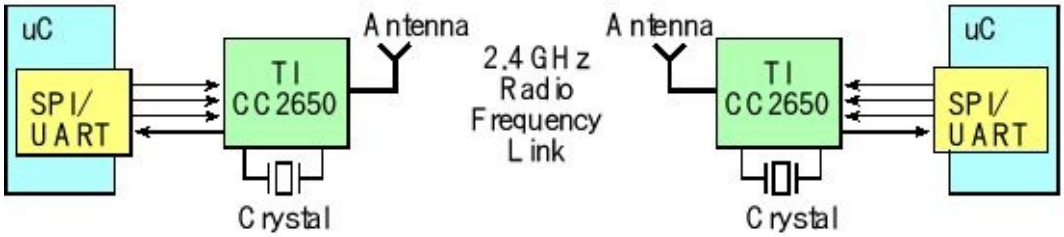


Figure 9.33. Block diagram of a wireless link between two microcontroller systems.

Figure 9.34 shows a CC2650 BoosterPack. This board comes preprogrammed with the simple network processor described in the next section. With a JTAG debugger, other programs can be loaded onto this CC2650. For more information, see

http://www.ti.com/tool/boostxl-cc2650ma

Figure 9.34. CC2650 BoosterPack (BOOSTXL-CC2650MA).

Figure 9.35 shows a CC2650 LaunchPad. The top part of the PCB is the debugger and the bottom part implements the CC2650 target system. To see the pin connections, see

http://www.ti.com/ww/en/launchpad/launchpads-connected-launchxl-cc2650.html

Figure 9.35. CC2650 LaunchPad (LAUNCHXL-CC2650).

9.5.2. Single Chip Solution, CC2650 LaunchPad

The CC2650 microcontroller is a complete System-on-Chip (SoC) Bluetooth solution, as shown in Figure 9.36. One could deploy the application, the Bluetooth stack, and the RF radio onto the CC2650.

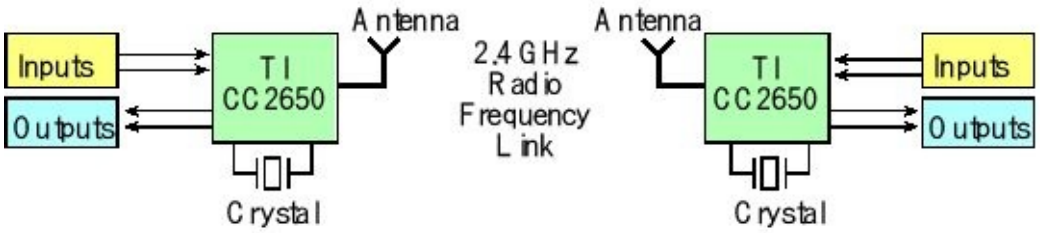


Figure 9.36. Block diagram of a wireless link between two single-chip embedded systems.

9.6. Network Processor Interface (NPI)

9.6.1. Overview

Simple Network Processor (SNP) is TI's name for the application that runs on the CC2650 when using the CC2650 with another microcontroller such as the MSP432 or TM4C123. In this configuration the controller and host are implemented together on the CC2650, while the profiles and application are implemented on an external MCU. The application and profiles communicate with the CC2650 via the **Application Programming Interface** (API) that simplifies the management of the BLE network processor. The SNP API communicates with the BLE device using the **Network Protocol Interface** (NPI) over a serial (SPI or UART) connection. In this chapter, we will use a UART interface as shown in Figure 9.37. This configuration is useful for applications that wish to add Bluetooth functionality to an existing device. In this paradigm, the application runs on the existing microcontroller, and BLE runs on the CC2650. For a description of the Simple Network Processor, refer to

SNP http://processors.wiki.ti.com/index.php/CC2640_BLE_Network_Processor
Developer guide http://www.ti.com/lit/ug/swru393c/swru393c.pdf
TI wiki page http://processors.wiki.ti.com/index.php/NPI

In this chapter, our TM4C123/MSP432 LaunchPad will be the application processor (AP) and the CC2650 will be the network processor (NP). There are 7 wires between the AP and the NP. Two wires are power and ground, one wire is a negative logic reset, two wires are handshake lines, and two wires are UART transmit and receive.

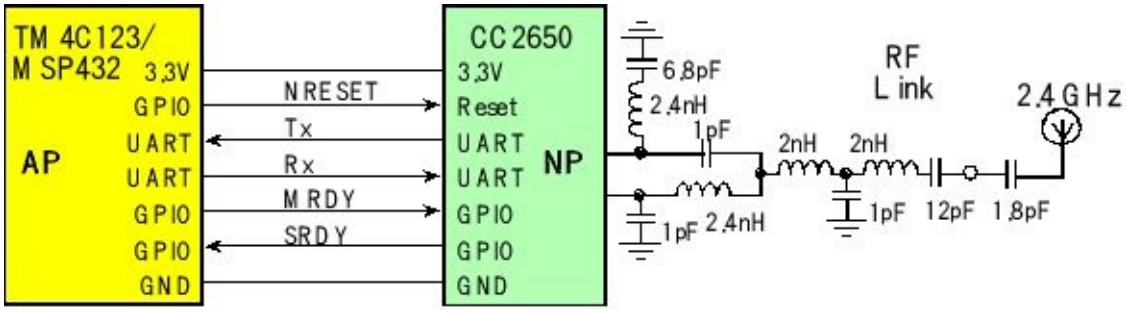


Figure 9.37. Hardware interface between the LaunchPad AP and the CC2650 NP.

To initialize Bluetooth, the master (AP) first resets the slave (NP). The **reset** line is a GPIO output of the AP and is the hardware reset line on the NP. There are two handshake lines: master ready and slave ready. **Master ready** (MRDY) is a GPIO output of the AP and a GPIO input to the NP. **Slave ready** (SRDY) is a GPIO output of the NP and a GPIO input of the AP. If the AP wishes to reset the NP, it sets MRDY

high and pulses reset low for 10 ms, Figure 9.38. Normally, the reset operation occurs once, and thereafter the reset line should remain high.

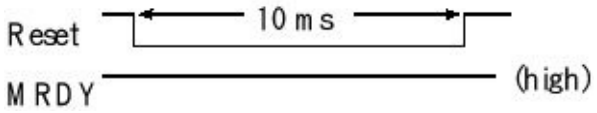


Figure 9.38. The LaunchPad AP can reset the CC2650 NP.

There are two types of communication. Messages can be sent from master to slave, or from slave to master. If the master (AP) wishes to send a message to the slave (NP), it follows 5 steps, Figure 9.39. First, the master sets MRDY low (Master: "I wish to send"). Second, the slave responds with SRDY low (Slave: "ok, I am ready"). The communication is **handshaked** because the master will wait for SRDY to go low. Third, the master will transmit a message on its UART output (Rx input to slave). The format of this message will be described later. Fourth, after the message has been sent, the master pulls MRDY high (Master: "I am done"). Fifth, the slave pulls its SRDY high (Slave: "ok"). Again, the handshaking requires the master to wait for SRDY to go high.

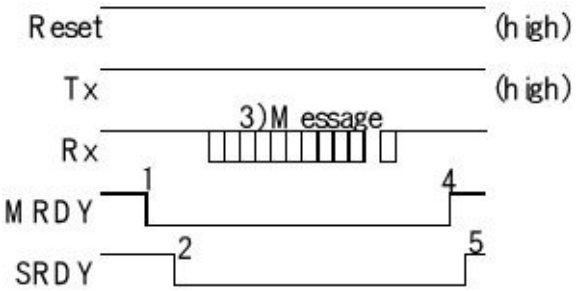


Figure 9.39. The LaunchPad AP can send a message to the CC2650 NP. Handshake means the steps 1 – 5 always occur in this sequence.

If the slave (NP) wishes to send a message to the master (AP), there are also 5 steps, Figure 9.40. First, the slave sets SRDY low (Slave: "I wish to send"). Second, the master responds with MRDY low (Master: "ok, I am ready"). You will notice in the example projects that the master will periodically check to see if the SRDY line has gone low, and if so it will receive a message. Third, the slave will transmit a message on its UART output (Tx output from slave). The format of this message will be the same for all messages. Fourth, after the message has been sent, the slave pulls SRDY high (Slave: "I am done"). The master will wait for SRDY to go high. Fifth, the master pulls its MRDY high (Master: "ok").

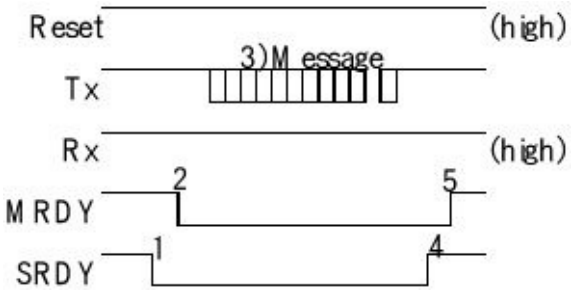


Figure 9.40. The CC2650 NP can send a message to the LaunchPad AP. Handshake means the steps 1 – 5 always occur in this sequence.

The format of the message is shown in Figure 9.41. The boxes in the figure represent UART frames. Each UART frame contains 1 start bit, 8 data bits, and 1 stop bit, sent at 115,200 bits/sec. All messages begin with a *start of frame* (SOF), which is a 254 (0xFE). The next two bytes are the payload length in little endian format. Since all the payloads in this chapter are less than 256 bytes, the second byte is the length, *L*, and the third byte is 0. The fourth and fifth bytes are the command. Most commands have a payload, which contains the parameters of the command. Some commands do not have a payload. All messages end with a **frame check sequence** (FCS). The FCS is the 8-bit exclusive or of all the data, not including the SOF and the FCS itself.

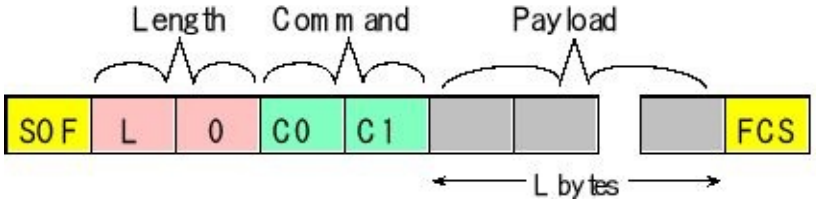


Figure 9.41. The format of an NPI message.

The following steps occur in this order

1. Initialize GATT (add services, characteristics, CCCD's);
2. Initialize GAP (advertisement data, connection parameters);
3. Advertise and optionally wait for a connection;
4. Respond to GATT requests and send notifications / indications as desired.

9.6.2. Services and Characteristics

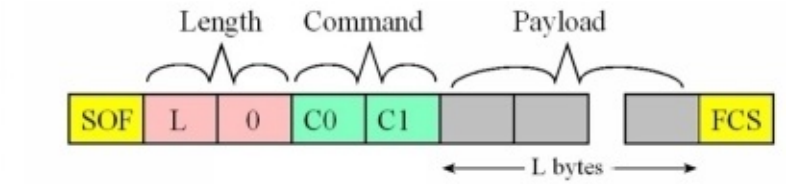
After the CC2650 is reset, the next step is to services and characteristics. In the example projects we will define one service with multiple characteristics. To create a service, the master first issues an **Add Service** command (0x35,0x81). For each characteristic, the master sends an **Add Characteristic Value** (0x35,0x82) and an **Add Characteristic Description** (0x35,0x83) message. Once all the characteristics are defined, the master sends a **Register Service** command (0x35,0x84). Each of the commands has an acknowledgement response. The debugger output for a service with one characteristic is shown in Figure 9.42. The detailed syntax of these messages can be found in the TI CC2640 Bluetooth low energy Simple Network Processor API Guide.

Add service
LP->SNP FE,03,00,35,81,01,F0,FF,B9
SNP->LP FE,01,00,75,81,00,F5
Add CharValue1
LP->SNP FE,08,00,35,82,03,0A,00,00,00,02,F1,FF,BA

SNP->LP FE,03,00,75,82,00,1E,00,EA
Add CharDescriptor1
LP->SNP FE,0B,00,35,83,80,01,05,00,05,00,44,61,74,61,00,0C
SNP->LP FE,04,00,75,83,00,80,1F,00,6D
Register service
LP->SNP FE,00,00,35,84,B1
SNP->LP FE,05,00,75,84,00,1C,00,29,00,C1

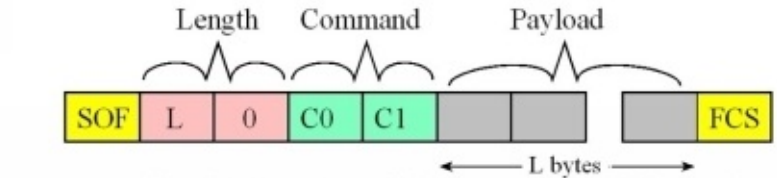
Figure 9.42. TExaSdisplay output as the device sets up a service with one characteristic. These data were collected running the VerySimpleApplicationProcessor_xxx project.

Figures 9.43 through 9.46 show the four messages used to define a service with one characteristic. The **add service** creates a service. The **add characteristic value declaration** defines the read/write/notify properties of a characteristic in that service. The response to this message includes the handle. The **add characteristic description declaration** defines the name of the characteristic. When we create services with multiple characteristics, we simply repeat the "add characteristic value" and "add characteristic description" declarations for each. The **register service** makes that service active.



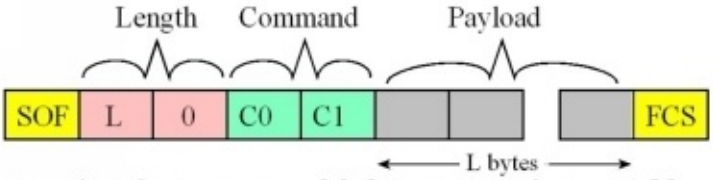

```
const uint8_t NPI_AddServiceMsg[] = {
  SOF,3,0x00,      // length = 3
  0x35,0x81,       // SNP Add Service
  0x01,            // Primary Service
  0xF0,0xFF,       // UUID
  0xB9};           // FCS
```

Figure 9.43. Add service message from the VerySimpleApplicationProcessor_xxx project.



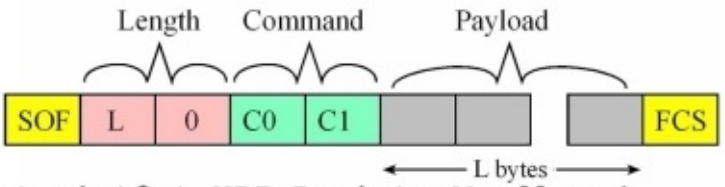

```
const uint8_t NPI_AddCharValue1[] = {
  SOF,0x08,0x00,   // length = 8
  0x35,0x82,       // Add Char Value Decl
  0x03,            // Read+Write Permission
  0x0A,0x00,       // Read+Write Properties
  0x00,            // RFU
  0x00,0x02,       // Maximum length
  0xF1,0xFF,       // UUID
  0xBA};           // FCS
```

Figure 9.44. Add characteristic value declaration message from the VerySimpleApplicationProcessor_xxx project.




```
const uint8_t NPI_AddCharDescriptor1[] = {
  SOF,11,0x00,     // length = 11
  0x35,0x83,       // Add Char Desc Decl
  0x80,            // User Description String
  0x01,            // GATT Read Permissions
  0x05,0x00,       // Maximum length
  0x05,0x00,       // Initial length
  'D','a','t','a',0, // description string
  0x0C};
```

Figure 9.45. Add characteristic declaration message from the VerySimpleApplicationProcessor_xxx project.




```
const uint8_t NPI_RegisterMsg[] = {
  SOF,0x00,0x00,   // length = 0
  0x35,0x84,       // SNP Register Service
  0x00};           // FCS
```

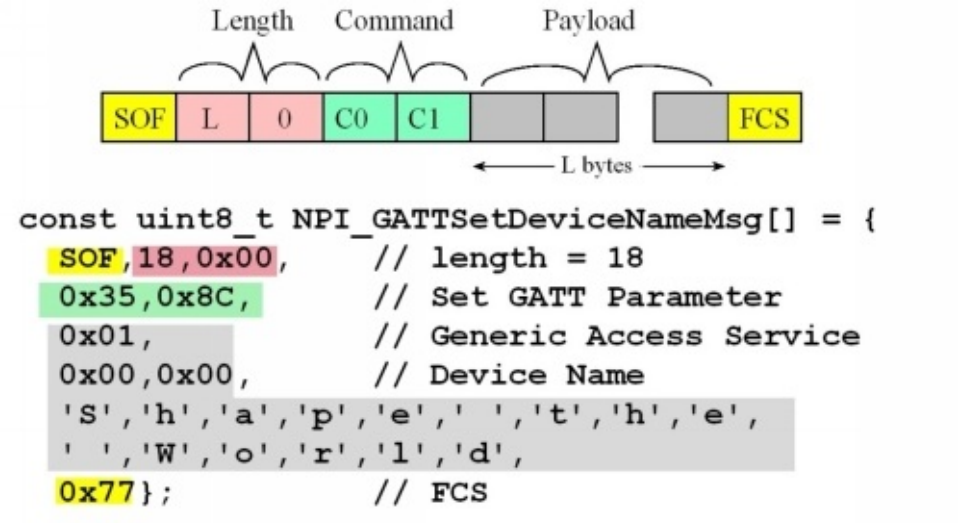
Figure 9.46. Register service message from the VerySimpleApplicationProcessor_xxx project.

9.6.3. Advertising

After all the services and characteristics are defined, the master will setup and initiate advertising. The master will send four messages to set up advertising. The debugger output for advertising is shown in Figure 9.47. Each message will be acknowledged by the NP. A 0x35,0x85 message will set the device name. There are two 0x55,0x43 messages to configure the parameters of the advertising. The 0x55,0x42 message will start advertising. Again, detailed syntax of these messages can be found in the TI CC2640 Bluetooth low energy Simple Network Processor API Guide. Figure 9.48 shows the C code to define a **Set Device Name** message.

GATT Set DeviceName
LP->SNP FE,12,00,35,8C,01,00,00,53,68,61,70,65,20,74,68,65,20,57,6F,72,6C,64,DE
SNP->LP FE,01,00,75,8C,00,F8
SetAdvertisement1
LP->SNP FE,0B,00,55,43,01,02,01,06,06,FF,0D,00,03,00,00,EE
SNP->LP FE,01,00,55,43,00,17
SetAdvertisement2
LP->SNP FE,1B,00,55,43,00,10,09,53,68,61,70,65,20,74,68,65,20,57,6F,...,00,0C
SNP->LP FE,01,00,55,43,00,17
StartAdvertisement
LP->SNP FE,0E,00,55,42,00,00,00,64,00,00,00,00,01,00,00,00,C5,02,BB
SNP->LP FE,03,00,55,05,08,00,00,5B

Figure 9.47. TExaSdisplay output as the device sets up advertising. These data were collected running the VerySimpleApplicationProcessor_xxx project.




```
const uint8_t NPI_GATTSetDeviceNameMsg[] = {
  SOF,18,0x00,      // length = 18
  0x35,0x8C,        // Set GATT Parameter
  0x01,             // Generic Access Service
  0x00,0x00,        // Device Name
  'S','h','a','p','e',' ','t','h','e',
  ' ','W','o','r','l','d',
  0x77};            // FCS
```

Figure 9.48. A set device name message from the VerySimpleApplicationProcessor_xxx project.

9.6.4. Read and Write Indications

Figure 9.49 shows the message exchange when the client issues a read request. The

NP sends a **read indication** to the AP, containing the connection and handle of the characteristic. The AP responds with a read confirmation containing status, connection, handle, and the data.

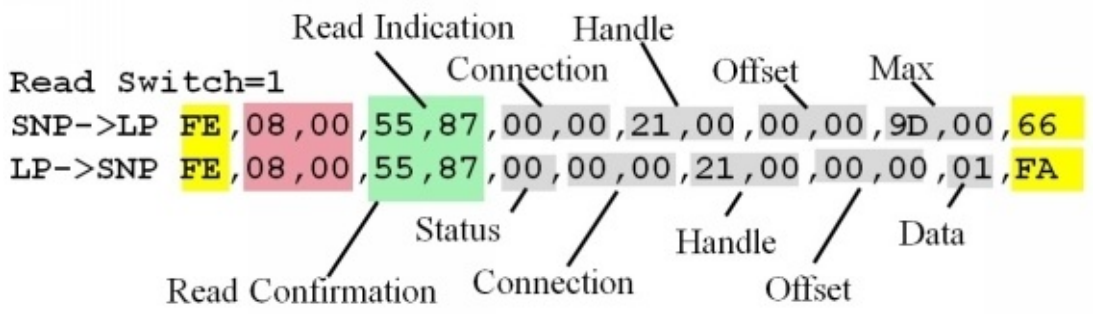


Figure 9.49. TExaSdisplay output occurring when the client issues a read request. These data were collected running the VerySimpleApplicationProcessor_xxx project.

Figure 9.50 shows the message exchange when the client issues a write request. The NP sends a **write indication** to the AP, containing the connection, handle of the characteristic, and the data to be written. The AP responds with a **write confirmation** containing status, connection, and handle.

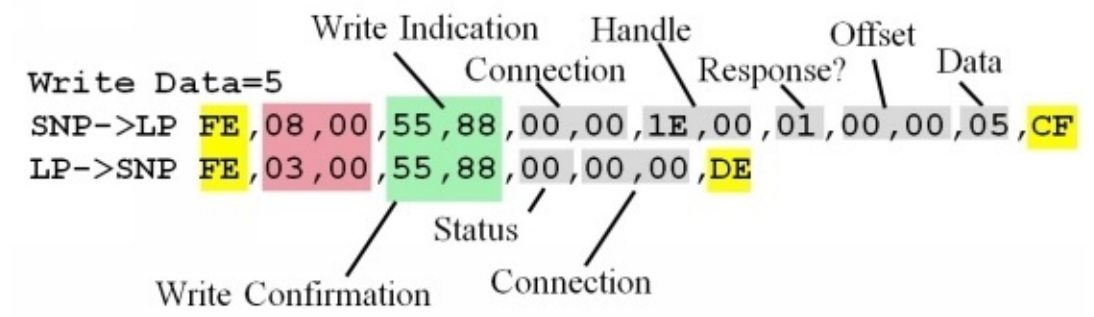


Figure 9.50. TExaSdisplay output occurring when the client issues a write request. These data were collected running the VerySimpleApplicationProcessor_xxx project.

9.7. Application Layer Protocols for Embedded Systems

9.7.1. CoAP

The **Constrained Application Protocol** (CoAP) was specifically developed to allow resource-constrained devices to communicate over the Internet using UDP instead of TCP. In particular, many embedded devices have limited memory, processing power, and energy storage. Developers can interact with any CoAP-enabled device the same way they would with a device using a traditional Representational state transfer (REST) based API like HTTP. CoAP is particularly useful for communicating with low-power sensors and devices that need to be controlled via the Internet.

CoAP is a simple request/response protocol very similar to HTTP, that follows a traditional client/server model. Clients can make GET, PUT, POST, and DELETE requests to resources. CoAP packets use bitfields to maximize memory efficiency, and they make extensive usage of mappings from strings to integers to keep the data packets small enough to transport and interpret on-device. A CoAP message header is only 4-bytes long with most control messages being just that length. Most optional fields in the message format are in binary with the payload restricted in size so all CoAP messages fit inside a UDP datagram.

TCP is a connection oriented protocol, which means the server, or a client, will open a socket and establish a connection with the server. And the communication is done over a connection. For the duration of the communication, the connection is on. Whereas, COAP works on UDP, which means that it's connectionless. And it allows what we call as a disconnected operation, which means that the client and the server are not connected to each other. And therefore, they can act asynchronously.

Aside from the extremely small packet size, another major advantage of CoAP is its usage of UDP; using datagrams allows for CoAP to be run on top of packet-based technologies like SMS. There is a one-to-one mapping between CoAP and HTTP effectively providing a bridge between the all popular HTTP protocol to the emerging CoAP protocol.

All CoAP messages can be marked as either “confirmable” or “nonconfirmable,” serving as an application-level Quality of Service (QoS) to provide reliability. While SSL/TLS encryption isn’t available over UDP, CoAP makes use of Datagram Transport Layer Security (DTLS), which is analogous to the TCP version of TLS. The default level of encryption is equivalent to a 3,072-bit RSA key. Even with all of this, CoAP is designed to work on microcontrollers with as little as 10KB of RAM.

One of the downsides of CoAP: It's a one-to-one protocol. Though extensions that

make group broadcasts possible are available, broadcast capabilities are not inherent to the protocol. Arguably, an even more important disadvantage is the need for both devices to be simultaneously powered, so when one sends a UDP, the other can receive it. In summary, the highlights of CoAP include:

Small 4-byte header
Option fields in binary
Messages fit into one UDP datagram (no fragmentation)
Works with SMS (text messaging)
Connectionless
Needs less than 10 kB of RAM

http://www.infoworld.com/article/2972143/internet-of-things/real-time-protocols-for-iot-apps.html

9.7.2 MQTT

Message Queue Telemetry Transport (MQTT) is a publish-subscribe messaging protocol, abbreviated as **pub-sub**. The MQTT name was inherited from a project at IBM. Similar to CoAP, it was built with resource-constrained devices in mind. MQTT has a lightweight packet structure designed to conserve both memory usage and power. A connected device subscribes to a topic hosted on an MQTT broker. Every time another device or service publishes data to a topic, all of the devices subscribed to it will automatically get the updated information.

Figure 9.51 shows the basic idea of the pub-sub model. MQTT uses an intermediary, which is called a **broker**. There are clients, or publishers, which produce data. The MQTT protocol calls this data a **topic**, and each topic must have a unique identifier. The figure shows a temperature sensor, which is an embedded device with a sensor attached, and it periodically publishes the topic “temperature”. To publish a topic means to send data to the broker. The broker keeps track of all the published information. Subscribers are devices consumers, which are interested in the data. What the subscribers do is they express their interest in a topic by sending a subscription message. In this figure we have two devices that have subscribed to the topic “temperature”. Whenever new data is available, the broker will serve it to both subscribers.

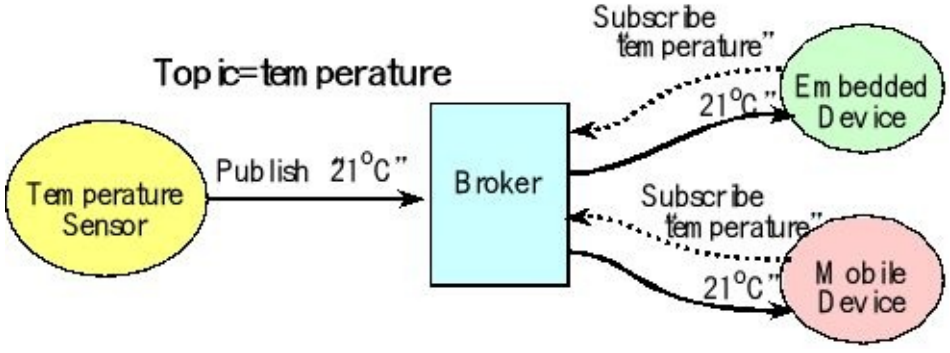


Figure 9.51. With MQTT, the broker acts as an intermediary between producers and consumers.

The fundamental advantage of a pub/sub model for communication in contrast with a client-server model is the decoupling of the communicating entities in space, time and synchronization. That is, the publisher and subscribed do not need to know each other, they do not run at the same time and they can act asynchronously. Other advantages of MQTT are the use of a publish-subscribe message queue and the many-to-many broadcast capabilities. Using a long-lived outgoing TCP connection to the MQTT broker, sending messages of limited bandwidth back and forth is simple and straightforward.

The downside of having an always-on connection is that it limits the amount of time the devices can be put to sleep. If the device mostly sleeps, then another MQTT protocol can be used: MQTT-SN, which is an extension of MQTT for sensor networks, originally designed to support ZigBee. MQTT-S is another extension that allows the use of UDP instead of TCP as the transport protocol, with support for peer-to-peer messaging and multicasting.

Another disadvantage of MQTT is the lack of encryption in the base protocol. MQTT was designed to be a lightweight protocol, and incorporating encryption would add a significant amount of overhead to the connection. One can however, use Transport Layer Security(TLS) extensions to TCP, or add custom security at the application level.

References:

http://www.hivemq.com/blog/mqtt-essentials/
http://www.infoworld.com/article/2972143/internet-of-things/real-time-protocols-for-iot-apps.html

9.8. Exercises

9.1 Consider a wired communication system (like UART or CAN).
a) Assume the signal has a rise time of 25 us. What is the approximate highest frequency component of this signal?
b) Assuming a VF of 0.7, what is the wavelength of this highest frequency?
c) Over what cable length would you have to consider this system as a transmission line?

9.2 Consider a communication with a channel bandwidth of 10 kHz and an SNR of 60 dB. What is the maximum possible data transfer rate in bits/sec?

9.3 What are there so many frequency bands for Bluetooth and WiFi?

9.4 Consider bit-stuffing
a) Define bit-stuffing
b) Why do Ethernet and CAN implement bit-stuffing?
c) UART does not implement bit-stuffing. How does the lack of bit-stuffing limit the UART?
d) SPI does not implement bit-stuffing. Why does the lack of bit-stuffing not limit the SPI transmission in the same way as UART is limited?

9.5 Consider how the ACK bit is used in a CAN network.
a) What do the receivers do during the ACK bit?
b) What does it mean if the ACK bit is dominant?
c) What does it mean if the ACK bit is recessive?

9.6 If the CAN channel is noisy, it is possible that some bits will be transmitted in error. Assume there are four nodes, one is transmitting and three are receiving. What happens if a data bit is flipped in the channel due to noise being added into the channel?

9.7 Consider a situation where two microcontrollers are connected with a CAN network. Computer 1 generates 8-bit data packets that must be sent to computer 2, and computer 2 generates 8-bit data packets that must be sent to computer 1. The packets are generated at random times, and the goal is to minimize the latency between when a data packet is generated on one computer to when it is received on the other. Describe the CAN protocol you would use: 11-bit versus 29-bit ID, number of bytes of data, and bandwidth. Clearly describe what is in the ID and how the data is formatted.

9.8 A CAN system has a baud rate of 100,000 bits/sec, 29-bit ID, and three bytes of

data per frame. Assuming there is no bit-stuffing, what is the maximum bandwidth of this network, in bytes/s.

9.9 A CAN system has a baud rate of 200,000 bits/sec, 11-bit ID, and five bytes of data per frame. Assuming there is no bit-stuffing, what is the maximum bandwidth of this network, in bytes/s.

9.10 Consider a situation where 4 microcontrollers are connected together using a CAN network. Assume for this question that each frame contains 100 bits. Also assume the baud rate is 100,000 bits/sec, therefore it takes 1ms to send a frame. Initially, the CAN controllers are initialized (i.e., all computers have previously executed **CAN_Open**).

At time = 0 computer A calls **CAN_Send** with ID=1000
At time = 300 µs computer B calls **CAN_Send** with ID=800
At time = 500 µs computer C calls **CAN_Send** with ID=900
At time = 700 µs computer D calls **CAN_Send** with ID=600

Specify the time sequence in which the four frames occur on the CAN network. Clearly define the begin and end times when each message is visible on the CAN network.

9.11 In a CAN network, what is the purpose of the **CRC** field? I.e., what is CRC used for?

9.12 Why is BLE considered a personal area network, and WiFi is not?

9.13 How does BLE achieve low energy?

9.14 Define the following terms in 16 words or less as they apply to BLE.

a) Service **b)** Characteristic **c)** Advertising
d) Client **e)** Server **f)** Profile
g) Stack **g)** UUID **h)** Handle
i) Read indication **j)** Write indication **k)** Notify indication

10. Robotic Systems

Chapter 10 objectives are to:

- Introduce the general approach to digital control systems
- Design and implement some simple closed-loop control systems
- Develop a methodology for designing PID control systems
- Present the terminology and give examples of fuzzy logic control system

Throughout all three volumes of this series of books, we developed systems that collected information concerning the external environment. A typical application of embedded systems is to use this information in order to control the external environment. To build this microcontroller-based control system we will need an output device that the computer can use to manipulate the external environment. Control systems originally involved just analog electronic circuits and mechanical devices. With the advent of inexpensive yet powerful microcontrollers, implementing the control algorithm in software provided a lower cost and more powerful product. The goal of this chapter is to provide a brief introduction to this important application area. Control theory is a richly developed discipline, and most of this theory is beyond the scope of this book. Consequently, this chapter focuses more on implementing the control system with an embedded computer and less on the design of the control equations.

10.1. Introduction to Digital Control Systems

A **control system** is a collection of mechanical and electrical devices connected for the purpose of commanding, directing, or regulating a **physical plant** (see Figure 10.1). The **real state variables** are the properties of the physical plant that are to be controlled. The **sensor** and **state estimator** comprise a data acquisition system. The goal of this data acquisition system is to estimate the state variables. A **closed-loop** control system uses the output of the state estimator in a feedback loop to drive the errors to zero. The control system compares these **estimated state variables**, *X'(t)*, to the **desired state variables**, *X*(t)*, in order to decide appropriate action, *U(t)*. The **actuator** is a transducer that converts the control system commands, *U(t)*, into driving forces, *V(t)*, that are applied to the physical plant. In general, the goal of the control system is to drive the real state variables to equal the desired state variables. In actuality though, the controller attempts to drive the estimated state variables to equal the desired state variables. It is important to have an accurate state estimator, because any differences between the estimated state variables and the real state variables will translate directly into controller errors. If we define the error as the difference between the desired and estimated state variables:

$$e(t) = X^*(t) - X'(t)$$

then the control system will attempt to drive *e(t)* to zero. In general control theory, *X(t)*, *X'(t)*, *X*(t)*, *U(t)*, *V(t)* and *e(t)* refer to vectors, but the examples in this chapter control only a single parameter. Even though this chapter shows one-dimensional systems, and it should be straight-forward to apply standard multivariate control theory to more complex problems. We usually evaluate the effectiveness of a control system by determining three properties: steady state controller error, transient response, and stability. The **steady state controller error** is the average value of *e(t)*. The **transient response** is how long does the system take to reach 99% of the final output after *X** is changed. A system is **stable** if steady state (smooth constant output) is achieved. The error is small and bounded on a stable system. An **unstable** system oscillates, or it may saturate.

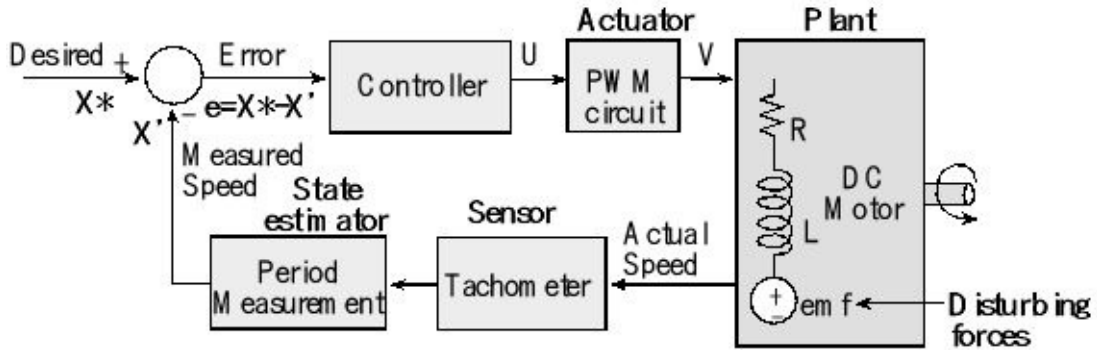


Figure 10.1. Block diagram of a microcomputer-based closed-loop control system.

An **open-loop** control system does not include a state estimator. It is called open loop because there is no feedback path providing information about the state variable to the controller. It will be difficult to use open-loop with the plant that is complex because the disturbing forces will have a significant effect on controller error. On the other hand, if the plant is well-defined and the disturbing forces have little effect, then an open-loop approach may be feasible. Because an open-loop control system does not know the current values of the state variables, large errors can occur. Stepper motors are often used in open loop fashion.

10.2. Binary Actuators

10.2.1. Electrical Interface

Relays, solenoids, and DC motors are grouped together because their electrical interfaces are similar. We can add speakers to this group if the sound is generated with a square wave. In each case, there is a coil, and the computer must drive (or not drive) current through the coil. To interface a coil, we consider **voltage**, **current** and **inductance**. We need a power supply at the desired voltage requirement of the coil. If the only available power supply is larger than the desired coil voltage, we use a voltage regulator (rather than a resistor divider to create the desired voltage.) We connect the power supply to the positive terminal of the coil, shown as **+V** in Figure 10.2. We will use a transistor device to drive the negative side of the coil to ground. The computer can turn the current on and off using this transistor. The second consideration is current. In particular, we must however select the power supply and an interface device that can support the coil current. The 7406 is an open collector driver capable of sinking up to 40 mA. The 2N2222 is a **bipolar junction transistor** (BJT), NPN type, with moderate current gain. The TIP120 is a **Darlington transistor**, also NPN type, which can handle larger currents. The IRF540 is a **MOSFET** transistor that can handle even more current. BJT and Darlington transistors are current-controlled (meaning the output is a function of the input current), while the MOSFET is voltage-controlled (output is a function of input voltage). When interfacing a coil to the microcontroller, we use information like Table 10.1 to select an interface device capable the current necessary to activate the coil. It is a good design practice to select a driver with a maximum current at least twice the required coil current. When the digital **Port** output is high, the interface transistor is active and current flows through the coil. When the digital **Port** output is low, the transistor is not active and no current flows through the coil.

Device	Type	Maximum current
TM4C	CMOS	8 mA (set bits in DR8R)
MSP432	CMOS	20 mA (DS=1, P2.0 – P2.3)
7406	TTL logic	40 mA
PN2222	BJT NPN	150 mA
2N2222	BJT NPN	500 mA
TIP120	Darlington NPN	5 A
IRF540	power MOSFET	28 A

Table 10.1. Four possible devices that can be used to interface a coil to the microcontroller.

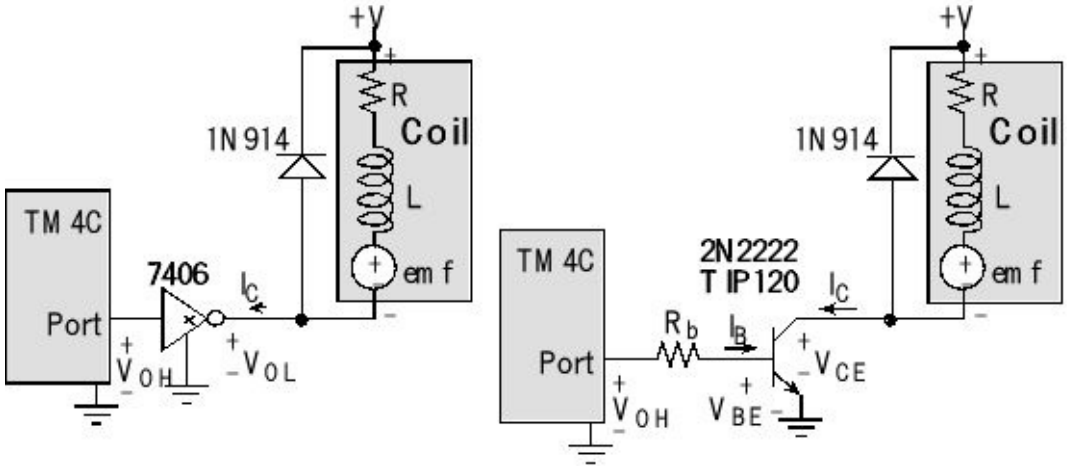


Figure 10.2. Binary interface to EM relay, solenoid, DC motor or speaker.

The third consideration is inductance in the coil. The 1N914 diode in Figure 10.1 provides protection from the **back emf** generated when the switch is turned off, and the large *dI/dt* across the inductor induces a large voltage (on the negative terminal of the coil), according to $V=L \cdot dI/dt$. For example, if you are driving 0.1A through a 0.1 mH coil (**Port** output = 1) using a 2N2222, then disable the driver (**Port** output = 0), the 2N2222 will turn off in about 20ns. This creates a *dI/dt* of at least $5 \cdot 10^6$ A/s, producing a back emf of 500 V! The 1N914 diode shorts out this voltage, protecting the electronics from potential damage. The 1N914 is called a **snubber diode**.

> **Observation:** It is important to realize that many devices cannot be connected directly up to the microcontroller. In the specific case of motors, we need an interface that can handle the voltage and current required by the motor.

If you are sinking 16 mA (I_{OL}) with the 7406, the output voltage (V_{OL}) will be 0.4V. However, when the I_{OL} of the 7406 equals 40 mA, its V_{OL} will be 0.7V. 40 mA is not a lot of current when it comes to typical coils. However, the 7406 interface is appropriate to control small relays.

> **Checkpoint 10.1**: A relay is interfaced with the 7406 circuit in Figure 10.2. The positive terminal of the coil is connected to +5V and the coil requires 40 mA. What will be the voltage across the coil when active?

When designing an interface, we need to know the desired coil voltage (V_{coil}) and coil current (I_{coil}). Let V_{be} be the base-emitter voltage that activates the NPN transistor and let h_{fe} be the current gain. There are three steps when interfacing an N-channel (right side of Figure 10.2.)

1) Choose the interface voltage V equal to Vcoil (since V_{CE} is close to zero)
2) Calculate the desired base current $I_b = I_{coil} / h_{fe}$ (since I_C equals I_{coil})
3) Calculate the interface resistor $R_b \leq (V_{OH} - V_{be}) / I_b$ (choose a resistor 2 to 5 times smaller)

With an N-channel switch, like Figure 10.2, current is turned on and off by connecting/disconnecting one side of the coil to ground, while the other side is fixed

at the voltage supply. A second type of binary interface uses P-channel switches to connect/disconnect one side of the coil to the voltage supply, while the other side fixed at ground, as shown in Figure 10.3. In other to activate a PNP transistor (e.g., PN2907 or TIP125), there must be a V_{EB} greater than 0.7 V. In order to deactivate a PNP transitory, the V_{EB} voltage must be 0. Because the transistor is a current amplifier, there must be a resistor into the base in order to limit the base current.

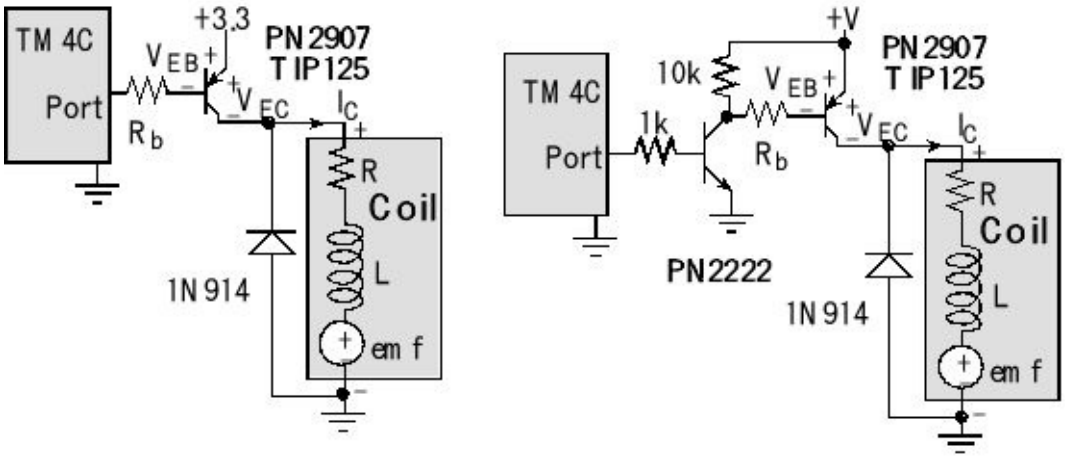


Figure 10.3. PNP interface to EM relay, solenoid, DC motor or speaker.

To understand how the PNP interface on the right of Figure 10.3 operates, consider the behavior for the two cases: the Port output is high and the Port output is low. If the Port output is high, its output voltage will be between 2.4 and 3.3 V. This will cause current to flow into the base of the PN2222, and its V_{be} will saturate to 0.7 V. The base current into the PN2222 could be from (2.4-0.7)/1000 to (3.3-0.7)/1000, or 1.7 to 2.6 mA. The microcontroller will be able to source this current. This will saturate the PN2222 and its V_{CE} will be 0.3 V. This will cause current to flow out of the base of the PN2907, and its V_{EB} will saturate to 0.7 V. If the supply voltage is V, then the PN2907 base current is (V-0.7-0.3)/R_b. Since the PNP transistor is on, V_{EC} will be small and current will flow from the supply to the coil. If the port output is low, the voltage output will be between 0 and 0.4V. This not high enough to activate the PN2222, so the NPN transistor will be off. Since there is no I_C current in the PN2222, the 10k and R_b resistors will place $+V$ at the base of the PN2907. Since the V_{EB} of the PN2907 is 0, this transistor will be off, and no current will flow into the coil.

MOSFETs can handle significantly more current than BJT or Darlington transistors. MOSFETs are voltage controlled switches. The difficulty with interfacing MOSFETs to a microcontroller is the large gate voltage needed to activate it. The left side of Figure 10.4 is an N-channel interface. The IRF540 N-channel MOSFET can sink up to 28A when the gate-source voltage is above 7V. This circuit is negative logic. When the port pin is high, the 2N2222 is active making the MOSFET gate voltage 0.3V (V_{CE} of the PN2222). A V_{GS} of 0.3V turns off the MOSFET. When the port pin is low, the 2N2222 is off making the MOSFET gate voltage $+V$ (pulled up through the

10kΩ resistor). The V_{GS} is +V, which turns the MOSFET on.

The right side of Figure 10.4 shows a P-channel MOSFET interface. The IRF9540 P-channel MOSFET can source up to 20A when the source-gate voltage is above 7V. The FQP27P06 P-channel MOSFET can source up to 27A when the source-gate voltage is above 6V. This circuit is positive logic. When the port pin is high, the 2N2222 is active making the MOSFET gate voltage 0.3V. This makes V_{SG} equal to $+V$-0.3, which turns on the MOSFET. When the port pin is low, the 2N2222 is off. Since the 2N2222 is off, the 10kΩ pull-up resistor makes the MOSFET gate voltage +V. In this case V_{SG} equals 0, which turns off the MOSFET.

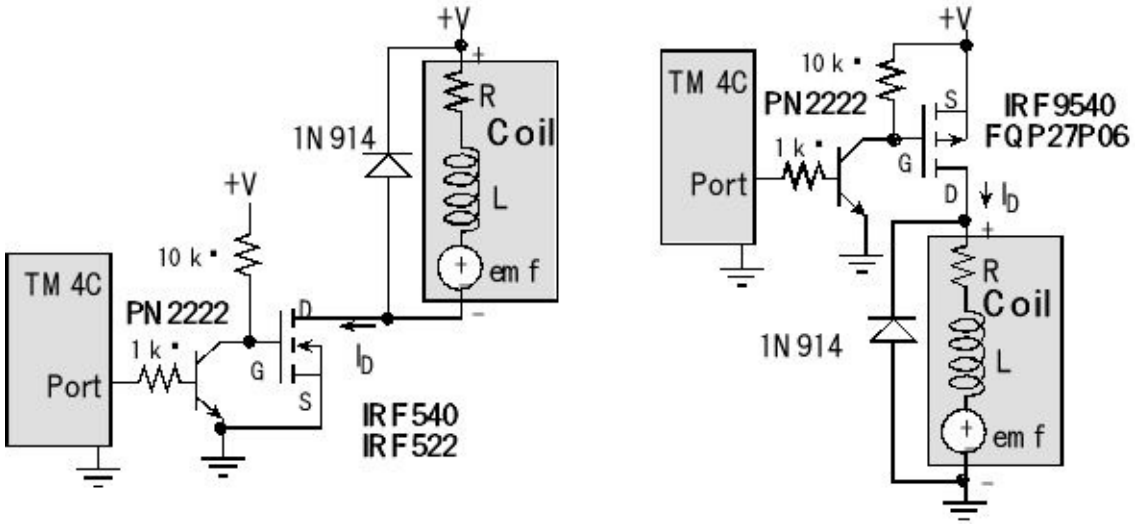


Figure 10.4. MOSFET interfaces to EM relay, solenoid, DC motor or speaker.

An H-bridge combines P-channel and N-channel devices allowing current to flow in either direction. Figures 4.26 and 4.27 in Volume 2 show applications of the L293 H-bridge, while Figure 10.5 shows one of the H-bridge circuits internal to the L293. If 1A is high, Q_1 is on and Q_2 is off. If 1A is low, Q_1 is off and Q_2 is on. 2A controls Q_3 and Q_4 in a similar fashion. If 1A is high and 2A is low, then Q_1 Q_4 are on and current flows left to right across coil A. If 1A is low and 2A is high, then Q_2 Q_3 are on and current flows right to left across coil A.

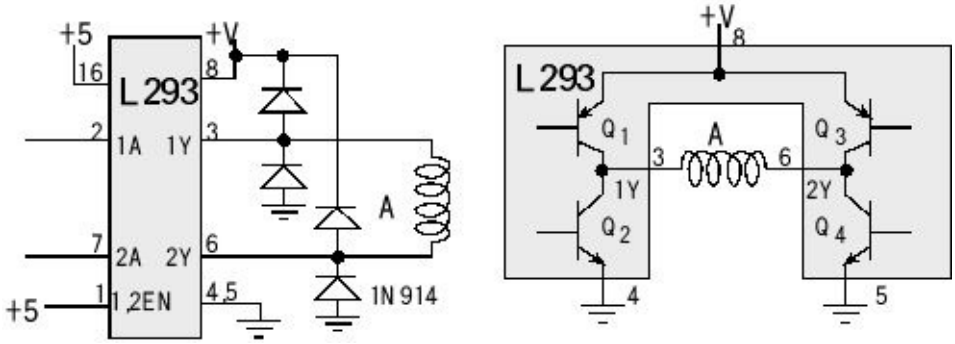


Figure 10.5. An H-bridge can drive current in either direction (the actual L293 uses all N-channel devices).

10.2.2. DC Motor Interface with PWM

Similar to the solenoid and EM relay, the DC motor has a frame that remains motionless, and an armature that moves. In this case, the armature moves in a circular manner (shaft rotation).

In the previous interfaces the microcontroller was able to control electrical power to a device in a binary fashion: either all on or all off. Sometimes it is desirable for the microcontroller to be able to vary the delivered power in a variable manner. One effective way to do this is to use pulse width modulation (PWM). The basic idea of PWM is to create a digital output wave of fixed frequency, but allow the microcontroller to vary its duty cycle. The system is designed in such a way that **High+Low** is constant (meaning the frequency is fixed). The **duty cycle** is defined as the fraction of time the signal is high:

$$duty\ cycle = \frac{High}{High + Low}$$

Hence, duty cycle varies from 0 to 1. We interface this digital output wave to an external actuator (like a DC motor), such that power is applied to the motor when the signal is high, and no power is applied when the signal is low. We purposely select a frequency high enough so the DC motor does not start/stop with each individual pulse, but rather responds to the overall average value of the wave. The average value of a PWM signal is linearly related to its duty cycle and is independent of its frequency. Let P ($P=V*I$) be the power to the DC motor, Figures 10.2 - 10.5, when the PWM signal is high. Under conditions of constant speed and constant load, the delivered power to the motor is linearly related to duty cycle.

Delivered Power =

$$duty\ cycle * P = \frac{High}{High + Low} * P$$

Unfortunately, as speed and torque vary, the developed emf will affect delivered power. Nevertheless, PWM is a very effective mechanism, allowing the microcontroller to adjust delivered power.

A DC motor has an electro-magnet as well. When current flows through the coil, a magnetic force is created causing a rotation of the shaft. Brushes positioned between the frame and armature are used to alternate the current direction through the coil, so that a DC current generates a continuous rotation of the shaft. When the current is removed, the magnetic force stops, and the shaft is free to rotate. The resistance in the coil (R) comes from the long wire that goes from the + terminal to the – terminal of the motor. The inductance in the coil (L) arises from the fact that the wire is wound into coils to create the electromagnetics. The coil itself can generate its own voltage (emf) because of the interaction between the electric and magnetic fields. If the coil is a DC motor, then the emf is a function of both the speed of the motor and the developed torque (which in turn is a function of the applied load on the motor.)

Because of the internal emf of the coil, the current will depend on the mechanical load. For example, a DC motor running with no load might draw 50 mA, but under load (friction) the current may jump to 500 mA.

There are lots of motor driver chips, but they are fundamentally similar to the circuits shown in Figure 10.2. For the 2N2222 and TIP120 NPN transistors, if the port output is low, no current can flow into the base, so the transistor is off, and the collector current, I_C, will be zero. If the port output is high, current does flow into the base and V_{BE} goes above V_{BEsat} turning on the transistor. The transistor is in the linear range if $V_{BE} \le V_{BEsat}$ and $I_c = h_{fe} \cdot I_b$. The transistor is in the saturated mode if $V_{BE} \ge V_{BEsat}$, V_{CE} = 0.3V and $I_c < h_{fe} \cdot I_b$. We select the resistor for the NPN transistor interfaces to operate right at the transition between linear and saturated mode. We start with the desired coil current, I_{coil} (the voltage across the coil will be +V-V_{CE} which will be about +V-0.3V). Next, we calculate the needed base current (I_b) given the current gain of the NPN

$$I_b = I_{coil} / h_{fe}$$

knowing the current gain of the NPN (h_{fe}), see Table 10.2. Finally, given the output high voltage of the microcontroller (V_{OH} is about 3.3 V) and base-emitter voltage of the NPN (V_{BEsat}) needed to activate the transistor, we can calculate the desired interface resistor.

$$R_b \le (V_{OH} - V_{BEsat}) / I_b = h_{fe} * (V_{OH} - V_{BEsat}) / I_{coil}$$

The inequality means we can choose a smaller resistor, creating a larger I_b. Because the of the transistors can vary a lot, it is a good design practice to make the R_b resistor about ½ the value shown in the above equation. Since the transistor is saturated, the increased base current produces the same V_{CE} and thus the same coil current.

Parameter	PN2222 (I_C=150mA)	2N2222 (I_C=500mA)	TIP120 (I_C=3A)
h_{fe}	100	40	1000
h_{ie}	60 Ω	250 to 8000 Ω	70 to 7000 Ω
V_{BEsat}	0.6	2	2.5 V
V_{CE} at saturation	0.3	1	2 V

Table 10.2. Design parameters for the 2N2222 and TIP120.

The IRF540 MOSFET is a voltage-controlled device, if the **Port** output is high, the 2N2222 is on, the MOSFET is off, and the coil current will be zero. If the **Port** output is low, the 2N2222 is off, the gate voltage of the MOSFET will be +V, the MOSFET is on, and the V_{DS} will be very close to 0. The IRF540 needs a large gate voltage (> 10V) to fully turn so the drain will be able to sink up to 28 A.

Because of the resistance of the coil, there will not be significant *dI/dt* when the device is turned on. Consider a DC motor as shown in Figure 10.2 with V= 12V, R = 50 Ω and L = 100 µH. Assume we are using a 2N2222 with a V_{CE} of 1 V at saturation. Initially the motor is off (no current to the motor). At time t=0, the digital port goes from 0 to +3.3 V, and transistor turns on. Assume for this section, the emf is zero (motor has no external torque applied to the shaft) and the transistor turns on instantaneously, we can derive an equation for the motor (I_c) current as a function of time. The voltage across both *LC* together is 12-V_{CE} = 11 V at time = 0^+. At time = 0^+, the inductor is an open circuit. Conversely, at time = ∞, the inductor is a short circuit. The I_c at time 0^- is 0, and the current will not change instantaneously because of the inductor. Thus, the I_c is 0 at time = 0^+. The I_c is 11V/50Ω = 220mA at time = ∞.

$$11\ \text{V} = I_c * R + L * d\,I_c/dt$$

General solution to this differential equation is

$$I_c = I_0 + I_1 e^{-t/T} \qquad d\,I_c/dt = -(I_1/T)e^{-t/T}$$

We plug the general solution into the differential equation and boundary conditions.

$$11\ \text{V} = (I_0 + I_1 e^{-t/T}) * R - L * (I_1/T) e^{-t/T}$$

To solve the differential equation, the time constant will be $T = L/R$ = 2 µsec. Using initial conditions, we get

$$I_c = 220\text{mA} * (1 - e^{-t/2\mu s})$$

Example 10.4. Design an interface for two +12V 1A geared DC motors. These two motors will be used to propel a robot with two independent drive wheels.

Solution: We will use two copies of the TIP120 circuit in Figure 10.6 because the TIP120 can sink at least three times the current needed for this motor. We select a +12V supply and connect it to the +V in the circuit. The needed base current is

$$I_b = I_{coil}/h_{fe} = 1\text{A}/1000 = 1\text{mA}$$

The desired interface resistor.

$$R_b \le (V_{OH} - V_{be})/I_b = (3.3\text{-}2.5)/1\text{mA} = 800\ \Omega$$

To cover the variability in h_{fe}, we will use a 330 Ω resistor instead of the 800 Ω. The actual voltage on the motor when active will be +12-2 = 10V. The coils and transistors can vary a lot, so it is appropriate to experimentally verify the design by measuring the voltages and currents. Two PWM outputs are used to control the robot. The period of the PWM output is chosen to be about 10 times shorter than the time constant of the motor. The electronic driver will turn on and off at this rate, but the motor only responds to the average level. The software sets the duty cycle of the PWM to adjust the delivered power. When active, the interface will drive +10 V across the motor. The current will be a function of the friction applied to the shaft.

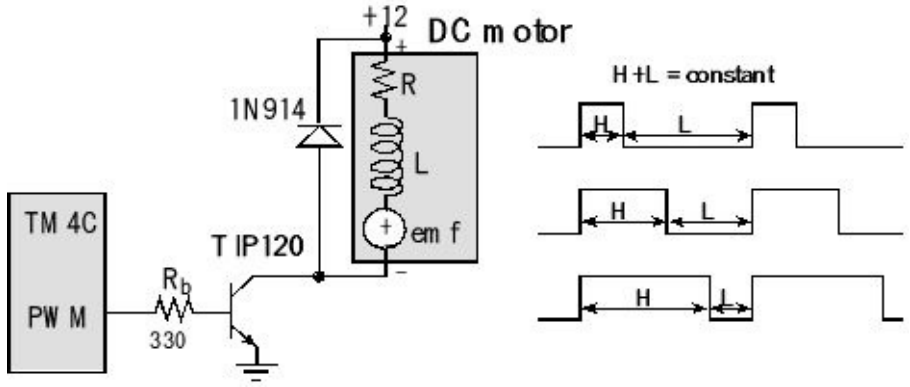


Figure 10.6. DC motor interface.

Similar to the solenoid and EM relay, the DC motor has a frame that remains motionless (called the **stator**), and an armature that moves (called the **rotor**), see Figure 10.7.

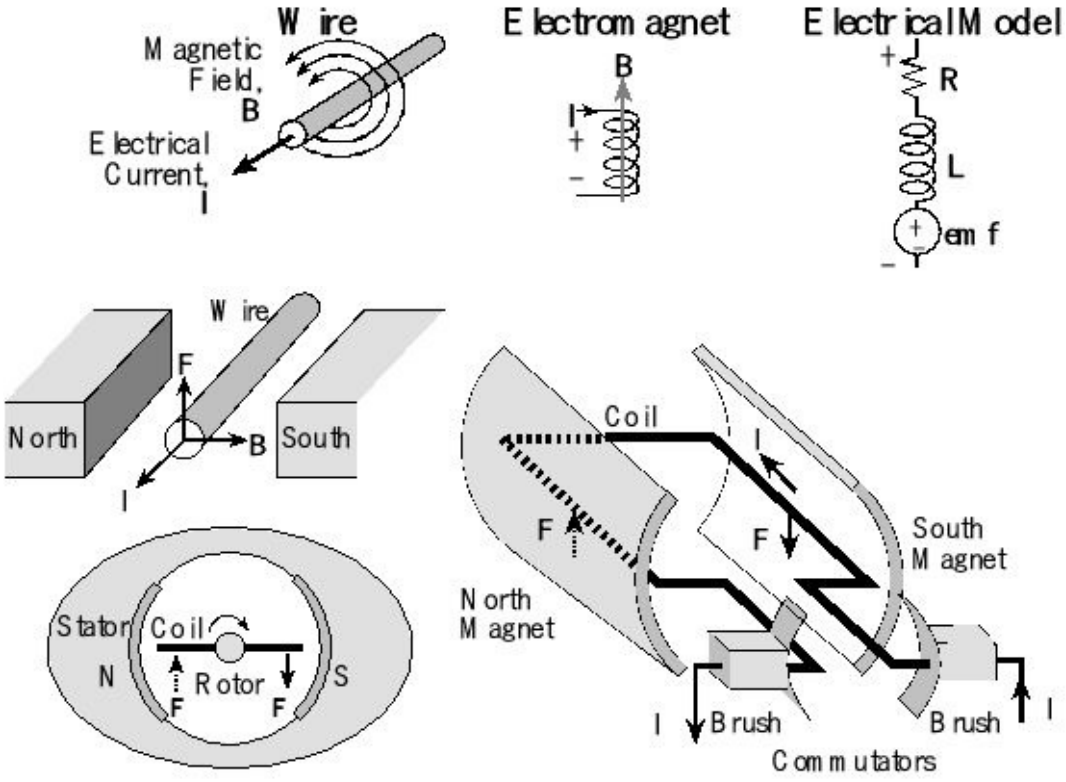


Figure 10.7. A brushed DC motor uses a commutator to flip the coil current.

A **brushed DC motor** has an electromagnetic coil as well, located on the rotor, and the rotor is positioned inside the stator. In Figure 10.7, North and South refer to a permanent magnet, generating a constant *B* field from left to right. In this case, the rotor moves in a circular manner. When current flows through the coil, a magnetic force is created causing a rotation of the shaft. A brushed DC motor uses commutators to flip the direction of the current in the coil. In this way, the coil on the right always has an up force, and the one on the left always has a down force. Hence, a constant current generates a continuous rotation of the shaft. When the current is

removed, the magnetic force stops, and the shaft is free to rotate. In a pulse-width modulated DC motor, the computer activates the coil with a current of fixed magnitude but varies the duty cycle in order to adjust the power delivered to the motor.

10.3. Sensors

Tachometers can be used to measure rotational speed of a motor. Some tachometers produce a sine wave with a frequency and amplitude proportional to motor speed. To use input capture, we need to convert the sine wave into a corresponding square wave of the same period. We can use a voltage comparator to detect events in an analog waveform. The input voltage range is determined by the analog supply voltages of the comparator. The output is takes on two values, shown an V_h and V_l in Figure 10.8. To reduce noise, a comparator with hysteresis has two thresholds, V_{t+} and V_{t-}. In both the positive and negative logic cases the threshold (V_{t+} or V_{t-}) depends on the present value of the output.

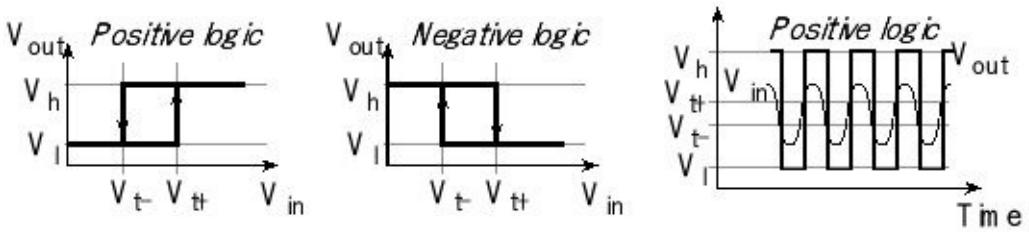


Figure 10.8. Input/output response of voltage converters with hysteresis.

Hysteresis prevents small noise spikes from creating a false trigger.

> **Performance Tip:** In order to eliminate false triggering, we select a hysteresis level (V_{t+}-V_{t-}) greater than the noise level in the signal.

In Figure 10.9, a rail-to-rail op amp is used to design a voltage comparator. Since the output swings from 0 to 3.3 V, it can be connected directly to an input pin of the microcontroller. On the other hand, since +3.3 and 0 are used to power the op amp, the analog input must remain in the 0 to +3.3 V range. The hysteresis level is determined by the amplitude of the output and the $R_1/(R_1+R_2)$ ratio. If the output is at 0V, the voltage at the +terminal is $V_{in}*R_2/(R_1+R_2)$. The output switches when the voltage at the +terminal goes above 1.65. By solving for V_{in}*200k/(10k+200k)=1.65, we see V_{in} must go above +1.73 for the output to switch. Similarly, if the output is at +3.3 V, the voltage at the +terminal can be calculated as $V_{in}+(3.3-V_{in})*R_1/(R_1+R_2)$. The output switches back when the voltage at the +terminal goes below 1.65. By solving for $V_{in}+(3.3-V_{in})*R_1/(R_1+R_2)=1.65$, we see V_{in} go below +1.57 before the +terminal of the op amp falls below 1.65 V. In linear mode circuits we should not use the supply voltage to create voltage references, but in a saturated mode circuit, power supply ripple will have little effect on the response.

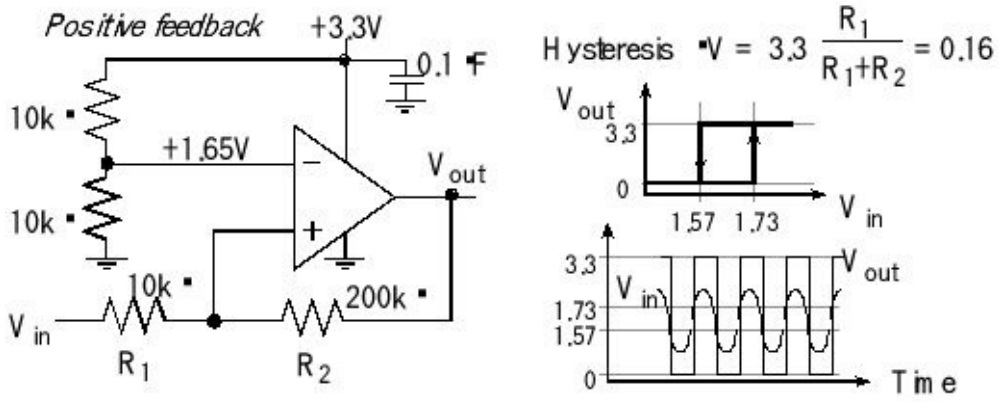


Figure 10.9. A voltage comparator with hysteresis using a rail to rail op amp.

10.4. Odometry

Odometry is a method to predict position from wheel rotations. We assume the wheels do not slip along the ground. If one wheel moves but the other does not, it will rotate about a single contact point of the wheel to the ground. If one wheel moves more than the other, then there will be both a motion and a rotation about a point somewhere along line defined by the axle connecting the two wheels. We define the robot center of gravity (cog) as a point equidistant from the pivot points. The robot position is defined as the (x,y) location and the compass direction, or yaw angle θ, of the cog. See Figure 10.10.

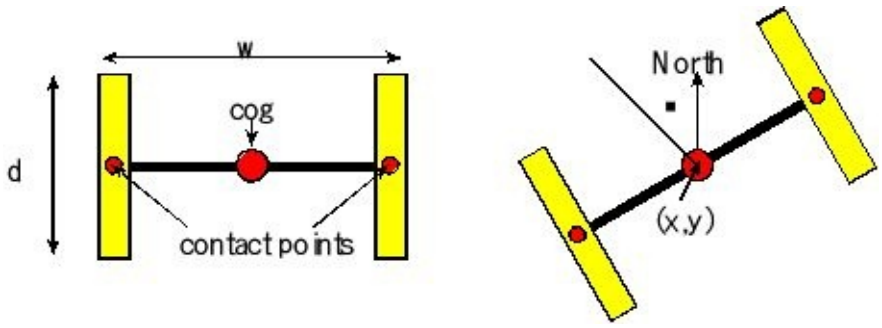


Figure 10.10. A robot with two drive wheels is defined by the wheel base and wheel diameter.

Constants

Number of slots/rotation, n=32
Wheel diameter, d = 886 (0.01cm)

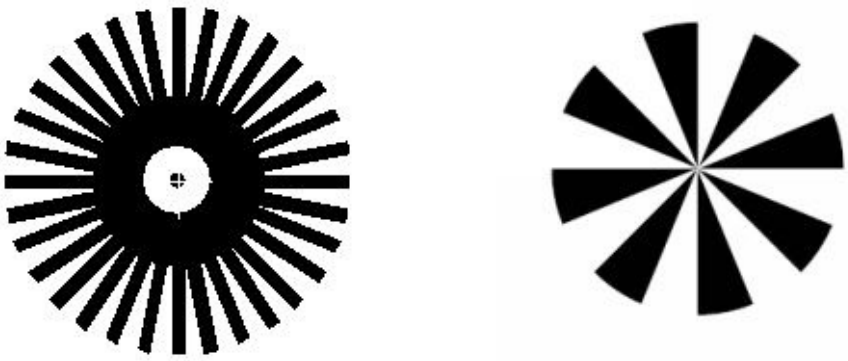


Figure 10.11. To measure wheel motion we used an encoder on each wheel.

Wheelbase (distance between wheels), w = 1651 (0.01cm)
Wheel circumference, $c = \pi d$ = 2783 (0.01cm)

Measurements

L_{Count} the number of slots of left wheel in 349.5ms. R_{Count} the number of slots of right

wheel in 349.5ms. At 150 RPM, there will be 28 counts in 349.5 ms. Some simple cases are found in Table 10.3, where *m* is any number from -28 to +28.

LCount	***RCount***	*Motion*
m	*m*	straight line motion in the current direction
0	*m*	pivot about stopped left motor
m	0	pivot about stopped right motor
m	*-m*	pure rotation about cog

Table 10.3. Example measurements, relationship between counts and motion.

Derivations

$Lr = L_{Count} * c/n$ the arc distance traveled by the left wheel (0.01cm)

$Rr = R_{Count} * c/n$ the arc distance traveled by the right wheel (0.01cm)

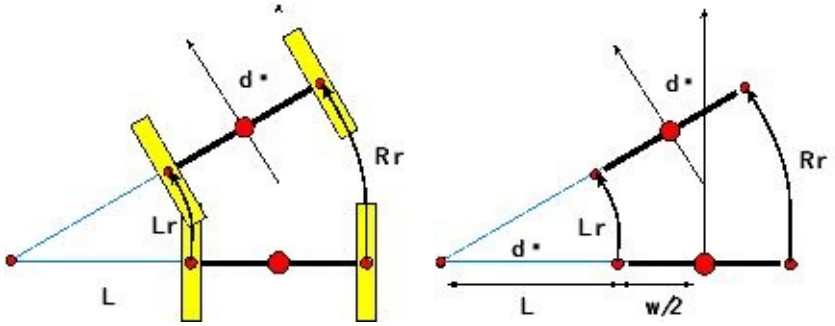


Figure 10.12. Motions occurring during a left turn.

Using similar triangles, we can find the new pivot point. Assuming *Rr* and *Lr* are both positive and *Rr>Lr*, we get

$L/Lr = (L+w)/Rr$

$L/Lr - L/Rr = w/Rr$

$L\ Rr - L\ Lr = w\ Lr$

$L = w\ Lr/(Rr - Lr)$

Notice also the change in yaw, dθ, is the same angle as the sector created by the change in axle. The change in angle is

$\mathbf{d}\theta = \mathbf{Lr}/\mathbf{L} = Rr/(L+w)$

We can divide the change in position into two components

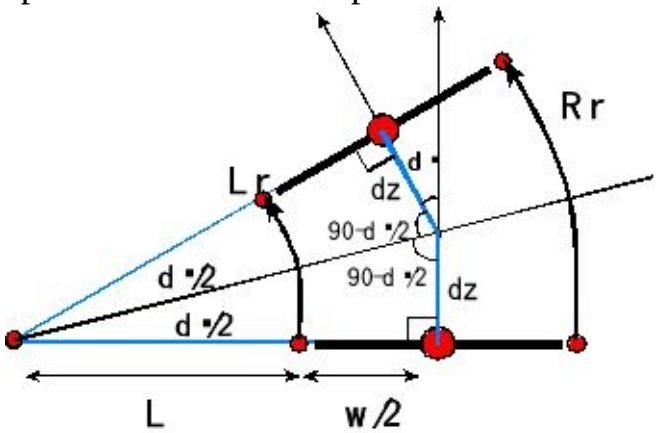


Figure 10.13. Geometry of a left turn.

The exact calculation for position change is
$dz = (L+w/2)*\tan(d\theta/2)$
but if dθ is small, we can approximate *dz* by the arc length.
$dz = d\theta/2*(L+w/2)$

Initialize
We initialize the system by specifying the initial position and yaw.
(x, y, θ) (0.01cm, 0.01cm, 0.01 radian)
Calculations (run this periodically, measuring L_{Count} R_{Count})

$Lr = L_{Count} * c/n$ (0.01cm)
$Rr = R_{Count} * c/n$ (0.01cm)
$L = (w*Lr)/(Rr - Lr)$ (0.01cm)

$d\theta = (100*Lr)/L$ (0.01 radian)
$dz = ((d\theta/2)*(L+w/2))/100$ (0.01cm) *approximation*
or $dz = (\tan(d\theta/2)*(L+w/2))/100$ (0.01cm) *more accurate*
$x = x + dz*\cos(\theta)$ (0.01cm)
$y = y + dz*\sin(\theta)$ (0.01cm) *first part of move*
$\theta = \theta + d\theta$ (0.01 radian)
$x = x + dz*\cos(\theta)$ (0.01cm)
$y = y + dz*\sin(\theta)$ (0.01cm) *second part of move*

10.5. Simple Closed-Loop Control Systems.

A **bang-bang controller** uses a binary actuator, meaning the microcontroller output can be on or off. Other names for this controller are **binary controller**, **two-position controller**, and **on/off controller**. It is a closed-loop control system, because there is a sensor that measures the status of the system. This signal is called the measurand or state variable. Assume when the actuator is on the measurand increases, and when the actuator is off, the measurand decreases. There is a desired point for the measurand. The bang-bang controller is simple. If the measurand is too small, the actuator is turned on, and if the measurand is too large the actuator is turned off.

This digital control system applies heat to the room in order to maintain the temperature as close to the desired temperature as possible (Figure 10.14). This is a closed-loop control system because the control signals (heat) depend on the state variables (temperature). In this application, the actuator has only two states: *on* that warms up the room and *off* that does not apply heat. For this application to function properly, there must be a passive heat loss that lowers the room temperature when the heater is turned off. On a hot summer day, this heater system will not be able to keep the house cool. A bang-bang controller turns on the power if the measured temperature is too low and turns off the power if the temperature is too high. To implement **hysteresis**, we need two set-point temperatures, T_{high} and T_{low}. The controller turns on the power (activate relay) if the temperature goes below T_{low} and turns off the power (deactivate relay) if the temperature goes above T_{high}. The difference T_{high} - T_{low} is called hysteresis. The hysteresis extends the life of the relay by reducing the number of times the relay opens and closes.

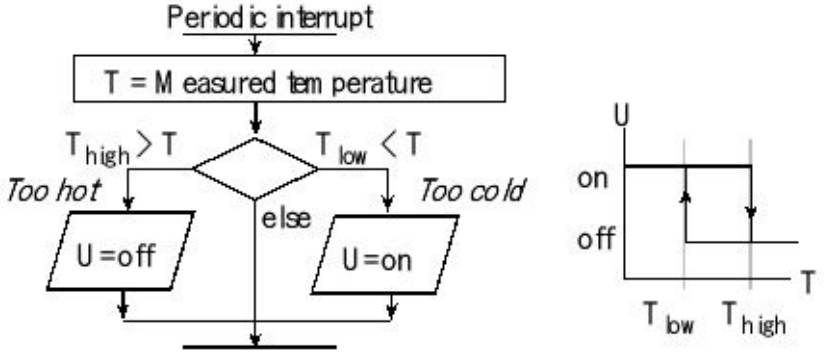


Figure 10.14. Flowchart of a Bang-Bang Temperature Controller

Assume the function **SE** () returns the estimated temperature as a binary fixed-point number with a resolution of 0.5 °C. Program 10.1 uses a periodic interrupt so that the bang-bang controller runs in the background. The interrupt period is selected to be about the same asthe time constant of the physical plant. The temperature variables **Tlow** , **Thigh** and **T** could be in any format, as long as the three formats are the same.

Checkpoint 10.2: What happens if **Tlow** and **Thigh** are too close together? What

happens if **Tlow** and **Thigh** are too far apart?

Observation: Bang-bang control works well with a physical plant with a very slow response.

```
int32_t Tlow,Thigh;     // controller set points, 0.5 C
void Timer0A_Handler(void){
int32_t T=SE();         // estimated temperature, 0.5 C
 if(T < Tlow){
   TurnOn();}       // too cold so turn on heat
 else if (T > Thigh){
   TurnOff();       // too hot so turn off heat
 }                  // leave as is if Tlow<T<Thigh
 TIMER0_ICR_R = 0x01;// acknowledge timer0A periodic timer
}
```

Program 10.1. Bang-bang temperature control software.

An **incremental control system** uses an actuator with a finite number of discrete output states. For example, the actuator might be a PWM output with 249 possibilities from 2, 3, 4, … 249 (0 to 100%). It is a closed-loop control system, because there is a sensor that measures the state variable. Assume when the actuator increases the measurand increases, and when the actuator decreases, the measurand decreases. There is a desired point for the measurand. The incremental controller is simple. If the measurand is too small, the actuator is increased, and if the measurand is too large, the actuator is decreased. It is important to choose the rate to run the controller properly. A good rule of thumb is to run the controller about 10 times faster than the time constant of the plant. The control system should make sure the actuator signal remains in the appropriate range. E.g., you do not want to increment an actuator output of 255 and get 0! The incremental controller is usually slow, but it has good accuracy and is very stable.

The objective of this incremental control system is to control the speed, *X*, of a DC motor shown in Figure 10.15. The actuator uses PWM to apply variable power to the motor. A tachometer is used to measure speed, *X'*.

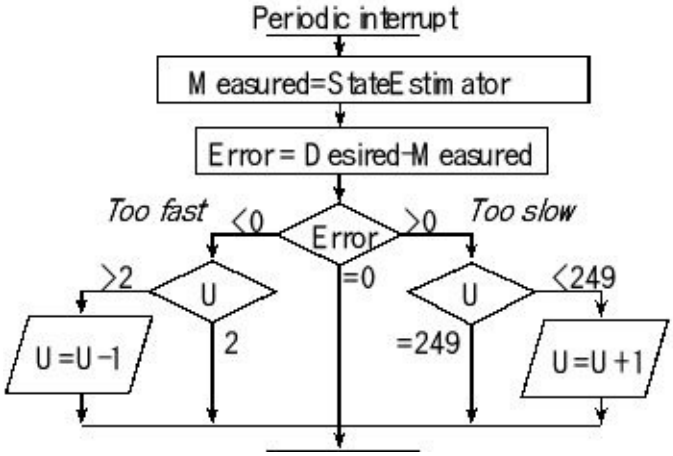


Figure 10.15. Flowchart of a position controller implemented using incremental control.

An incremental control algorithm simply adds or subtracts a constant from *U* depending on the sign of the error. In other words, if *X* is too slow then *U* is incremented and if *X* is too fast then *U* is decremented. It is important to choose the proper rate at which the incremental control software is executed. If it is executed too many times per second, then the actuator will saturate resulting in a Bang-Bang system. If it is not executed often enough then the system will not respond quickly to changes in the physical plant or changes in *X**. In this incremental controller we add or subtract "1" from the actuator, but a value larger than "1" would have a faster response at the expense of introducing oscillations.

Common error: An error will occur if the software does not check for overflow and underflow after U is changed.

Observation: If the incremental control algorithm is executed too frequently, then the resulting system behaves like a simple bang-bang controller.

Observation: Many control systems operate well when the control equations are executed about 10 times faster than the step response time of the physical plant.

Assume the function **SE()** returns measured speed. Program 10.2 uses a periodic interrupt so that the incremental controller runs in the background. The interrupt period is selected to be about 10 times smaller than the time constant of the physical plant. The optimal controller rate depends on the significance of the ±1 value added to *U*. Experimental testing may be required to select an optimal controller rate, trading off response time for stability. Even though the position variables **X** and **Xstar** may be unsigned, the error calculation **E** will be signed.

```
int32_t X,Xstar,E;       // speed, fixed-point in the same format
int32_t U;
void Timer0A_Handler(void){
  X = SE();           // estimated speed
  E = Xstar-X;        // error
  if(E < -10)     U--; // decrease if too fast
```

```
  else if(E > 10) U++; // increase if too slow
                  // leave as is if close enough
  if(U<2)  U=2;       // underflow (minimum PWM)
  if(U>249) U=249;    // overflow (maximum PWM)
  PWM0A_Duty(U);      // output to actuator, Section 2.8
  TIMER0_ICR_R = 0x01;  // acknowledge timer0A periodic timer
}
```

Program 10.2. Incremental control software for a DC motor.

Checkpoint 10.3: In what ways would the controller behave differently if **-10** and **+10** were to be changed to **0** ?

Checkpoint 10.4: What happens if the interrupt period is too small (i.e., executes too frequently)?

Observation: It is a good debugging strategy to observe the assembly listing generated by the compiler when performing calculations on variables of mixed types (signed/unsigned, char/short).

Observation: Incremental control will work moderately well (accurate and stable) for an extremely wide range of applications. Its only short-coming is that the controller response time can be quite slow.

10.6. PID Controllers

10.6.1. General Approach to a PID Controller

The simple controllers presented in the last section are easy to implement, but will have either large errors or very slow response times. In order to make a faster and more accurate system, we can use linear control theory to develop the digital controller. There are three components of a proportional integral derivative **PID controller**.

$$U(t) = K_p E(t) + \int_0^t K_i E(\tau) d\tau + K_d \frac{dE(t)}{dt}$$

The error, *E(t)*, is defined as the present set-point, *X*(t)*, minus the measured value of the controlled variable, *X'(t)*. See Figure 10.16.

$$E(t) = X^*(t) - X'(t)$$

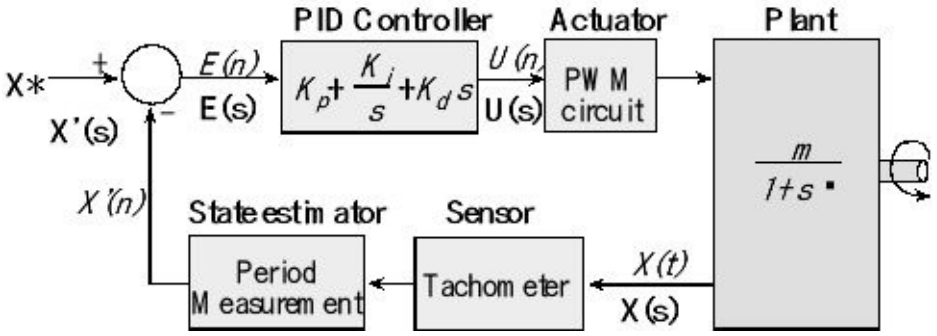


Figure 10.16. Block diagram of a linear control system in the frequency domain.

The PID controller calculates its output by summing three terms. The first term is proportional to the error. The second is proportional to the integral of the error over time, and the third is proportional to the rate of change (first derivative) of the error term. The values of K_p, K_i and K_d are design parameters and must be properly chosen in order for the control system to operate properly. The proportional term of the PID equation contributes an amount to the control output that is directly proportional to the current process error. The gain term K_p adjusts exactly how much the control output response should change in response to a given error level. The larger the value of $K_{p,}$ the greater the system reaction to differences between the set-point and the actual state variable. However, if K_p is too large, the response may exhibit an undesirable degree of oscillation or even become unstable. On the other hand, if K_p is too small, the system will be slow or unresponsive. An inherent disadvantage of proportional-only control is its inability to eliminate the steady state errors (offsets)

that occur after a set-point change or a sustained load disturbance.

The integral term converts the first order proportional controller into a second order system capable of tracking process disturbances. It adds to the controller output a factor that takes corrective action for any changes in the load level of the system. This integral term is scaled to the sum of all previous process errors in the system. As long as there is a process error, the integral term will add more amplitude to the controller output until the sum of all previous errors is zero. Theoretically, as long as the sign of K_i is correct, any value of K_i will eliminate offset errors. But, for extremely small values of K_i, the controlled variables will return to the set-point very slowly after a load upset or set-point change occurs. On the other hand, if K_i is too large, it tends to produce oscillatory response of the controlled process and reduces system stability. The undesirable effects of too much integral action can be avoided by proper tuning (adjusting) the controller or by including derivative action which tends to counteract the destabilizing effects.

The derivative action of a PID controller adds a term to the controller output scaled to the slope (rate of change) of the error term. The derivative term "anticipates" the error, providing a greater control response when the error term is changing in the wrong direction and a dampening response when the error term is changing in the correct direction. The derivative term tends to improve the dynamic response of the controlled variable by decreasing the process setting time, the time it takes the process to reach steady state. But if the process measurement is noisy, that is, if it contains high-frequency random fluctuations, then the derivative of the measured (controlled) variable will change wildly, and derivative action will amplify the noise unless the measurement is filtered.

Checkpoint 10.5: What happens in a PID controller if the sign of K_i is incorrect?

We can also use just some of the terms. For example a proportional/integrator (PI) controller drops the derivative term. We will analyze the digital control system in the frequency domain. Let *X(s)* be the Laplace transform of the state variable *x(t)***.** Let *X* (s)* be the Laplace transform of the desired state variable *x*(t)***.** Let *E(s)* be the Laplace transform of the error

$$E(s) = X^*(s) - X(s)$$

Let *G(s)* be the transfer equation of the PID linear controller. PID controllers are unique in this aspect. In other words, we cannot write a transfer equation for a bang-bang, incremental or fuzzy logic controller.

$$G(s) = K_p + K_d s + \frac{K_i}{s}$$

Let *H(s)* be the transfer equation of the physical plant. If we assume the physical plant (e.g., a DC motor) has a simple single pole behavior, then we can specify its response in the frequency domain with two parameters. *m* is the DC gain and *t* is its time constant. The transfer function of this simple motor is

$H(s) = m/(1+ts)$

The overall gain of the control system is

$$\frac{X(s)}{X^*(s)} = \frac{G(s)H(s)}{1+G(s)H(s)}$$

Theoretically we can choose controller constants, K_p K_i and K_d, to create the desired controller response. Unfortunately it can be difficult to estimate *m* and *t*. If a load is applied to the motor, then *m* and *t* will change.

To simplify the PID controller implementation, we break the controller equation into separate proportion, integral and derivative terms. I.e., let

$$U(t) = P(t) + I(t) + D(t)$$

where *U(t)*is the actuator output, and *P(t)*, *I(t)* and *D(t)* are the proportional, integral and derivative components respectively. The proportional term makes the actuator output linearly related to the error. Using a proportional term creates a control system that applies more energy to the plant when the error is large. To implement the proportional term, we simply convert it to discrete time.

$$P(t) = K_p E(t) \quad \Rightarrow \quad P(n) = K_p E(n)$$

where the index "*n*" refers to the discrete time input of *E(n)* and output of *P(n)*.

Observation: In order to develop digital signal processing equations, it is imperative that the control system be executed on a regular and periodic rate.

Common error: If the sampling rate varies, then controller errors will occur.

The integral term makes the actuator output related to the integral of the error. Using an integral term often will improve the steady state error of the control system. If a small error accumulates for a long time, this term can get large. Some control systems put upper and lower bounds on this term, called anti-reset-windup, to prevent it from dominating the other terms. The implementation of the integral term requires the use of a discrete integral or sum. If *I(n)* is the current control output, and *I(n-1)* is the previous calculation, the integral term is simply

$$I(t) = \int_0^t K_i E(\tau)\,d\tau \quad \Rightarrow \quad I(n) = \sum_1^n K_i E(n)\Delta t = I(n-1) + K_i E(n)\Delta t$$

where Δt is the sampling rate of *E(n)*.

The derivative term makes the actuator output related to the derivative of the error. This term is usually combined with either the proportional and/or integral term to improve the transient response of the control system. The proper value of K_d will provide for a quick response to changes in either the set point or loads on the physical plant. An incorrect value may create an overdamped (very slow response) or an underdamped (unstable oscillations) response. There are a couple of ways to implement the discrete time derivative. The simple approach is

$$D(t) = K_d \frac{dE(t)}{dt} \quad \Rightarrow \quad D(n) = K_d \frac{E(n) - E(n-1)}{\Delta t}$$

In practice, this first order equation is quite susceptible to noise. Figure 10.17 shows a sequence of *E(n)* with some added noise. Notice that huge errors occur when the above equation is used to calculate derivative.

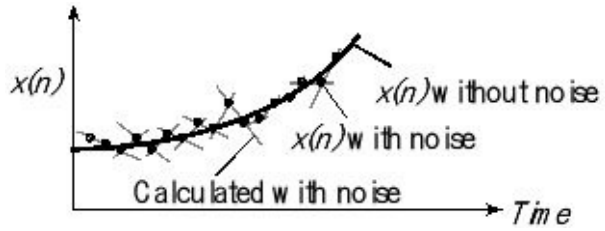


Figure 10.17. Illustration of the effect noise plays on the calculation of discrete derivative.

In most practical control systems, the derivative is calculated using the average of two derivatives calculated across different time spans. For example

$$D(n) = K_d \left(\frac{1}{2} \frac{E(n) - E(n-3)}{3\Delta t} + \frac{1}{2} \frac{E(n-1) - E(n-2)}{\Delta t} \right)$$

that simplifies to

$$D(n) = K_d \left(\frac{E(n) + 3E(n-1) - 3E(n-2) - E(n-3)}{6\Delta t} \right)$$

Linear regression through multiple points can yield the slope and yet be immune to noise.

Checkpoint 10.6: How is the continuous integral related to the discrete integral?

Checkpoint 10.7: How is the continuous derivative related to the discrete derivative?

10.6.2. Design Process for a PID Controller

The first design step is the analysis phase, where we determine specifications such as range, accuracy, stability, and response time for our proposed control system. A data acquisition system will be used to estimate the state variables. Thus, its range, accuracy and response time must be better than the desired specifications of the control system. We can use time-based techniques using input capture, or develop an ADC-based state estimator. In addition, we need to design an actuator to manipulate the state variables. It too must have a range and response time better than the controller specifications. The **actuator resolution** is defined as the smallest reliable change in output. For example, a 100 Hz PWM output generated by a 1 µsec clock has 10,000 different outputs. For this actuator, the actuator resolution is MaxPower/10000. We wish to relate the actuator performance to the overall objective of controller accuracy. Thus, we need to map the effect on the state

variable caused a change in actuator output. This change in state variable should be less than or equal to the desired controller accuracy.

After the state estimator and actuator are implemented, the controller settings (K_P, K_I and K_D) must be adjusted so that the system performance is satisfactory. This activity is referred to as **controller tuning** or **field tuning**. If you perform controller tuning by guessing the initial setting then adjusting them by trial and error, it can be tedious and time consuming. Thus, it is desirable to have good initial estimates of controller settings. A good first setting may be available from experience with similar control loops. Alternatively, initial estimates of controller settings can be derived from the transient response of the physical plant. A simple open-loop method, called the **process reaction curve approach**, was first proposed by Ziegler/Nichols and Cohen/Coon in 1953. In this discussion, the term "process" as defined by Ziegler/Nichols means the same thing as the "physical plant" described earlier in this chapter. This open-loop method requires only that a single step input be imposed on the process. The process reaction method is based on a single experimental test that is made with the controller in the manual mode. A small step change, ΔU, in the controller output is introduced and the measured process response is recorded, as shown in Figure 10.18. To obtain parameters of the process, a tangent is drawn to the process reaction curve at its point of maximum slope (at the inflection point). This slope is R, which is called the **process reaction rate**. The intersection of this tangent line with the original base line gives an indication of **L**, the process lag. L is really a measure of equivalent dead time for the process. If the tangent drawn at the inflection point is extrapolated to a vertical axis drawn at the time when the step was imposed, the amount by which this value is below the horizontal base line will be represented by the product $L*R$. ΔT is the time step for the digital controller. It is recommended to run P and PI controllers with $\Delta T = 0.1L$, and a PID controller at a rate 20 times faster ($\Delta T = 0.05L$.) Using these parameters, Ziegler and Nichol proposed initial controller settings as

Proportional Controller

$K_p = \Delta U/(L*R)$

Proportional-Integral Controller

$K_p = 0.9\ \Delta U/(L*R)$

$K_i = K_p/(3.33L)$

Proportional-Integral-Derivative Controller

$K_p = 1.2\ \Delta U/(L*R)$

$K_i = 0.5\ K_p/L$

$K_d = 0.5\ K_p\ L$

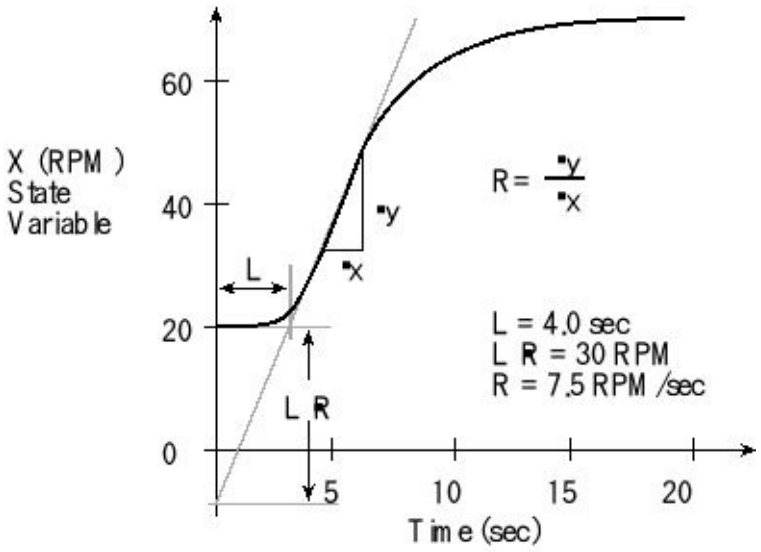


Figure 10.18. A process reaction curve used to determine controller settings.

Checkpoint 10.8: Are the Ziegler/Nichol equations consistent from a dimensional analysis perspective? In other words, are the units correct?

The **response time** is the delay after X^* is changed for the system to reach a new constant state. **Steady state controller accuracy** is defined as the average difference between X^* and X'. **Overshoot** is defined as the maximum positive error that occurs when X^* is increased. Similarly, **undershoot** is defined as the maximum negative error that occurs when X^* is decreased. During the testing phase, it is appropriate to add minimally intrusive debugging software that specifically measures performance parameters, such as response time, accuracy, overshoot, and undershoot. In addition, we can add instruments that allow us to observe the individual *P(n)*, *I(n)* and *D(n)* components of the PID equation and their relation to controller error *E(n)*.

Once the initial parameters are selected, a simple empirical method can be used to fine-tune the controller. This empirical approach starts with proportional term (K_p). As the proportional term is adjusted up or down, evaluate the quickness and smoothness of the controller response to changes in set-point and to changes in the load. K_p is too big if the actuator saturates both at the maximum and minimum after X^* is changed. The next step is to adjust the integral term (K_i) a little at a time to improve the steady state controller accuracy without adversely affecting the response time. Don't change both K_p and K_i at once. Rather, you should vary them one at a time. If the response time, overshoot, undershoot and accuracy are within acceptable limits, then a PI controller is adequate. On the other hand, if accuracy and response are OK but overshoot and undershoot are unacceptable, adjust the derivative term (K_d) to reduce the overshoots and undershoots.

We will design a proportional-integral motor control system. The overall objective is to control the speed of an object with an accuracy of 0.1 RPM and a range of 0 to 100

RPM as shown in Figure 10.1. Let X^* be the desired state variable. In this example, X^* will be a decimal fixed-point number and is set by the main program. Let X' be the estimated state variable that comes from the **state estimator,** which encodes the current position as the period of a squarewave, interfaced to an input capture pin. The period output of the sensor is linearly related to the position X with a fixed offset. The accuracy of the state estimator needs to match the 0.1 RPM specification of the controller. If p is the measured period in 0.1 ms and X' is the estimated speed in 0.1 RPM, the state estimator measures the period and calculates X'.

$X' = p\text{-}100$

Let U be the actuator control variable ($100 \le U \le 19900$). This system uses **pulse width modulation** with a 100 Hz squarewave that applies energy to the physical plant as shown in Figure 10.19. U will be the number of clock cycles (out of 20000) that the output is high. There is an external friction force slowing down on the motor. The PWM output from the computer creates a force causing the motor to spin faster.

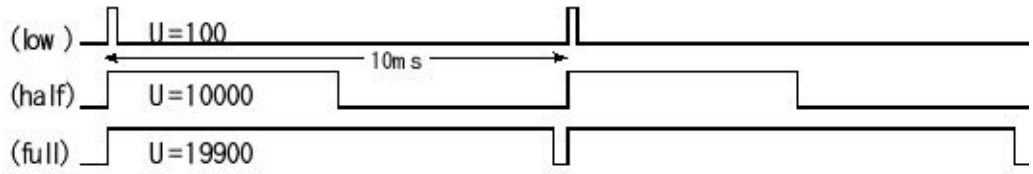


Figure 10.19. Pulse width modulated actuator signals.

The process reaction curve shown previously in Figure 10.18 was measured for this system after the actuator was changed from 250 to 2000, thus ΔU is 1750 (units of clock cycles). From Figure 10.18, the lag L is 4.0 sec and the process reaction rate R is 7.5RPM/sec. The controller rate is selected to be about 10 times faster than the lag L, so ΔT= 0.4 sec . In this way, the controller runs at a rate faster than the physical plant. We calculate the initial PI controller settings using the Ziegler/Nichol equations.

$K_p = 0.9\,\Delta U/(L \cdot R) = 0.9*1750/(4.0*7.5) = 52.5$ cycles/RPM
$K_i = K_p/(3.33L) = 52.5/(3.33*4.0) = 3.94144$ cycles/RPM/sec

We will execute the PI control equation once every 0.4 second. X^* and X' are decimal fixed-point numbers with a resolution of 0.1 RPM. The constant 52.5 is expressed as 105/2. The extra divide by 10 handles the decimal fixed-point representation of X^* and X'.

$P(n) = K_p \cdot (X^*\text{-}X')/10 = 105 \cdot (X^*\text{-}X')/20$

We will also execute the integral control equation once every 0.4 second. Binary fixed-point is used to approximate 1.57658 as 101/64.

$I(n) = I(n\text{-}1) + K_i \cdot (X^*\text{-}X') \cdot \Delta T/10$
$= I(n\text{-}1) + 3.94144 \cdot (X^*\text{-}X') \cdot 0.4/10 = I(n\text{-}1) + 101 \cdot (X^*\text{-}X')/640$

Program 10.3 shows an interrupt service handler, which runs at10 kHz. The handler will establish the current **Time** in 0.1 ms. After 4000 interrupts (0.4 second), the control algorithm is implemented.

```
uint32_t Time; // Time in 0.1 msec
int32_t X;           // Estimated speed in 0.1 RPM, 0 to 1000
int32_t Xstar;       // Desired speed in 0.1 RPM, 0 to 1000
int32_t E;           // Speed error in 0.1 RPM, -1000 to +1000
int32_t U,I,P;       // Actuator duty cycle, 100 to 19900 cycles
uint32_t Cnt;  // incremented every 0.1 msec
uint32_t Told; // used to measure period
void Timer0A_Handler(void){
  Time++;           // used to measure period
  if((Cnt++)==4000){ // every 0.4 sec
    Cnt = 0;        // 0<X<100, 0<Xstar<100, 100<U<19900
    E = Xstar-X;
    P = (105*E)/20;
    I = I+(101*E)/640;
    if(I < -500) I=-500;  // anti-reset windup
    if(I > 4000) I=4000;
    U = P+I;            // PI controller has two parts
    if(U < 100) U=100;   // Constrain actuator output
    if(U>19900) U=19900;
    PWM0A_Duty(U);      // output to actuator, Section 2.8
  }
  TIMER0_ICR_R = 0x01;   // acknowledge timer0A periodic timer
}
```

Program 10.3. PI control software.

Checkpoint 10.9: What is the output *U* of the controller if the speed *X* is much greater than the set-point *X**? In this situation, what does the object do?

Observation: PID control will work extremely well (fast, accurate and stable) if the physical plant can be described with a set of linear differential equations.

10.7. Fuzzy Logic Control

There are a number of reasons to consider fuzzy logic approach to control. It is much simpler than PID systems. It will require less memory and execute faster. In other words, an 8-bit fuzzy system may perform as well (same steady state error and response time) as a 16-bit PID system. When complete knowledge about the physical plant is known, then a good PID controller can be developed. Since the fuzzy logic control is more robust (still works even if the parameter constants are not optimal), then the fuzzy logic approach can be used when complete knowledge about the plant is not known or can change dynamically. Choosing the proper PID parameters requires knowledge about the plant. The fuzzy logic approach is more intuitive, following more closely to the way a "human" would control the system. It is easy to modify an existing fuzzy control system into a new problem. The framework allows rapid prototyping.

Fuzzy logic was conceived in the mid-1960s by Lotfi Zadeh while at the University of California at Berkeley. However, the first commercial application didn't come until 1987, when the Matsushita Industrial Electric used it to control the temperature in a shower head. Named after the nineteenth-century mathematician George Boole, Boolean logic is an algebra where values are either true or false. This algebra includes operations of AND OR and NOT. Fuzzy logic is also an algebra, but where conditions may exist in the continuum between true and false. While Boolean logic defines two states, 8-bit fuzzy logic consists of 256 states all the way from "not at all" (0) to "definitely true" (255). "128" means half way between true and false. The fuzzy logic algebra also includes the operations of AND OR and NOT. A **fuzzy membership set**, a **fuzzy variable**, and a **fuzzy set** all refer to the same entity, which is a software variable describing the level of correctness for a condition within fuzzy logic. If we have a fuzzy membership set for the condition "hungry", then as the value of hungry moves from 0 to 255, the condition "hungry" becomes more and more true.

....0.....32.....64.....96.....128.....160.....192.....224.....255
Not at all ... a little bit ... somewhat ... mostly ... pretty much ... definitely

The design process for a fuzzy logic controller solves the following eight components. These components are listed in the order we would draw a data flow graph, starting with the state variables, progressing through the controller, and ending with the actuator output.

• The **Physical plant** has *real state variables*.

• The **Data Acquisition System** monitors these signals creating the *estimated state variables*.

- The **Preprocessor** may calculate relevant parameters called *crisp inputs*.
- **Fuzzification** will convert crisp inputs into *input fuzzy membership sets*.
- The **Fuzzy Logic** is a set of rules that calculate *output fuzzy membership sets*.
- **Defuzzification** will convert output sets into *crisp outputs*.
- The **Postprocessor** modify crisp outputs into a more convenient format.
- The **Actuator System** affects the Physical plant based on these output.

We will work through the concepts of fuzzy logic by considering examples of how we as humans control things like driving a car at a constant speed. During the initial stages of the design, we study the **physical plant** and decide which state variables to consider. For example, if we wish to control speed, then speed is obviously a state variable, but it might be also useful to know other forces acting on the object such as gravity (e.g., going up and down hills), wind speed and friction (e.g., rain and snow on the roadway). The purpose of the **data acquisition system** is to accurately measure the state variables. It is at this stage that the system converts physical signals into binary numbers to be processed by the software controller. We have seen two basic approaches in this book for this conversion: the measurement of period/frequency using input capture and the analog to digital conversion using an ADC. The **preprocessor** calculates **crisp inputs**, which are variables describing the input parameters in our software having units (like miles/hr). For example, if we measured speed, then some crisp inputs we might calculate would include speed error, and acceleration. Just like the PID controller, the accuracy of the data acquisition system must be better than the desired accuracy of the control system as a whole.

The next stage of the design is to consider the actuator and postprocessor. It is critical to be able to induce forces on the physical plant in a precise and fast manner. The step response of the actuator itself (time from software command to the application of force on the plant) must be faster than the step response of the plant (time from the application of force to the change in state variable.) Consider the case where we wish to control the temperature of a pot of water using a stove. The speed of the actuator is the time between turning the stove on and the time when heat is applied to the pot. The actuator on a gas stove is much faster than the actuator on an electric stove. The resolution of an actuator is the smallest change in output it can reliably generate. Just like the PID controller, the resolution of the actuator (converted into equivalent units on the input) must be smaller than the desired accuracy of the control system as a whole. A **crisp output** is a software variable describing the output parameters having units (like watts, Newtons, dynes/cm^2 etc.). The **postprocessor** converts the crisp output into a form that can be directly output to the actuator. The postprocessor can verify the output signals are within the valid range of the actuator. One of the advantages of fuzzy logic design is the usage of human intuition. Think carefully about how you control the actuator (gas pedal) when attempting to drive a car at a constant speed. There is no parameter in your brain specifying the exact position of the pedal (e.g., 50% pressed, 65% pressed etc.), unless of course you are city taxicab driver (where your brain allows two actuator states: full gas and full brake.) Rather, what your brain creates as actuator commands

are statements like “press the pedal little harder” and “press the pedal a lot softer.” So, the crisp output of fuzzy logic controller might be change in pedal pressure ΔU, and the postprocessor would calculate $U = U+\Delta U$, then check to make sure U is within an acceptable range.

We continue the design of a fuzzy logic controller by analyzing its crisp inputs. As a design step, we create a list of true/false conditions that together describe the current state of the physical plant. In particular, we define **input fuzzy membership sets**, which are fuzzy logic variables describing conditions related to the state of the physical plant. These fuzzy variables do not need to be orthogonal. In other words, it is acceptable to have variables that are related to each other. When designing a speed controller, we could define multiple fuzzy variables referring to similar conditions, such as **WayTooFast** , **Fast** , and **LittleBitFast** . Given the scenario where we are driving too fast, there should be generous overlap in conditions, such that two or even three fuzzy sets are simultaneously partially true. On the other hand, it is important that the entire list of input membership fuzzy sets, when considered as an ensemble, form a complete definition of the status of the physical plant. For example, if we are attempting to drive a car at a constant speed, then **SlowingUp** , **GoingSteady** , and **SpeedingUp** might be input fuzzy variables describing the car’s acceleration. **Fuzzification** is the mathematical step converting the crisp inputs into input fuzzy membership sets. When implementing fuzzy logic explicitly with C code, we will have available the full set of AND, OR, NOT fuzzy logic operations.

The heart of a fuzzy logic controller is the **fuzzy logic** itself, which is set of logic equations that calculate fuzzy outputs as a function of fuzzy inputs. An **output fuzzy membership set**is a fuzzy logic variable describing a condition related to the actuator. **QuickStop** , **SlowDown** , **JustRight** , **MorePower** , and **MaxPower** are examples of output fuzzy variables that might be used to describe the action to perform on the gas pedal. Like input fuzzy variables, output fuzzy variables exist in the continuum from definitely false (0) to definitely true (1). Just like the input specification, it is also important to create a list of output membership fuzzy sets, when considered as an ensemble, form a complete characterization of what we wish to be able to do with the actuator. We write fuzzy logic equations using AND and OR functions in a way similar to Boolean logic. The fuzzy logic AND is calculated as the minimum value of the two inputs, and the fuzzy logic OR is calculated as the maximum value of the two inputs. The design of the rules, like the other aspects of fuzzy control, follows the human intuition.

SlowDown = **WayTooFast** + **SpeedingUp***LittleBitFast**

Checkpoint 10.10: If **WayTooFast** is 50, **SpeedingUp** is 40, and **LittleBitFast** is 60, then what would be the calculated value for **SlowDown** ?

The **defuzzification** stage of the controller converts the output fuzzy variables into

crisp outputs. Although any function could be used, an effective approach is to use a weighted average. Consider the case where the pedal pressure *U* varies from 0 to 100, thus the crisp output *ΔU*can take on values from -100 to +100. We think about what crisp output we want if just **QuickStop** were to be true. In this case, we wish to make *ΔU*equal to -100. We then define crisp output values for **SlowDown** , **JustRight** , **MorePower** , and **MaxPower** as -10, 0, +10, and +100 respectfully. We can combine the five factors using a weighted average.

$$\Delta U = \frac{-100*\text{QuickStop}-10*\text{SlowDown}+10*\text{MorePower}+100*\text{MaxPower}}{\text{QuickStop}+\text{SlowDown}+\text{JustRight}+\text{MorePower}+\text{MaxPower}}$$

Because the fuzzy controller is modular, we begin by testing each of the modules separately. The system-level testing of a fuzzy logic controller follows a procedure similar to the PID controller tuning. Debugging instruments can be added to record the crisp inputs, fuzzy inputs, fuzzy outputs, and crisp outputs during the real-time operation of the system. Fuzzification parameters are adjusted so that the status of the plant is captured in the set of values contained in the fuzzy input variables. Next, the rules are adjusted so that fuzzy output variables properly describe what we want to do with the actuator. Lastly, the defuzzification parameters are adjusted so the proper crisp outputs are created.

Next we will design a fuzzy logic motor controller. The actuator is a PWM (Figure 10.2). The power to the motor is controlled by varying the 8-bit PWM duty cycle. The motor speed is estimated with a tachometer connected to an input capture pin.

Our system has:

- two control inputs
  - *S** the desired motor speed in RPM
  - *S'* the current estimated motor speed RPM
- one control output
  - *N* the digital value that we write to the PWM

To utilize 8-bit math, we change the units of speed to 1000/256=3.90625 RPM.

*T** =(256•*S**)/1000 the desired motor speed in 3.9 RPM

T' =(256•*S'*)/1000 the current estimated motor speed 3.9 RPM

For example, if the desired speed is 500 RPM, then *T** will be 128. Notice that the estimated speed, *T'*, is measured by the input capture pin. In other words, the control system functions (estimate state variables, control equation calculations, and actuator output) are performed on a regular and periodic basis, every Δ*t* time units. This allows signal processing techniques to the used. We will let *T'(n)* refer to the current measurement and *T'(n-1)* refer to the previous measurement, i.e., the one measured Δ*t* time ago.

In the fuzzy logic approach, we begin by considering how a "human" would control the motor. Assume your hand were on a joystick (or your foot on a gas pedal) and consider how you would adjust the joystick to maintain a constant speed. We select

crisp inputs and outputs to base our control system on. It is logical to look at the error and the change in speed when developing a control system. Our fuzzy logic system will have two crisp inputs

E = T-T'* the error in motor speed in 3.9rpm
D = T'(n)-T'(n-1) the change in motor speed in 3.9rpm/time

Notice that if we perform the calculations of *D* on periodic intervals, then *D* will represent the derivative of *T'*, *dT'/dt*. *T** and *T'* are 8-bit unsigned numbers, so the potential range of *E* varies from -255 to +255. Errors beyond ±127 will be adjusted to the extremes +127 or -128 without loss of information.

```
int8_t static Subtract(uint8_t N, uint8_t M){
// returns N-M
uint32_t N16,M16;
int32_t Result16;
   N16 = N;            // Promote N,M
   M16 = M;
   Result16 = N16-M16;   // -255≤Result16≤+255
   if(Result16<-128) Result16 = -128;
   if(Result16>127)  Result16 = 127;
   return(Result16);}
```

Program 10.4. Subtraction with overflow/underflow checking.

These are the global definitions of the input signals and fuzzy logic crisp input,

```
uint8_t Ts;     // Desired Speed in 3.9 rpm units
uint8_t T;      // Current Speed in 3.9 rpm units
uint8_t Told;   // Previous Speed in 3.9 rpm units
int8_t D;      // Change in Speed in 3.9 rpm/time units
int8_t E;       // Error in Speed in 3.9 rpm units
```

Program 10.5. Inputs and crisp inputs.

Common error: Neglecting overflow and underflow can cause significant errors.

The need for the special **Subtract** function can be demonstrated with the following example:

```
   E = Ts-T;  // if Ts=200 and T=50 then E will be -106!!
```

This function can be used to calculate both *E* and *D*,

```
void CrispInput(void){
   E    = Subtract(Ts,T);
   D    = Subtract(T,Told);
   Told = T;}     // Set up Told for next time
```

Program 10.6. Calculation of crisp inputs.

Now, if **Ts=200** and **T=50** then **E** will be +127. To control the actuator, we could simply choose a new PWM value *N* as the crisp output. Instead, we will select, ΔN that is the change in *N*, rather than *N* itself because it better mimics how a "human" would control it. Again, think about how you control the speed of your car when driving. You do not adjust the gas pedal to a certain position, but rather make small or large changes to its position in order to speed up or slow down. Similarly, when controlling the temperature of the water in the shower, you do not set the hot/cold controls to certain absolute positions. Again you make differential changes to affect the "actuator" in this control system. Our fuzzy logic system will have one crisp output:

ΔN	change in output, $N=N+\Delta N$ in PWM units

Next we introduce fuzzy membership sets that define the current state of the crisp inputs and outputs. Fuzzy membership sets are variables that have true/false values. The value of a fuzzy membership set ranges from definitely true (255) to definitely false (0). For example, if a fuzzy membership set has a value of 128, you are stating the condition is half way between true and false. For each membership set, it is important to assign a meaning or significance to it. The calculation of the input membership sets is called **Fuzzification**. For this simple fuzzy controller, we will define 6 membership sets for the crisp inputs:

Slow	True if the motor is spinning too slow
OK	True if the motor is spinning at the proper speed
Fast	True if the motor is spinning too fast
Up	True if the motor speed is getting larger
Constant	True if the motor speed is remaining the same
Down	True if the motor speed is getting smaller.

We will define 3 membership sets for the crisp output:

Decrease	True if the motor speed should be decreased
Same	True if the motor speed should remain the same
Increase	True if the motor speed should be increased

The fuzzy membership sets are usually defined graphically, but software must be written to actually calculate each. In this implementation, we will define three adjustable thresholds, TE, TD and TN. These are software constants and provide some fine tuning to the control system. We will set each threshold to 20. If you build one of these fuzzy systems, try varying one threshold at a time and observe the system behavior (steady state controller error and transient response.) If the error, E, is -5 (3.9rpm units), the fuzzy logic will say that *Fast* is 64 (25% true), *OK* is 192 (75% true), and Slow is 0 (definitely false.) If the error, E, is +21 (in 3.9rpm units), the fuzzy logic will say that *Fast* is 0 (definitely false), *OK* is 0 (definitely false), and *Slow* is 255 (definitely true.) TE is defined to be the error (e.g., 20 in 3.9 rpm units is 78 rpm) above which we will definitely consider the speed to be too fast.

Similarly, if the error is less than -TE, then the speed is definitely too slow.

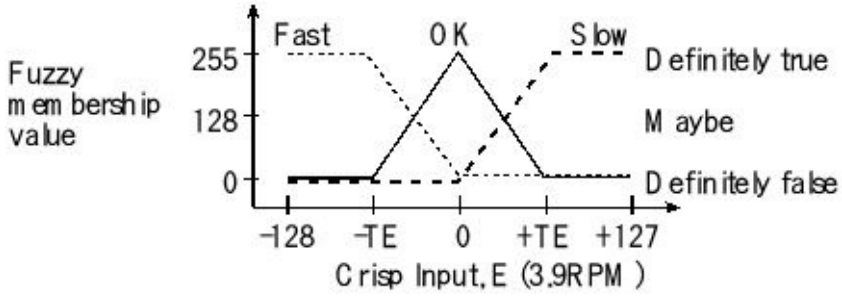


Figure 10.20. Fuzzification of the error input.

In this fuzzy system, the input membership sets are continuous piece-wise linear functions. Also, for each crisp input value, *Fast*, *OK*, *Slow* sum to 255. In general, it is possible for the fuzzy membership sets to be nonlinear or discontinuous, and the membership values do not have to sum to 255. The other three input fuzzy membership sets depend on the crisp input, *D*. *TD* is defined to be the change in speed (e.g., 20 in 3.9 rpm/time units is 78 rpm/time) above which we will definitely consider the speed to be going up. Similarly, if the change in speed is less than -*TD*, then the speed is definitely going down.

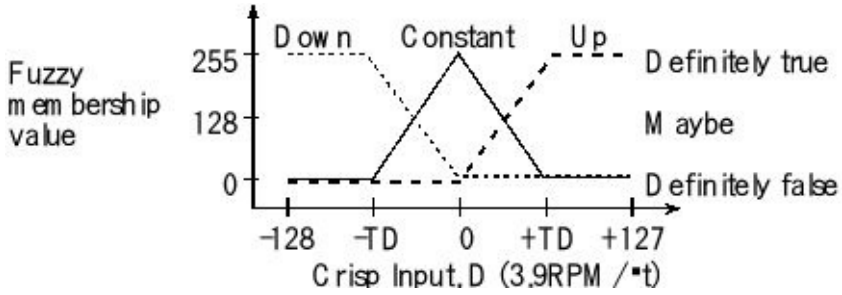


Figure 10.21. Fuzzification of the acceleration input.

In C, we could define a fuzzy function that takes the crisp inputs and calculates the fuzzy membership set values. Again *TE* and *TD* are software constants that will affect the controller error and response time.

```
#define TE 20
uint8_t Fast, OK, Slow, Down, Constant, Up;
#define TD 20
uint8_t Increase,Same,Decrease;
#define TN 20
void InputMembership(void){
    if(E <= -TE) {          // E≤-TE
      Fast = 255;
      OK = 0;
      Slow = 0;}
    else
      if(E < 0){            // -TE<E<0
```

```
      Fast = (255*(-E))/TE;
      OK = 255-Fast;
      Slow = 0;}
    else
      if(E < TE){          //  0<E<TE
        Fast = 0;
        Slow = (255*E)/TE;
        OK = 255-Slow;}
      else {               // +TE≤E
        Fast = 0;
        OK = 0;
        Slow = 255;}
  if(D <= -TD) {           // D≤-TD
    Down = 255;
    Constant = 0;
    Up = 0;}
  else
    if(D < 0){             // -TD<D<0
      Down = (255*(-D))/TD;
      Constant = 255-Down;
      Up = 0;}
    else
      if(D < TD){          // 0<D<TD
        Down = 0;
        Up = (255*D)/TD;
        Constant = 255-Up;}
      else{                // +TD≤D
        Down = 0;
        Constant = 0;
        Up = 255;}
}
```

Program 10.7. Calculation of the fuzzy membership variables in C.

The fuzzy rules specify the relationship between the input fuzzy membership sets and the output fuzzy membership values. It is in these rules that one builds the intuition of the controller. For example, if the error is within reasonable limits and the speed is constant, then the output should not be changed. In fuzzy logic we write:

If *OK* and *Constant* then *Same*

If the error is within reasonable limits and the speed is going up, then the output should be reduced to compensate for the increase in speed. I.e.,

If *OK* and *Up* then *Decrease*

If the motor is spinning too fast and the speed is constant, then the output should be reduced to compensate for the error. I.e.,

If *Fast* and *Constant* then *Decrease*

If the motor is spinning too fast and the speed is going up, then the output should be reduced to compensate for both the error and the increase in speed. I.e.,

If *Fast* and *Up* then *Decrease*

If the error is within reasonable limits and the speed is going down, then the output should be increased to compensate for the drop in speed. I.e.,

If *OK* and *Down* then *Increase*

If the motor is spinning too slowly and the speed is constant, then the output should be increased to compensate for the error. I.e.,

If *Slow* and *Constant* then *Increase*

If the motor is spinning too slowly and the speed is going down, then the output should be increase to compensate for both the error and the drop in speed. I.e.,

If *Slow* and *Down* then *Increase*

These 7 rules can be illustrated in a table form.

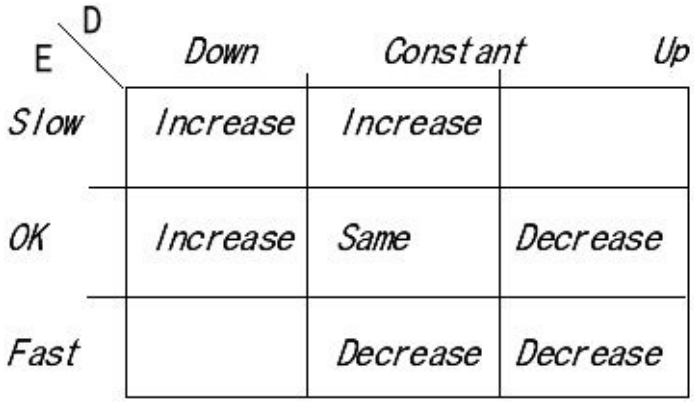


Figure 10.22. Fuzzy logic rules shown in table form.

It is not necessary to provide a rule for all situations. For example, we did not specify what to do if *Fast*&*Down* or for *Slow*&*Up*. Although we could have added (but did not):

If *Fast* and *Down* then *Same*
If *Slow* and *Up* then *Same*

When more than one rule applied to an output membership set, then we can combine the rules:

Same=(*OK*and*Constant*)
Decrease=(*OK*and*Up*)or(*Fast*and*Constant*)or(*Fast*and*Up*)
Increase=(*OK*and*Down*)or(*Slow*and*Constant*)or(*Slow*and*Down*)

In fuzzy logic, the **and** operation is performed by taking the minimum and the **or** operation is the maximum. Thus the C function that calculates the three output fuzzy membership sets is

```
uint8_t static min(uint8_t u1,uint8_t u2){
   if(u1>u2) return(u2);
   else return(u1);}
uint8_t static max(uint8_t u1,uint8_t u2){
   if(u1<u2) return(u2);
   else return(u1);}
void OutputMembership(void){
   Same     = min(OK,Constant);
   Decrease = min(OK,Up)
   Decrease = max(Decrease,min(Fast,Constant));
   Decrease = max(Decrease,min(Fast,Up));
   Increase = min(OK,Down)
   Increase = max(Increase,min(Slow,Constant));
   Increase = max(Increase,min(Slow,Down));}
```

Program 10.8. Calculation of the output fuzzy membership variables in C.

The calculation of the crisp outputs is called **Defuzzification**. The fuzzy membership sets for the output specifies the crisp output, Δ*N*, as a function of the membership value. For example, if the membership set *Decrease* were true (255) and the other two were false (0), then the change in output should be -*TN* (where *TN* is another software constant). If the membership set *Same* were true (255) and the other two were false (0), then the change in output should be 0. If the membership set *Increase* were true (255) and the other two were false (0), then the change in output should be +*TN*.

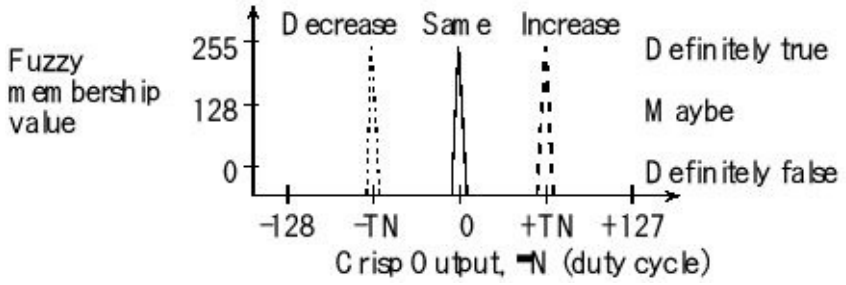


Figure 10.23. Defuzzification of the ΔN crisp output.

In general, we calculate the crisp output as the weighted average of the fuzzy membership sets:

ΔN=(*Decrease*•(-TN) + *Same*•0 +*Increase*•TN)/(*Decrease*+*Same*+*Increase*)

The C compiler will promote the calculations to 32 bits, and perform the calculation using 32-bit signed math that will eliminate overflow on intermediate terms. The output, *dN*, will be bounded in between -*TN* and +*TN*. Thus the C function that calculates the crisp output is

```
int32_t dN;
void CrispOutput(void){
  dN=(TN*(Increase-Decrease))/(Decrease+Same+Increase);
}
```

Program 10.9. Calculation of the crisp output in C.

```
void Timer0A_Handler(void){
 T = SE();           // estimate speed, set T, 0 to 255
 CrispInput();       // Calculate E,D and new Told
 InputMembership();  // Sets Fast,OK,Slow,Down,Constant,Up
 OutputMembership(); // Sets Increase,Same,Decrease
 CrispOutput();      // Sets dN
 N = max(0,min(N+dN,255));
 PWM0A_Duty(N);      // output to actuator, Section 2.8
 TIMER0_ICR_R = 0x01; // acknowledge timer0A periodic timer
}
```

Program 10.10. Periodic interrupt service for fuzzy logic controller.

Observation: Fuzzy logic control will work extremely well (fast, accurate and stable) if the designer has expert knowledge (intuition) of how the physical plant behaves.

10.8. Exercises

10.1 For each term give a definition in 16 words or less.
a) State variable b) State estimator c) Closed loop
d) Transient response e) Stability f) Steady state accuracy
g) Process reaction curve h) Process reaction rate i) Anti-reset windup

10.2 For each control algorithm give a definition in 16 words or less.
a) Open loop b) Bang-bang c) Incremental
d) PID e) Input PI f) Fuzzy logic

10.3 For each Fuzzy Logic term give a definition in 16 words or less.
a) Crisp input b) Fuzzification c) Fuzzy membership set
d) Fuzzy logic e) Defuzzification f) Crisp output

10.4 Briefly explain why it is important to choose the proper update rate for a fuzzy logic controller. In particular, explain what happens to a fuzzy logic controller if the controller is executed too infrequently. Similarly, explain what happens to a fuzzy logic controller if the controller is executed too frequently.

10.5. Assume you have an 8-bit fuzzy logic system like the ones described in this chapter. Write formal descriptions for the complement and exclusive or fuzzy logic operations. Show C code implementations for these two functions.

10.6 The objective of this problem is to use the Ziegler and Nichol approach to develop the PI controller equations that allow an embedded system to control a DC motor. The state variable is speed, which is measured using 16-bit input capture and has a measurement resolution of 1 RPM. The input capture device driver repeatedly updates a global variable, called **Speed** . This 16-bit unsigned variable has units of RPM and a range of 0 to 20000. The microcontroller uses pulse-width modulation to control power to the motor. The controller software writes to a global variable, called **Duty** , which ranges from 0 (0%) to 10000 (100%). The following plot shows an experimental measurement obtained when **Duty** is changed from 2500 to 5000. The desired speed is stored in the global variable, **Desired** , which has the same units as **Speed** . Design a fixed-point PI controller that takes **Speed** and **Desired** as inputs and calculates **Duty** as an output. From the response graph in Figure 10.24, estimate the **L** and **R**parameters of the Ziegler and Nichol method. How often should the controller be executed? Show just the equations (no software or hardware is required), calculating **Duty** as a function of **Speed** and **Desired** .

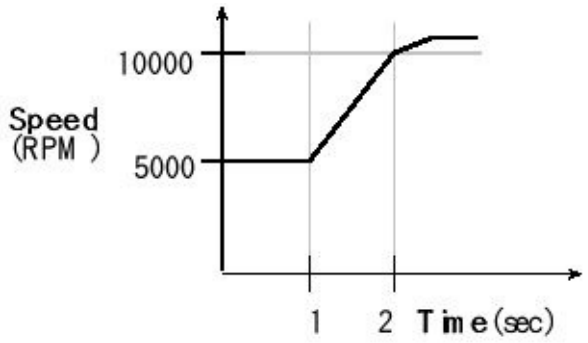


Figure 10.24. A process reaction curve for the DC motor.

10.7 The objective of this problem is to use the Ziegler and Nichol approach to develop the PID controller equations that allow an embedded system to control the DC motor presented in Question 10.6. I.e., work through the steps of Question 10.6 for a PID system.

10.8 Create a definition for Fuzzy Logic complement. Let ~A be the complement of A. Some of these logic equations are valid for Fuzzy Logic and some are not. For each valid equation, present a formal proof of its correctness. For each invalid equation, give a counter example.

a) A*B = B*A **b)** A+B = B+A
c) (A*B)*C = A*(B*C) **d)** (A+B)+C = A+(B+C)
e) (A+B)*C = (A*C)+(B*C) **f)** A + ~A = true
g) A * ~A = false **h)** (A*B)+A = A

Appendix 1. Glossary

1/f noise A fundamental noise in resistive devices arising from fluctuating conductivity. Same as pink noise.

2's complement (see two's complement).

60 Hz noise An added noise from electromagnetic fields caused by either magnetic field induction or capacitive coupling.

accumulator High-speed memory located in the processor used to perform arithmetic or logical functions. The accumulators on the ARM Cortex M are Registers R0 through R12.

accuracy A measure of how close our instrument measures the desired parameter referred to the NIST.

acknowledge Clearing the interrupt flag bit that requested the interrupt.

active thread A thread that is in the ready-to-run circular linked list. It is either running or is ready to run.

actuator Electro-mechanical or electro-chemical device that allows computer commands to affect the external world.

ADC Analog to digital converter, an electronic device that converts analog signals (e.g., voltage) into digital form (i.e., integers).

address bus A set of digital signals that connect the CPU, memory and I/O devices, specifying the location to read or write for each bus cycle. See also control bus and data bus.

aging A technique used in priority schedulers that temporarily increases the priority of low priority treads so they are run occasionally. (See starvation)

aliasing When digital values sampled at $\mathbf{f_S}$ contain frequency components above ½ $\mathbf{f_S}$, then the apparent frequency of the data is shifted into the 0 to ½ $\mathbf{f_S}$ range. See Nyquist Theory.

alternatives The total number of possibilities. E.g., an 8-bit number scheme can represent 256 different numbers. An 8-bit digital to analog converter (DAC) can generate 256 different analog outputs.

anode The positive side of a diode. Current enters the anode side of a diode. Contrast with cathode.

answer modem The device that receives the telephone call.

anti-reset-windup Establishing an upper bound on the magnitude of the integral term in a PID controller, so this term will not dominate, when the errors are large.

arithmetic logic unit (ALU) Component of the processor that performs arithmetic and logic operations.

arm Activate so that interrupts are requested. Trigger flags that can request interrupts will have a corresponding arm bit to allow or disallow that flag to request interrupts. Contrast to enable.

armature The moving structure in a relay, the part that moves when the relay is activated. Contrast to frame.

ASCII American Standard Code for Information Interchange, a code for representing characters, symbols, and synchronization messages as 7 bit, 8-bit or 16-bit binary values.

assembler System software that converts an assembly language program (human readable format) into object code (machine readable format).

assembly directive Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as pseudo-op.

assembly listing Information generated by the assembler in human readable format, typically showing the object code, the original source code, assembly errors, and the symbol table.

asynchronous bus A communication protocol without a central clock where is the data is transferred using two or three control lines implementing a handshaked interaction between the memory and the computer.

asynchronous protocol A protocol where the two devices have separate and distinct clocks

atomic Software execution that cannot be divided or interrupted. Once started, an atomic operation will run to its completion without interruption. Most assembly language instructions are atomic. All instructions on the Cortex-Mprocessor are atomic except store and load multiple, **STM LDM** .

autoinitialization The process of automatically reloading the address registers and block size counters at the end of a previous block transfer, so that DMA transfer can occur indefinitely without software interaction.

availability The portion of the total time that the system is working. MTBF is the mean time between failures, MTTR is the mean time to repair, and availability is MTBF/(MTBF+MTTR).

bandwidth In communication systems, the information transfer rate, the amount of data transferred per second. Same as throughput. In analog circuits, the frequency at which the gain drops to 0.707 of the normal value. For a low pass system, the frequency response ranges from 0 to a maximum value. For a high pass system, the frequency response ranges from a minimum value to infinity. For a bandpass system, the frequency response ranges from a minimum to a maximum value. Compare to frequency response.

bandwidth coupling Module A is connected to Module B, because data flows from A to B.

bang-bang A control system where the actuator has only two states, and the system “bangs” all the way in one direction or “bangs” all the way in the other, same as binary controller.

bank-switched memory A memory module with two banks that interfaces to two separate address/data buses. At any given time one memory bank is attached to one address/data bus the other bank is attached to the other bus, but this attachment can be switched.

basis Subset from which linear combinations can be used to reconstruct the entire set.

baud rate In general, the baud rate is the total number of bits (information, overhead, and idle) per time that are transmitted. In a modem application it is the total number of sounds per time are transmitted

bi-directional Digital signals that can be either input or output.

biendian The ability to process numbers in both big and little-endian formats.

big endian Mechanism for storing multiple byte numbers such that the most significant byte exists first (in the smallest memory address). See also little endian.

binary A system that has two states, on and off.

binary controller Same as bang-bang.

binary recursion A recursive technique that makes two calls to itself during the execution

of the function. See also recursion, linear recursion, and tail recursion.
binary semaphore A semaphore that can have two values. The value=1 means OK and the value=0 means busy. Compare to counting semaphore.
bipolar transistor Either a NPN or PNP transistor.
bipolar stepper motor A stepper motor where the current flows in both directions (in/out) along the interface wires; a stepper with four interface wires. Contrast to unipolar stepper motor.
bit Basic unit of digital information taking on the value of either 0 or 1.
bit rate The information transfer rate, given in bits per second. Same as bandwidth and throughput.
bit time The basic unit of time used in serial communication. With serial channel bit time is 1/baud rate.
blind-cycle A software/hardware synchronization method where the software waits a specified amount of time for the hardware operation to complete. The software has no direct information (blind) about the status of the hardware.
block correction code (BCC) A code (e.g., horizontal parity) attached to the end of a message used to detect and correct transmission errors.
blocked thread A thread that is not scheduled for running because it is waiting on an external event.
blocking semaphore A semaphore where the threads will block (so other threads can perform useful functions) when they execute wait on a busy semaphore. Contrast to spinlock semaphore.
Bluetooth A low-power, wireless personal area network that allows pairing, which is two devices communicating with each other.
Board Support Package (BSP) A set of software routines that abstract the I/O hardware such that the same high-level code can run on multiple computers.
borrow During subtraction, if the difference is too small, then we use a borrow to pass the excess information into the next higher place. For example, in decimal subtraction 36-27 requires a borrow from the ones to tens place because 6-7 is too small to fit into the 0 to 9 range of decimal numbers.
bounded waiting The condition where once a thread begins to wait on a resource, there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed.
break-before-make In a double-throw relay or double-throw switch, there is one common contact and two separate contacts. Break-before-make means as the common contact moves from one of separate contacts to another, it will break off (finish bouncing and no longer touch) the first contact before it makes (begins to bounce and starts to touch) the other contact. A *form C* relay has a *break-before-make* operation.
break or trap A break or a trap is a debugging instrument that halts the processor. With a resident debugger, the break is created by replacing specific op code with a software interrupt instruction. When encountered it will stop your program and jump into the debugger. Therefore, a break halts the software. The condition of being in this state is also referred to as a break.
breakdown A transducer that stops functioning when its input goes above a maximum value

or below a minimum value. Contrast to dead zone.
breakpoint The place where a break is inserted, the time when a break is encountered, or the time period when a break is active.
brushed DC motor A motor where the current reversals are produced with brushes between the stator and rotor. They are less expensive than brushless DC motors.
brushless DC motor (BLDC) A motor where the current reversals are produced with shaft sensors and an electronic controller. They are faster and more reliable than brushed DC motors.
buffered I/O A FIFO queue is placed in between the hardware and software in an attempt to increase bandwidth by allowing both hardware and software to run in parallel.
burn The process of programming a ROM, PROM or EEPROM.
burst DMA An I/O synchronization scheme that transfers an entire block of data all at once directly from an input device into memory, or directly from memory to an output device.
bus A set of digital signals that connect the CPU, memory and I/O devices, consisting of address signals, data signals and control signals. See also address bus, control bus and data bus.
bus bandwidth The number of bytes transferred per second between the processor and memory.
bus interface unit (BIU) Component of the processor that reads and writes data from the bus. The BIU drives the address and control buses.
busy-wait synchronization A software/hardware synchronization method where the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as gadfly.
byte Digital information containing eight bits.
carrier frequency the average or midvalue sound frequency in the modem.
cathode The negative side of a diode. Current exits the cathode side of a diode. Contrast to anode.
causal The property where the output depends on the present and past inputs, but not on any future inputs.
ceiling Establishing an upper bound on the result of an operation. See also floor.
certification A process where a governing body (e.g., FDA, NASA, FCC, DOD etc.) gives approval for the use of the device. It usually involves demonstrating the device meets or exceeds safety and performance criteria.
channel The hardware that allows communication to occur.
characteristic A Bluetooth functionalities that allows data to be exchanged.
checksum The simple sum of the data, usually in finite precision (e.g., 8, 16, 24 bits).
closed-loop control system A control system that includes sensors to measure the current state variables. These inputs are used to drive the system to the desired state.
CMOS A digital logic system called complementary metal oxide semiconductor. It has properties of low power and small size. Its power is a function of the number of transitions per second. Its speed is often limited by capacitive loading.
cohesion A cohesive module is one such that all parts of the module are related to each other to satisfy a common objective.

common mode For a system with differential inputs, the common mode properties are defined as signals applied to both inputs simultaneously. Contrast to differential mode.

common mode input impedance Common mode input voltage divided by common mode input current.

common mode rejection ratio For a differential amplifier, CMRR is the ratio of the common mode gain divided by the differential mode gain. A perfect CMRR would be zero.

compiler System software that converts a high-level language program (human readable format) into object code (machine readable format).

complex instruction set computer (CISC) A computer with many instructions, instructions that have varying lengths, instructions that execute in varying times, many instructions can access memory, one instruction may both read and write memory, fewer and more specialized registers, and many different types of addressing modes. Contrast to RISC.

compression ratio The ratio of the number of original bytes to the number of compressed bytes.

concurrent programming A software system that supports two tasks to be active at the same time. A computer with interrupts implements concurrent programming. Compare to distributed and parallel.

condition code register (CCR) Register in the processor that contains the status of the previous ALU operation, as well as some operating mode flags such as the interrupt enable bit.

control coupling Module A is connected to Module B, because actions in A affect the control path in B.

control unit (CU) Component of the processor that determines the sequence of operations.

cooperative multi-tasking A scheduler that cannot suspend execution of a thread without the thread's permission. The thread must cooperate and suspend itself. Same as nonpreemptive scheduler.

counting semaphore A semaphore that can have any signed integer value. The value>0 means OK and the value≤0 means busy. Compare to binary semaphore.

CPU bound A situation where the input or output device is faster than the software. In other words, it takes less time for the I/O device to process data, than for the software to process data. Contrast to I/O bound.

CPU cycle A memory bus cycle where the address and R/W are controlled by the processor. On microcontrollers without DMA, all cycles are CPU cycles. Contrast to DMA cycle.

crisp input An input parameter to the fuzzy logic system, usually with units like cm, cm/sec, ¤ C etc.

crisp output An output parameter from the fuzzy logic system, usually with units like dynes, watts etc.

critical section Locations within a software module, which if an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash, etc.) Same as vulnerable window.

cross-assembler An assembler that runs on one computer but creates object code for a different computer.

cross-compiler A compiler that runs on one computer but creates object code for a different

computer.

cycle steal DMA An I/O synchronization scheme that transfers data one item at a time directly from an input device into memory, or directly from memory to an output device. Same as single cycle DMA.

cycle stretch The action where some memory cycles are longer allowing time for communication with slower memories, sometimes the memory itself requests the additional time and sometimes the computer has a preprogrammed cycle stretch for certain memory addresses

DAC Digital to analog converter, an electronic device that converts digital signals (i.e., integers) to analog form (e.g., voltage).

data acquisition system A system that collects information, same as instrument.

data bus A set of digital signals that connect the CPU, memory and I/O devices, specifying the value that is being read or written for each bus cycle. See also address bus and control bus.

data communication equipment (DCE) A modem or printer connected a serial communication network.

data terminal equipment (DTE) A computer or a terminal connected a serial communication network.

deadline The time when a task should complete. Compare to slack time.

dead zone A condition of a transducer when a large change in the input causes little or no change in the output. Contrast to breakdown.

deadlock A scenario that occurs when two or more threads are all blocked each waiting for the other with no hope of recovery.

defuzzification Conversion from the fuzzy logic output variables to the crisp outputs.

desk checking or dry run We perform a desk check (or dry run) by determining in advance, either by analytical algorithm or explicit calculations, the expected outputs of strategic intermediate stages and final results for a set of typical inputs. We then run our program can compare the actual outputs with this template of expected results.

device driver A collection of software routines that perform I/O functions.

differential mode For a system with differential inputs, the differential mode properties are defined as signals applied as a difference between the two inputs. Contrast to common mode.

differential mode input impedance Differential mode input voltage divided by differential mode input current.

digital signal processing Processing of data with digital hardware or software after the signal has been sampled by the ADC, e.g., filters, detection and compression/decompression.

direct memory access (DMA) the ability to transfer data between two modules on the bus, this transfer is usually initiated by the hardware (device needs service) and the software configures the communication, but the data is transferred without explicit software action for each piece of data

direction register Specifies whether a bi-directional I/O pin is an input or an output. We set a direction register bit to 0 (or 1) to specify the corresponding I/O pin to be input (or output.)

disarm Deactivate so that interrupts are not requested, performed by clearing the arm bit.
Discrete Fourier Transform (DFT) A technique to convert data in the time domain to data in the frequency domain. N data points are sampled at f_s. The resulting frequency resolution is f_s /N.
distributed processing A system implemented across separate computers connected with I/O or a network, so that two or more programs are executed at the same time. Compare to concurrent and parallel.
DMA Direct Memory Access is a software/hardware synchronization method where the hardware itself causes a data transfer between the I/O device and memory at the appropriate time when data needs to be transferred. The software usually can perform other work while waiting for the hardware. No software action is required for each individual byte.
DMA cycle A memory bus cycle where the address and R/W are controlled by the DMA controller. Contrast to CPU cycle.
double byte Two bytes containing 16 bits. Same as halfword.
double-pole relay Two separate and complete relays, which are activated together. Contrast to single pole.
double-pole switch Two separate and complete switches. The two switches are electrically separate, but mechanically connected. Such that both switches are activated together. Contrast to single pole.
double-throw relay A relay with three contact connections, one common and two throws. The common will be connected to exactly of one the two throws (see single-throw).
double-throw switch A switch with three contact connections. The center contact will be connected exactly one of the other two contacts. Contrast with single-throw.
double word Two words containing 64 bits.
download The process of transferring object code from the host (e.g., the PC) to the target microcontroller.
drop-out An error that occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. E.g., I=100*(N/51) can only result in the values 0, 100, or 200, whereas I=(100*N)/51 properly calculates the desired result.
dual address DMA Direct memory access that requires two bus cycles to transfer data from an input device into memory, or from memory to an output device.
dual port memory A memory module that interfaces to two separate address/data buses, and allows both systems read/write access the data.
duty cycle For a periodic digital wave, it is the percentage of time the signal is high. When an LED display is scanned, it is the percentage of time each LED is active. A motor interfaced using pulse-width-modulation allows the computer to control delivered power by adjusting the duty cycle.
dynamic allocation Data structures like the TCB that are created at runtime by calling malloc() and exist until the software releases the memory block back to the heap by calling free(). See static allocation.
dynamic RAM Volatile read/write storage built from a capacitor and a single transistor having a low cost, but requiring refresh. Contrast with static RAM.

EEPROM Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram. EEPROM can be erased and reprogrammed multiple times. Also see Flash EEPROM.

embedded computer system A system that performs a specific dedicated operation where the computer is hidden or embedded inside the machine.

emulator An in-circuit emulator is an expensive debugging hardware tool that mimics the processor pin outs. To debug with an emulator, you would remove the processor chip and attach the emulator cable into the processor socket. The emulator would sense the processor input signals and recreate the processor outputs signals on the socket as if an actual processor were actually there, running at full speed. Inside the emulator you have internal read/write access to the registers and processor state. Most emulators allow you to visualize/record strategic information in real-time without halting the program execution. You can also remove ROM chips and insert the connector of a ROM-emulator. This type of emulator is less expensive, and it allows you to debug ROM-based software systems.

EPROM Same as PROM. Electrically programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light.

erase The process of clearing the information in a PROM or EEPROM, using electricity or UV light. The information bits are usually all set to logic 1.

EVB Evaluation Board, a board-level product used to develop microcontroller systems. Same as LaunchPad.

even parity A communication protocol where the number of ones in the data plus a parity bit is an even number. Contrast with odd parity.

event thread A thread that is executed or triggered in response to an event. They are similar to ISR, but scheduled by the OS. Typcially, the event is a change in hardware status, such as input ready, output idle, or periodically. The trigger could also be a software event. Event threads cannot sleep, block, or be killed. Once they respond to the event, they simply return. Compare to main thread.

external fragmentation A condition when the largest file or memory block that can be allocated is less than the total amount of free space on the disk or memory.

fan out The number of inputs that a single output can drive if the devices are all in the same logic family.

Fast Fourier Transform (FFT) A fast technique to convert data in the time domain to data in the frequency domain. N data points are sampled at f_s. The resulting frequency resolution is f_s /N. Mathematically, the FFT is the same as the DFT, just faster.

FET Field effect transistor, also JFET.

filter In the debugging context, a filter is a Boolean function or conditional test used to make run-time decisions. For example, if we print information only if two variables x,y are equal, then the conditional (x==y) is a filter. Filters can involve hardware status as well.

finite impulse response filter (FIR) A digital filter where the output is a function of a finite number of current and past data samples, but not a function of previous filter outputs.

Finite State Machine (FSM) An abstract design method to build a machine with inputs and outputs. The machine can be in one of a finite number of states. Which state the system is in represents memory of previous inputs. The output and next state are a function of the input. There may be time delays as well.

firm real-time A system that expects all critical tasks to complete on time. Once a deadline as passed, there is no value to completing the task. However, the consequence of missed deadlines is real but the overall system operates with reduced quality. Streaming audio and video are typical examples. Compare to hard real-time and soft real-time.

fixed-point A technique where calculations involving nonintegers are performed using a sequence of integer operations. E.g., 0.123*x is performed in decimal fixed-point as (123*x)/1000 or in binary fixed-point as (126*x)>>10.

flash EEPROM Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram. Flash EEPROMs are typically larger than regular EEPROM.

floating A logic state where the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the floating state. Same as HiZ.

floor Establishing a lower bound on the result of an operation. See also ceiling.

fork The dynamic action of creating a new thread or process at run time. See also join.

frame A complete and distinct packet of bits occurring in a serial communication channel.

frame The fixed structure in a relay or transducer. Contrast to armature.

framing error An error when the receiver expects a stop bit (1) and the input is 0.

frequency response The frequency at which the gain drops to 0.707 of the normal value. For a low pass system, the frequency response ranges from 0 to a maximum value. For a high pass system, the frequency response ranges from a minimum value to infinity. For a bandpass system, the frequency response ranges from a minimum to a maximum value. Same as bandwidth.

frequency shift key (**FSK**) A modem that modulates the digital signals into frequency encoded sine waves.

friendly Friendly software modifies just the bits that need to be modified, leaving the other bits unchanged, making to easier to combine modules.

full duplex channel Hardware that allows bits (information, error checking, synchronization or overhead) to transfer simultaneously in both directions. Contrast with simplex and half duplex channels.

full duplex communication A system that allows information (data, characters) to transfer simultaneously in both directions.

functional debugging The process of detecting, locating, or correcting functional and logical errors in a program, typically not involving time. The process of instrumenting a program for such purposes is called functional debugging or often simply debugging.

fuzzification Conversion from the crisp inputs to the fuzzy logic input variables.

fuzzy logic Boolean logic (true/false) that can take on a range of values from true (255) to false (0). Fuzzy logic **and** is calculated as the minimum. Fuzzy logic **or** is the maximum.

gadfly A software/hardware synchronization method where the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as busy wait.

gauge factor The sensitivity of a strain gauge transducer, i.e., slope of the resistance versus displacement response.

gibibyte (GiB) 2^{30} or 1,073,741,824 bytes. Compare to **gigabyte**, which is 1,000,000,000 bytes.

half duplex channel Hardware that allows bits (information, error checking, synchronization or overhead) to transfer in both directions, but in only one direction at a

time. Contrast with simplex and full duplex channels.

half duplex communication A system that allows information to transfer in both directions, but in only one direction at a time.

halfword Two bytes containing 16 bits. Same as double byte.

handshake A software/hardware synchronization method where control and status signals go both directions between the transmitter and receiver. The communication is interlocked meaning each device will wait for the other.

hard real-time A system that can guarantee that a process will complete a critical task within a certain specified range. In data acquisition systems, hard real-time means there is an upper bound on the latency between when a sample is supposed to be taken (every 1/fs) and when the ADC is actually started. Hard real-time also implies that no ADC samples are missed. Compare to hard real-time and firm real-time.

heartbeat A debugging monitor, such as a flashing LED, we add for the purpose of seeing if our program is running.

hexadecimal A number system that uses base 16.

HiZ A logic state where the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the floating state. Same as floating.

hold time When latching data into a device with a rising or falling edge of a clock, the hold time is the time after the active edge of the clock that the data must continue to be valid. See setup time.

hook An indirect function call added to a software system that allows the user to attach their programs to run at strategic times. These attachments are created at run time and do not require recompiling the entire system.

horizontal parity A parity calculated across the entire message on a bit by bit basis, e.g., the horizontal parity bit 0 is the parity calculated on all the bit 0's of the entire message, can be even or odd parity

hysteresis A condition when the output of a system depends not only on the input, but also on the previous outputs, e.g., a transducer that follows a different response curve when the input is increasing than when the input is decreasing.

I/O bound A situation where the input or output device is slower than the software. In other words, it takes longer for the I/O device to process data than for the software to process data. Contrast to CPU bound.

I/O device Hardware and software components capable of bringing information from the external environment into the computer (input device), or sending data out from the computer to the external environment (output device.)

I/O port A hardware device that connects the internal software with external hardware.

I_{IH} Input current when the signal is high.

I_{IL} Input current when the signal is low.

immediate An addressing mode where the operand is a fixed data or address value.

impedance loading A condition when the input of stage n+1 of an analog system affects the output of stage n, because the input impedance of stage n+1 is too small and the output impedance of stage n is too large.

impedance The ratio of the effort (voltage, force, pressure) divided by the flow (current, velocity, flow).

incremental control system A control system where the actuator has many possible states, and the system increments or decrements the actuator value depending on either in error is positive or negative.
indexed An addressing mode where the data or address value for the instruction is located in memory pointed to by an index register.
infinite impulse response filter (IIR) A digital filter where the output is a function of an infinite number of past data samples, usually by making the filter output a function of previous filter outputs.
input bias current Difference between currents of the two op amp inputs.
input capture A mechanism to set a flag and capture the current time (TCNT value) on the rising, falling or rising&falling edge of an external signal. The input capture event can also request an interrupt.
input impedance Input voltage divided by input current.
instruction register (IR) Register in the control unit that contains the op code for the current instruction.
instrument An instrument is the code injected into a program for debugging or profiling. This code is usually extraneous to the normal function of a program and may be temporary or permanent. Instruments injected during interactive sessions are considered to be temporary because these instruments can be removed simply by terminating a session. Instruments injected in source code are considered to be permanent because removal requires editing and recompiling the source. An example of a temporary instrument occurs when the debugger replaces a regular op code with a breakpoint instruction. This temporary instrument can be removed dynamically by restoring the original op code. A print statement added to your source code is an example of a permanent instrument, because removal requires editing and recompiling.
instrument An embedded system that collects information, same as data acquisition system.
instrumentation The debugging process of injecting or inserting an instrument.
instrumentation amp A differential amplifier analog circuit, which can have large gain, large input impedance, small output impedance, and a good common mode rejection ration.
internal fragmentation Storage that is allocated for the convenience of the operating system but contains no information. This space is wasted.
interrupt A software/hardware synchronization method where the hardware causes a special software program (interrupt handler) to execute when its operation to complete. The software usually can perform other work while waiting for the hardware.
interrupt flag A status bit that is set by the timer hardware to signify an external event has occurred.
interrupt mask A control bit that, if programmed to 1, will cause an interrupt request when the associated flag is set. Same as **arm**.
interrupt service routine (ISR) Program that runs as a result of an interrupt.
interrupt vector 32-bit values in ROM specifying where the software should execute after an interrupt request. There is a unique interrupt vector for each type of interrupt including reset.
intrusive The debugger itself affects the program being tested. See nonintrusive.
Inverse Discrete Fourier Transform (IDFT) A technique to convert data in the frequency

domain to data in the time domain. If there are N data points and the sampling rate is f_s, the resulting frequency resolution will be f_s /N.

invocation coupling Module A is connected to Module B, because A calls B.

I/O mapped I/O A configuration where the I/O devices are interfaced to the computer in a manner different than the way memories are connected, from an interfacing perspective I/O devices and memory modules have separate bus signals, from a programmer's point of view the I/O devices have their own I/O address map separate from the memory map, and I/O device access requires the use of special I/O instructions

I_{OH} Output current when the signal is high. This is the maximum current that has a voltage above V_{OH}.

I_{OL} Output current when the signal is low. This is the maximum current that has a voltage below V_{OL}.

jerk The change in acceleration; the derivative of the acceleration.

Johnson noise A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as thermal noise and white noise.

join The dynamic action of combining multiple threads or processes at run time. The fork operation will take one thread/process and create multiple children. The join operation will take the multiple threads/processes and convert it back to one. See also fork.

kibibyte (KiB) 2^{10} or 1024 bytes. Compare to **kilobyte**, which is 1000 bytes.

latch As a noun, it means a register. As a verb, it means to store data into the register.

latched input port An input port where the signals are latched (saved) on an edge of an associated strobe signal.

latency In this book latency usually refers to the response time of the computer to external events. For example, the time between new input becoming available and the time the input is read by the computer. For example, the time between an output device becoming idle and the time the input is the computer writes new data to it. There can also be latency for an I/O device, which is the response time of the external I/O device hardware to a software command.

LCD Liquid Crystal Display, where the computer controls the reflectance or transmittance of the liquid crystal, characterized by its flexible display patterns, low power, and slow speed.

LED Light Emitting Diode, where the computer controls the electrical power to the diode, characterized by its simple display patterns, medium power, and high speed.

light-weight process Same as a thread.

linear filter A filter where the output is a linear combination of its inputs.

linear recursion A recursive technique that makes only one call to itself during the execution of the function. Linear recursive functions are easier to implement iteratively. We draw the execution pattern as a straight or linear path. See also recursion, binary recursion, and tail recursion.

little endian Mechanism for storing multiple byte numbers such that the least significant byte exists first (in the smallest memory address). Contrast with big endian.

loader System software that places the object code into the microcontroller's memory. If the object code is stored in EPROM, the loader is also called a EPROM programmer.

Local Area Network (LAN) A connection between computers confined to a small space, such as a room or a building.

logic analyzer A hardware debugging tool that allows you to visualize many digital logic signals versus time. Real logic analyzers have at least 32 channels and can have up to 200 channels, with sophisticated techniques for triggering, saving and analyzing the real-time data. In **TExaS**, logic analyzers have only 8 channels and simply plot digital signals versus time.

LSB The least significant bit in a number system is the bit with the smallest significance, usually the right-most bit. With signed or unsigned integers, the significance of the LSB is 1.

maintenance Process of verifying, changing, correcting, enhancing, and extending a system.

make before break in a double-throw relay or double-throw switch, there is one common contact and two separate contacts. Make before break means as the common contact moves from one of separate contacts to another, it will make (finishing bouncing) the second contact before it breaks off (start bouncing) the first contact. A *form D* relay has a *make before break* operation.

mailbox A formal communication structure, similar to a FIFO queue, where the source task puts data into the mailbox and the sink task gets data from the mailbox. The mailbox can hold at most one piece of data at a time, and has two states: mailbox has valid data or mailbox is empty.

main thread A thread runs like a main program. If it is static, it is created on startup and runs forever by the scheduler. It could also be dynamic, created at run time. Main threads can sleep, block, and be killed. Compare to event thread.

mark A digital value of true or logic 1. Contrast with space.

mask As a verb, mask is the operation that selects certain bits out of many bits, using the logical and operation. The bits that are not being selected will be cleared to zero. When used as a noun, mask refers to the specific bits that are being selected.

Mealy FSM A FSM where the both the output and next state are a function of the input and state

measurand A signal measured by a data acquisition system.

mebibyte (MiB) 2^{20} or 1,048,576 bytes. Compare to **megabyte**, which is 1,000,000 bytes.

membership sets Fuzzy logic variables that can take on a range of values from true (255) to false (0).

memory A computer component capable of storing and recalling information.

memory-mapped I/O A configuration where the I/O devices are interfaced to the computer in a manner identical to the way memories are connected, from an interfacing perspective I/O devices and memory modules shares the same bus signals, from a programmer's point of view the I/O devices exist as locations in the memory map, and I/O device access can be performed using any of the memory access instructions.

microcomputer A small electronic device capable of performing input/output functions containing a microprocessor, memory, and I/O devices, where small means you can carry it.

microcontroller A single chip microcomputer like the TI MSP430, Freescale 9S12, Intel 8051, PIC16, or the Texas Instruments TM4C123.

mnemonicThe symbolic name of an operation code, like **mov str push** .

modem An electronic device that MOdulates and DEModulates a communication signal.

Used in serial communication across telephone lines.

monitor or **debugger window** A monitor is a debugger feature that allows us to passively view strategic software parameters during the real-time execution of our program. An effective monitor is one that has minimal effect on the performance of the system. When debugging software on a windows-based machine, we can often set up a debugger window that displays the current value of certain software variables.

Moore FSM A FSM where the both the output is only a function of the state and the next state is a function of the input and state

MOSFET Metal oxide semiconductor field effect transistor.

MSB The most significant bit in a number system is the bit with the greatest significance, usually the left-most bit. If the number system is signed, then the MSB signifies positive (0) or negative (1).

multiple access circular queue MACQ A data structure used in data acquisition systems to hold the current sample and a finite number of previous samples.

multithreaded A system with multiple threads (e.g., main program and interrupt service routines) that cooperate towards a common overall goal.

mutual exclusion or **mutex** Thread synchronization where at most one thread at a time is allowed to enter.

negative feedback An analog system with negative gain feedback paths. These systems are often stable.

negative logic A signal where the true value has a lower voltage than the false value, in digital logic true is 0 and false is 1, in TTL logic true is less than 0.7 volts and false is greater than 2 volts, in RS232 protocol true is -12 volts and false is +12 volts. Contrast with positive logic.

nibble 4 binary bits or 1 hexadecimal digit.

nonatomic Software execution that can be divided or interrupted. Most lines of C code require multiple assembly language instructions to execute, therefore an interrupt may occur in the middle of a line of C code. The instructionsstore and load multiple, **STM LDM** , are nonatomic.

nonintrusive A characteristic when the presence of the collection of information itself does not affect the parameters being measured. Nonintrusiveness is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by an instrument. For example, a print statement added to your source code and single-stepping are very intrusive because they significantly affect the real-time interaction of the hardware and software. When a program interacts with real-time events, the performance is significantly altered. On the other hand, an instrument that toggles an LED on and off (requiring less than a 1 µs to execute) is much less intrusive. A logic analyzer that passively monitors the address and data by is completely nonintrusive. An in-circuit emulator is also non-intrusive because the software input/output relationships will be the same with and without the debugging tool.

nonlinear filter A filter where the output is not a linear combination of its inputs. E.g., median, minimum, maximum are examples of nonlinear filters. Contrast to linear filter.

nonpreemptive scheduler A scheduler that cannot suspend execution of a thread without

the thread's permission. The thread must cooperate and suspend itself. Same as cooperative multi-tasking.

nonreentrant A software module which once started by one thread, should not be interrupted and executed by a second thread. A nonreentrant modules usually involve nonatomic accesses to global variables or I/O ports: read modify write, write followed by read, or a multistep write.

nonvolatile A condition where information is not lost when power is removed. When power is restored, then the information is in the state that occurred when the power was removed.

Nyquist Theorem If a input signal is captured by an ADC at the regular rate of fs samples/sec, then the digital sequence can accurately represent the 0 to ½ fs frequency components of the original signal.

object code Programs in machine readable format created by the compiler or assembler.

odd parity A communication protocol where the number of ones in the data plus a parity bit is an odd number. Contrast with even parity.

op amp An integrated analog component with two inputs, (V_2,V_1) and an output (V_{out}), where $V_{out}=K\bullet(V_2-V_1)$. The amp has a very large gain, K. Same as operational amplifier.

op code, opcode, or **operation code** A specific instruction executed by the computer. The op code along with the operand completely specifies the function to be performed. In assembly language programming, the op code is represented by its mnemonic, like MOV. During execution, the op code is stored as a machine code loaded in memory.

open collector A digital logic output that has two states low and HiZ. Same as open drain and wire-or-mode.

open drain A digital logic output that has two states low and HiZ. Same as open collector and wire-or-mode.

open loop control system A control system that does not include sensors to measure the current state variables. An analog system with no feedback paths.

operand The second part of an instruction that specifies either the data or the address for that instruction. An assembly instruction typically has an op code (e.g., MOV) and an operand (e.g., R0,#55). Instructions that use inherent addressing mode have no operand field.

operating system System software for managing computer resources and facilitating common functions like input/output, memory management, and file system.

originate modem the device that places the telephone call.

oscilloscope A hardware debugging tool that allows you to visualize one or two analog signals versus time.

output compare A mechanism to cause a flag to be set and an output pin to change when the timer matches a preset value. The output compare event can also request an interrupt.

output impedance Open circuit output voltage divided by short circuit output current.

overflow An error that occurs when the result of a calculation exceeds the range of the number system. For example, with 8-bit unsigned integers, 200+57 will yield the incorrect result of 1.

overrun error An error that occurs when the receiver gets a new frame but the receive FIFO and shift register already have information.

paged memory A memory organization where logical addresses (used by software) have multiple and distinct components or fields. The number of bits in the least significant field defines the page size. The physical memory is usually continuous having sequential addresses. There is a dynamic address translation (logical to physical).
parallel port A port where all signals are available simultaneously. In this book the ports are 8 bits wide.
parallel processing A system that supports two or more programs being executed at the same time. A computer with multiple cores implements parallel programming. Compare to concurrent and distributed.
partially asynchronous bus a communication protocol that has a central clock but the memory module can dynamically extend the length of a bus cycle (cycle stretch) if it needs more time
path expression A software technique to guarantee subfunctions within a module are executed in a proper sequence. For example, it forces the user to initialize I/O device before attempting to perform I/O.
PC-relative addressing An addressing mode where the effective address is calculated by its position relative to the current value of the program counter.
performance debugging or profiling The process of acquiring or modifying timing characteristics and execution patterns of a program and the process of instrumenting a program for such purposes is called performance debugging or profiling.
periodic polling A software/hardware synchronization method that is a combination of interrupts and busy wait. An interrupt occurs at a regular rate (periodic) independent of the hardware status. The interrupt handler checks the hardware device (polls) to determine if its operation is complete. The software usually can perform other work while waiting for the hardware.
Personal Area Network (PAN) A connection between computers controlled by a single person or all working toward for a well-defined single task.
phase shift key (PSK) a protocol that encodes the information as phase changes between the sounds.
photosensor A transducer that converts reflected or transmitted light into electric current.
physical plant The physical device being controlled.
PID controller A control system where the actuator output depends on a linear combination of the current error (P), the integral of the error (I) and the derivative of the error (D).
pink noise A fundamental noise in resistive devices arising from fluctuating conductivity. Same as 1/f noise.
pole A place in the frequency domain where the filter gain is infinite.
polling A software function to look and see which of the potential sources requested the interrupt.
port External pins through which the microcontroller can perform input/output. Same as I/O port.
positive feedback An analog system with positive gain feedback paths. These systems will saturate.
positive logic a signal where the true value has a higher voltage than the false value, in digital logic true is 1 and false is 0, in TTL logic true is greater than 2 volts and false is

less than 0.7 volts, in RS232 protocol true is +12 volts and false is -12 volts. Contrast with negative logic.

potentiometer A transducer that converts position into electric resistance.

precision A term specifying the degrees of freedom from random errors. For an input signal, it is the number of distinguishable input signals that can be reliably detected by the measurement. For an output signal, it is the number of different output parameters that can be produced by the system. For a number system, precision is the number of distinct or different values of a number system in units of "alternatives". The precision of a number system is also the number of binary digits required to represent all its numbers in units of "bits".

preemptive scheduler A scheduler that has the power to suspend execution of a thread without the thread's permission.

priority When two requests for service are made simultaneously, priority determines which order to process them.

private Can be accessed only by software modules in that local group.

private variable A global variable that is used by a single thread, and not shared with other threads.

process The execution of software that does not necessarily cooperate with other processes.

producer-consumer A multithreaded system where the producers generate new data, and the consumers process or output the data.

profile A collection of services implemented by Bluetooth.

profiling See performance debugging.

program counter (PC) A register in the processor that points to the memory containing the instruction to execute next.

PROM Same as EPROM. Programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light.

promotion Increasing the precision of a number for convenience or to avoid overflow errors during calculations.

pseudo interrupt vector A secondary place for the interrupt vectors for the convenience of the debugger, because the debugger cannot or does not want the user to modify the real interrupt vectors. They provide flexibility for debugging but incur a run time delay during execution.

pseudo op Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as assembly directive.

pseudo-code A shorthand for describing a software algorithm. The exact format is not defined, but many programmers use their favorite high-level language syntax (like C) without paying rigorous attention to the punctuation.

public Can be accessed by any software module.

public variable A global variable that is shared by multiple programs or threads.

pulse width modulation A technique to deliver a variable signal (voltage, power, energy) using an on/off signal with a variable percentage of time the signal is on (duty cycle). Same as **variable duty cycle**.

Q The Q of a bandpass filter (passes f_{min} to f_{max}) is the center pass frequency (f_o= (f_{max}+f_{min})/2) divided by the width of the pass region, Q=f_o/(f_{max}-f_{min}). The Q of a bandreject filter (rejects f_{min} to f_{max}) is the center reject frequency (f_o=(f_{max}+f_{min})/2) divided by the width of the reject region, Q=f_o/(f_{max}-f_{min}).

quadrature amplitude modem (QAM) a protocol that used both the phase and amplitude to encode up to 6 bits onto each baud.

qualitative DAS A DAS that collects information not in the form of numerical values, but rather in the form of the qualitative senses, e.g., sight, hearing, smell, taste and touch. A qualitative DAS may also detect the presence or absence of conditions.

quantitative DAS A DAS that collects information in the form of numerical values.

RAM Random Access Memory, a type of memory where is the information can be stored and retrieved easily and quickly. Since it is volatile the information is lost when power is removed.

range Includes both the smallest possible and the largest possible signal (input or output). The difference between the largest and smallest input that can be measured by the instrument. The units are in the units of the measurand. When precision is in alternatives, range=precision•resolution. Same as span

read cycle data flows from the memory or input device to the processor, the address bus specifies the memory or input device location and the data bus contains the information at that address

read data available The time interval (start,end) during which the data will be valid during a read cycle, determined by the memory module

real-time A characteristic of a system that can guarantee an upper bound (worst case) on latency.

real-time system A system where time-critical operations occur when needed.

recursion A programming technique where a function calls itself.

reduced instruction set computer (RISC) A computer with a few instructions, instructions with fixed lengths, instructions that execute in 1 or 2 bus cycles, only load and store can access memory, no one instruction can both read and write memory, many identical general purpose registers, and a limited number of addressing modes. Contrast to CISC.

reentrant A software module that can be started by one thread, interrupted and executed by a second thread. A reentrant module allows both threads to properly execute the desired function. Contrast with non-reentrant.

registers High-speed memory located in the processor. The registers in the ARM® Cortex™-M include R0 through R15.

relay A mechanical switch that can be turned on and off by the computer.

reliability The ability of a system to operate within specified parameters for a stated period of time. Given in terms of mean time between failures (MTBF).

reproducibility (or **repeatability**) A parameter specifying how consistent over time the measurement is when the input remains fixed.

requirements document A formal description of what the system will do in a very complete way, but not including how it will be done. It should be unambiguous, complete, verifiable, and modifiable.

reset vector The 32-bit value at memory locations 0x0000.0004 specifying where the

software should start after power is turned on or after a hardware reset.

resolution For an input signal, it is the smallest change in the input parameter that can be reliably detected by the measurement. For an output signal, it is the smallest change in the output parameter that can be produced by the system, range equals precision times resolution. The units are in the units of the measurand. When precision is in alternatives, range=precision•resolution.

response time Similar to latency, it is the delay between when the time an event occurs and the time the software responds to the event.

ritual Software, usually executed once at the beginning of the program, that defines the operational modes of the I/O ports.

ROM Read Only Memory, a type of memory where is the information is programmed into the device once, but can be accessed quickly. It is low cost, must be purchased in high volume and can be programmed only once. See also EPROM, EEPROM, and flash EEPROM.

rotor The part of a motor that rotates.

round robin scheduler A scheduler that runs each active thread equally.

roundoff The error that occurs in a fixed-point or floating-point calculation when the least significant bits of an intermediate calculation are discarded so the result can fit into the finite precision.

sample and hold A circuit used to latch a rapidly changing analog signal, capturing its input value and holding its output constant.

sampling rate The rate at which data is collected in a data acquisition system.

saturation A device that is no longer sensitive to its inputs when its input goes above a maximum value or below a minimum value.

scan or scanpoint Any instrument used to produce a side effect without causing a break (halt) is a scan. Therefore, a scan may be used to gather data passively or to modify functions of a program. Examples include software added to your source code that simply outputs or modifies a global variable without halting. A scanpoint is triggered in a manner similar to a breakpoint but a scanpoint simply records data at that time without halting execution.

scheduler System software that suspends and launches threads.

Schmitt Trigger A digital interface with hysteresis making it less susceptible to noise.

scope A logic analyzer or an oscilloscope, hardware debugging tools that allows you to visualize multiple digital or analog signals versus time.

select signal The output of the address decoder (each module on the bus has a separate address decoder); a Boolean (true/false) signal specifying whether or not the current address of the bus matches the device address

semaphore A system function with two operations (wait and signal) that provide for thread synchronization and resource sharing.

sensitivity The sensitivity of a transducer is the slope of the output versus input response. The sensitivity of a qualitative DAS that detects events is the percentage of actual events that are properly recognized by the system.

serial communication A process where information is transmitted one bit at a time.

serial peripheral interface (SPI) A device to transmit data with synchronous serial

communication protocol. Same as SSI.
serial port An I/O port with which the bits are input or output one at a time.
service A collection of Bluetooth functionalities that taken together solve one coherent system function. Examples include blood pressure monitor and human interface device (mouse, keyboard).
servo A DC motor with built in controller. The microcontroller specifies desired position and the servo adds/subtracts power to move the shaft to that position.
setup time When latching data into a register with a clock, it is the time before an edge the input must be valid. Contrast with hold time.
shot noise A fundamental noise that occurs in devices that count discrete events.
signed two's complement binary A mechanism to represent signed integers where 1 followed by all 0's is the most negative number, all 1's represents the value -1, all 0's represents the value 0, and 0 followed by all 1's is the largest positive number.
sign-magnitude binary A mechanism to represent signed integers where the most significant bit is set if the number is negative, and the remaining bits represent the magnitude as an unsigned binary.
simplex channel Hardware that allows bits (information, error checking, synchronization or overhead) to transfer only in one direction. Contrast with half duplex and full duplex channels.
simplex communication A system that allows information to transfer only in one direction.
simulator A simulator is a software application that simulates or mimics the operation of a processor or computer system. Most simulators recreate only simple I/O ports and often do not effectively duplicate the real-time interactions of the software/hardware interface. On the other hand, they do provide a simple and interactive mechanism to test software. Simulators are especially useful when learning a new language, because they provide more control and access to the simulated machine, than one normally has with real hardware.
single address DMA Direct memory access that requires only one bus cycle to transfer data from an input device into memory, or from memory to an output device.
single cycle DMA An I/O synchronization scheme that transfers data one item at a time directly from an input device into memory, or directly from memory to an output device. Same as cycle steal DMA.
single-pole relay A simple relay with only one copy of the switch mechanism. Contrast with double pole.
single-pole switch A simple switch with only one copy of the switch mechanism. One switch that acts independent from other switches in the system. Contrast with double-pole.
single-throw switch A switch with two contact connections. The two contacts may be connected or disconnected. Contrast with double-throw.
slack time The time-to-deadline minus the how long it will take to complete the task. For example, if slack time is zero, then you could complete the task on time if you devoted 100% of your resources.
slew rate The maximum slope of a signal. If the time-varying signal V(t) is in volts, the slew rate is the maximum dV/dt in volts/s.
soft real-time A system that implements best effort to execute critical tasks on time, typically using a priority scheduler. Once a deadline as passed, the value of completing the

task diminishes over time. Compare to hard real-time and firm real-time.

software interrupt A software interrupt is similar to a regular or hardware interrupt: there is a trigger that invokes the execution of an ISR. On the Cortex™-M, there are two software interrupts: supervisor call and PendSV (vectors at 0x00000028 and 0x00000038 respectively). The difference between hardware and software interrupts is the trigger. Hardware interrupts are triggered by hardware events, while software interrupts are triggered explicitly by software. For example, to invoke a PendSV, the software sets bit 28 of the NVIC_INT_CTRL_R register. Same as trap.

software maintenance Process of verifying, changing, correcting, enhancing, and extending software.

solenoid A discrete motion device (on/off) that can be controlled by the computer usually by activating an electromagnet. For example, electronic door locks on automobiles.

source code Programs in human readable format created with an editor.

space A digital value of false or logic 0. Contrast with mark.

span Same as range.

spatial resolution The volume over which the DAS collects information about the measurand.

specificity The specificity of a transducer is the relative sensitivity of the device to the signal of interest versus the sensitivity of the device to other unwanted signals. The sensitivity of a qualitative DAS that detects events is the percentage of events detected by the system that are actually true events.

spinlock semaphore A semaphore where the threads will spin (run but do no useful function) when they execute wait on a busy semaphore. Contrast to blocking semaphore.

stabilize The debugging process of stabilizing a software system involves specifying all its inputs. When a system is stabilized, the output results are consistently repeatable. Stabilizing a system with multiple real-time events, like input devices and time-dependent conditions, can be difficult to accomplish. It often involves replacing input hardware with sequential reads from an array or disk file.

stack Last in first out data structure located in RAM and used to temporarily save information.

stack pointer (SP) A register in the processor that points to the RAM location of the stack.

start bit An overhead bit(s) specifying the beginning of the frame, used in serial communication to synchronize the receiver shift register with the transmitter clock. See also stop bit, even parity and odd parity.

starvation A condition that occurs with a priority scheduler where low priority threads are never run.

static allocation Data structures such as an FSM or TCB that are defined at assembly or compile time and exist throughout the life of the software. Contrast to dynamic allocation.

static RAM Volatile read/write storage built from three transistors having fast speed, and not requiring refresh. Contrast with dynamic RAM.

stator The part of a motor that remains stationary. Same as frame.

stepper motor A motor that moves in discrete steps.

stop bit An overhead bit(s) specifying the end of the frame, used in serial communication to separate one frame from the next. See also start bit, even parity and odd parity.

strain gauge A transducer that converts displacement into electric resistance. It can also be used to measure force or pressure.
string A sequence of ASCII characters, usually terminated with a zero.
symbol table A mapping from a symbolic name to its corresponding 16-bit address, generated by the assembler in pass one and displayed in the listing file.
synchronous bus a communication protocol that has a central clock; there is no feedback from the memory to the processor, so every memory cycle takes exactly the same time; data transfers (put data on bus, take data off bus) are synchronized to the central clock
synchronous protocol a system where the two devices share the same clock.
synchronous serial interface (SSI) A device to transmit data with synchronous serial communication protocol. Same as SPI.
tachometer a sensor that measures the revolutions per second of a rotating shaft.
tail recursion A technique where the recursive call occurs as the last action taken by the function. See also recursion, binary recursion, and linear recursion.
thermal noise A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as Johnson noise and white noise.
thermistor A nonlinear transducer that converts temperature into electric resistance.
thermocouple A transducer that converts temperature into electric voltage.
thread The execution of software that cooperates with other threads. A thread embodies the action of the software. One concept describes a thread as the sequence of operations including the input and output data.
thread control block TCB Information about each thread.
three-pole relay Three separate and complete relays, which are activated together (see single pole).
three-pole switch Three separate and complete switches. The switches are electrically separate, but mechanically connected. The three switches turned on and off together (see single pole).
throughput The information transfer rate, the amount of data transferred per second. Same as bandwidth.
time constant The time to reach 63.2% of the final output after the input is instantaneously increased.
time profile and execution profile Time profile refers to the timing characteristic of a program and execution profile refers to the execution pattern of a program.
time to deadline The time between now and the deadline.
tolerance The maximum deviation of a parameter from a specified value.
total harmonic distortion (THD) A measure of the harmonic distortion present and is defined as the ratio of the sum of the powers of all harmonic components to the power of the fundamental frequency.
transducer A device that converts one type of signal into another type.
trap A trap is similar to a regular or hardware interrupt: there is a trigger that invokes the execution of an ISR. On the Cortex™-M, there are two software interrupts: supervisor call and PendSV (vectors at 0x00000028 and 0x00000038 respectively). The difference between hardware and software interrupts is the trigger. Hardware interrupts are triggered by hardware events, while software interrupts are triggered explicitly by software. For

example, to invoke a PendSV, the software sets bit 28 of the NVIC_INT_CTRL_R register. Same as software interrupt.

tristate The state of a tristate logic output when off or not driven.

tristate logic A digital logic device that has three output states low, high, and off (HiZ).

truncation The act of discarding bits as a number is converted from one format to another.

two's complement A number system used to define signed integers. The MSB defines whether the number is negative (1) or positive (0). To negate a two's complement number, one first complements (flip from 0 to 1 or from 1 to 0) each bit, then add 1 to the number.

two-pole relay two separate and complete relays, which are activated together (same as double pole).

two-pole switch Two separate and complete switches. The switches are electrically separate, but mechanically connected. The two switches turned on and off together which are activated together, same as double-pole.

ultrasound A sound with a frequency too high to be heard by humans, typically 40 kHz to 100 MHz.

unbuffered I/O The hardware and software are tightly coupled so that both wait for each other during the transmission of data.

unipolar stepper motor A stepper motor where the current flows in only one direction (on/off) along the interface wires; a stepper with 5 or 6 interface wires.

universal asynchronous receiver/transmitter (UART) A device to transmit data with asynchronous serial communication protocol, same as ACIA.

unsigned binary A mechanism to represent unsigned integers where all 0's represents the value 0, and all 1's represents is the largest positive number.

vector A 32-bit address in ROM containing the location of the interrupt service routines. See also reset vector and interrupt vector.

velocity factor (VF) The ratio of the speed at which information travels relative to the speed of light.

vertical parity The normal parity bit calculated on each individual frame, can be even or odd parity

$\boldsymbol{V_{OH}}$ The smallest possible output voltage when the signal is high, and the current is less than I_{OH}.

$\boldsymbol{V_{OL}}$ The largest possible output voltage when the signal is low, and the current is less than I_{OL}.

volatile A condition where information is lost when power is removed.

volatile A property of a variable in C, such that the value of the variable can change outside the immediate scope of the software accessing the variable.

voltage follower An analog circuit with gain equal to 1, large input impedance and small output impedance. Same as follower.

vulnerable window Locations within a software module, which if an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash, etc.) Same as critical section.

white noise A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as Johnson noise and thermal noise.

wire-or-mode A digital logic output that has two states low and HiZ. Same as open collector.
word Four bytes containing 32 bits.
workstation A powerful general purpose computer system having a price in the $3K to 50K range and used for handling large amounts of data and performing many calculations.
write cycle data is sent from the processor to the memory or output device, the address bus specifies the memory or output device location and the data bus contains the information
write data available time interval (start,end) during which the data will be valid during a write cycle, determined by the processor
write data required time interval (start,end) during which the data should be valid during a write cycle, determined by the memory module
XON/XOFF A protocol used by printers to feedback the printer status to the computer. XOFF is sent from the printer to the computer in order to stop data transfer, and XON is sent from the printer to the computer in order to resume data transfer.
Z Transform A transform equation converting a digital time-domain sequence into the frequency domain. In both the time and frequency domain it is assumed the signal is band limited to 0 to ½fs.
zero A place in the frequency domain where the filter gain is zero.

Appendix 2. Solutions to Checkpoints

Checkpoint 1.1: A characteristic of a system that can guarantee that important tasks get run at the correct time. We define latency as the difference between the time a task is scheduled to run, and the time when the task is actually run. A real-time system guarantees the latency will be small and bounded.
Checkpoint 1.2: An embedded system performs a specific dedicated operation where the computer is hidden or embedded inside the machine.
Checkpoint 1.3: Minimize size, minimize weight, minimize power, provide for proper operation in harsh environments, maximize safety, and minimize cost.
Checkpoint 1.4: Multiple busses allow multiple operations to occur in parallel, resulting in higher performance (more operations/sec).
Checkpoint 1.5: The system does not run slower during debugging, because debugger functions occur simultaneously with program operation.
Checkpoint 1.6: Variables, the heap, and the stack go in RAM. Constants and machine code go in ROM. Basically, items that can change over time go in RAM and items that do not change go in ROM.
Checkpoint 1.7: The ROM on our microcontroller is electrically erasable programmable read only memory (EEPROM). So yes the software can erase the memory and reprogram it. Under normal conditions however software does not write to ROM. However, you can create a file system using a piece of ROM, where your software will be writing to ROM..
Checkpoint 1.8: 0x2200.0000 + 32*n + 4*b=0x2200.0000 + 32*0x1010 +4*3 =0x2200.0000 +0x20200 + 0x0C = 0x2202.020C.
Checkpoint 1.9: 0x2200.0000 + 32*n + 4*b=0x2200.0000 + 32*0x10000+4*22 =0x2200.0000 +0x200000 + 0x58 = 0x2220.0058.
Checkpoint 1.10: 0x4200.0000 + 32*n + 4*b=0x4200.0000 + 32*0x30 +4*7 =0x4200.0000 +0x00600 + 0x1C = 0x4200.061C.
Checkpoint 1.11: R13 is the stack pointer, used to create temporary storage (also called SP). R14 is the link register (also called LR), containing the return address when a function is called. R15 is the program counter, containing the address of the instruction as software executes (also called PC).
Checkpoint 1.12: The I bit in bit 0 of the PRIMASK register. If I=0 interrupts are enabled. If I=1 interrupts are disabled (postponed).
Checkpoint 1.13: A pin is an individual wire on the microcontroller, pins can be used for input, output, debugging, or power. A port is a collection of input/output pins with a common operation.
Checkpoint 1.14: Parallel, serial, analog and time.
Checkpoint 1.15: The addressing mode specifies how the instruction accesses data.
Checkpoint 1.16: Data are numbers and addresses are memory locations that point to data. The processor does not know if a value in R0 is data or an address. It is the responsibility of the programmer (you) to use data as numbers and addresses as pointers in the way you write your programs.
Checkpoint 1.17: Since this instruction pushes 4 registers, the SP is decremented by 16.
Checkpoint 1.18: The return address is saved in the link register, R14 or LR. However, when a first function calls a second function, the first function must save the LR onto the stack.

Checkpoint 1.19: Standards allows software written by one company to work properly with software written by another company. A similar concept is CMSIS, which allows the standardization of I/O functions, see **http://www.keil.com/pack/doc/CMSIS/General/html/index.html**.
Checkpoint 1.20: A pointer is an address that points to data. Pointers are important because they allow us to pass large amounts of data with a single 32-bit entity.
Checkpoint 1.21: An array of 10 elements is accessed with indices from 0 to 9.
Checkpoint 1.22: A linked list is a collection of nodes, where each node contains data and a pointer to the next node. The advantage of linked list is the data can grow and shrink in size, and you can sort the order dynamically. In real-time systems we must guarantee execution of important tasks occur at the proper time, so we will be careful when implementing flexible behavior, which in some instances may not finish. Sometimes we sacrifice flexibility of linked lists for the stability and simplicity of arrays.
Checkpoint 1.23: This is internal fragmentation because it is wasted space for efficiency or the convenience of the operating system.
Checkpoint 1.24: Search the free list to see if the address **&Heap[SIZE*i]** is free.
Checkpoint 1.25: Ignore size parameter, return 100 bytes regardless of the request.
Checkpoint 1.26: The block is lost. This is an example of a memory leak.
Checkpoint 1.27: Sort the free blocks by size using a binary tree. This way it will be faster to search for the best free block during allocation.
Checkpoint 1.28: Each block has two counters. Dividing a block into two creates one more block. There needs to be two more counters for the new block.
Checkpoint 1.29: Weird and crazy bugs will occur, because that memory may be allocated to another task.
Checkpoint 1.30: The existence of the instrument has a small but inconsequential effect on the system performance. The time to execute the instrument is small compared to the time between executions of the instrument. There are three advantages of leaving the instruments in the final system. First, the system was tested with the instruments and works to specification with the instruments. There is no guarantee the system will still work if the instruments are removed. Second, the instruments could provide run time checks to catch failures during operation. Third, the instruments could be used during system checkup (recalibration, diagnostic checkup etc.)

Checkpoint 2.1: Not all pins of a port must have the same direction. Some may be inputs while others are outputs. Furthermore, some pins may be off, meaning neither input or output.
Checkpoint 2.2: If we activate the HFXT to run the microcontroller at 48 MHz, then the SysTick counter decrements every 20.83 ns. To make it interrupt every 10ms, it should interrupt every 480000 cycles. Thus, we set reload to 479999.
Checkpoint 2.3: Since real-time events trigger interrupts, and the ISR software services the requests, disabling interrupts will postpone the response causing latency or jitter. The maximum jitter will be the maximum time running with interrupts disabled.
Checkpoint 2.4: Notice there are two disable interrupt and two enable interrupt functions, occurring in this order: 1) disable, 2) disable, 3) enable, 4) enable. Interrupts will be incorrectly enabled after step 3). Since the 1-4 represents a critical section and 2-3 is inside this section, a bug will probably be introduced. In this example **Stuff1B** runs with interrupts enabled.

```
Critical1              Critical2
  Disable // 1          Disable // 2
  Stuff1A               Stuff2A
  Call Critical2              Enable  // 3
  Stuff1B               return
  Enable // 4
  return
```

Checkpoint 2.5: Negative logic means when we touch the switch the voltage goes to 0 (low). Formally, negative logic means the true voltage is lower than the false voltage. Positive logic means when we touch the switch the voltage goes to +3.3 (high). Formally, positive logic means the true voltage is higher than the false voltage.

Checkpoint 2.6: For PF4, we need input with pull-up. DIR bit 4 is low (input), AFSEL bit 4 is low (not alternate), PUE bit 4 high (pull-up) and PDE bit 4 low (not pull-down). For PF0, we also need input with pull-down. DIR bit 0 is low (input), AFSEL bit 0 is low (not alternate), PUE bit 0 high (pull-up) and PDE bit 0 low (not pull-down).

Checkpoint 2.7: For the TM4C one interrupt is generated, both flags are set, and both counts will be increments. Compare this to the MSP432 version that will generate two sequential interrupts and each interrupt will service one request. In both cases, no events are lost.

Checkpoint 2.8: There is 1 byte of data per 10 bits of transmission. So, there are 11520 bytes/sec.

Checkpoint 2.9: The RxFifo is empty when there is no input data. Software is waiting for hardware. We classify this condition as I/O bound, because the system bandwidth is limited by I/O hardware.

Checkpoint 2.10: The TxFifo is empty when there is no output data. Hardware is waiting for software. We classify this condition as CPU bound, because the system bandwidth is limited by software execution speed.

Checkpoint 2.11: PWM: on the cycle when the timer equals the value in the Match Register or the Interval Load Register.

Checkpoint 2.12: PWM: output pin cleared (set if inverting mode) on match or set (cleared if inverting mode) on reload.

Checkpoint 2.13: 1V*16384/2.5V = 6553 (or 6554) . The TM4C range is 0 to 3.3V, 1V*4095/3.3V = 1241.

Checkpoint 2.14: P1OUT ^= 0x08; GPIO_PORTA_DATA_R ^= 0x08;

```
#define PA3 (*((volatile uint32_t *)0x40000020))
#define Debug_HeartBeat() (PA3 ^= 0x08)
```

Checkpoint 3.1: A program is a list of commands, while a thread is the action cause by the execution of software. For example, there might be one copy of a program that searches the card catalog of a library, while separate threads are created for each user that logs into a terminal to perform a search. Similarly, there might be one set of programs that implement the features of a window (open, minimize, maximize, etc.), while there will be a separate thread for each window created.

Checkpoint 3.2: Threads can't communicate with each other using the stack, because they have physically separate stacks. Global variables will be used, because one thread may write to the

global, and another can read from it.

Checkpoint 3.3: It is hard real time because if the response is late, data may be lost.

Checkpoint 3.4: It is firm real time because it causes an error that can be perceived but the effect is harmless and does not significantly alter the quality of the experience.

Checkpoint 3.5: It is soft real time because the faster it responses the better, but the value of the system (bandwidth is amount of data printed per second) diminishes with latency.

Checkpoint 3.6: With the flowchart in Figure 3.8, the Status will be set twice and the first data value will be lost. We will fix this error in the next using a first in first out (FIFO) queue.

Checkpoint 3.7: The system will not work, because there is more work to do than there are processor resources to accomplish them.

Checkpoint 3.8: The system will work some of the time, but there are times the system will not work.

Checkpoint 3.9: The function **OS_Wait** will crash because it is spinning with interrupts disabled.

Checkpoint 3.10: The function **OS_Wait** has a critical section around the read-modify-write access to the semaphore. If we remove the mutual exclusion, multiple threads could pass.

Checkpoint 3.11: Notice this function discards the new data on error

```
void SendMail(uint32_t int data){
  if(Send){
    Lost++; // discard new data
  }else{
    Mail = data;
    OS_Signal(&Send);
  }
}
```

Checkpoint 4.1: Each thread runs for 1ms, so each thread runs every 5ms. The spinning thread will be run 200 times, wasting 200ms while it waits for its semaphore to be signaled. This is a 20% waste of processor time.

Checkpoint 4.2: Other threads run for 1 ms each, the semaphore is checked every 4 ms. However, the amount of time wasted will be quite small because the spinning thread will go through the loop once and suspend. Obviously, once the semaphore goes above 0, the OS_Wait will return.

Checkpoint 4.3: The worst case is you must look at all 5 blocked threads, so the while loop executes 5 times. This is a waste of 5*150 = 750ns. Since the scheduler runs every 1 ms, this waste is 0.075% of processor time.

Checkpoint 4.4: Since Signal increments and Wait decrements, we expect the average to be equal. On average, over a long period of time, the number of calls to Wait equals the number of calls to Signal. If Signal were called more often, then the semaphore value would become infinite. If Wait were called more often, then all threads would become blocked/stalled.

Checkpoint 4.5: Since put enters data and get removes, we expect the average to be equal. If put were called more often, then the FIFO would become full and another call to put could not occur. If get were called more often, then FIFO would become empty and another successful call to get

could not occur. If the FIFO can store N pieces of data, then the total number of successful puts minus the total number of successful gets must be a value between 0 and N. On average, over a long period of time, the number of calls to put equals the number of calls to get.

Checkpoint 4.6: If CurrentSize is 0, the FIFO is empty. If CurrentSize is equal to FIFOSIZE, the FIFO is full.

Checkpoint 4.7: Use AND instead of modulo divide when incrementing the index because it is faster

```
PutI = (PutI+1)&(FIFOSIZE-1);
GetI = (GetI+1)&(FIFOSIZE-1);
```

Checkpoint 5.1: This priority scheduler must look at them all, so it will run *N* times through the loop. Looking at all the threads is ok if *N* is small, but becomes inefficient if I is large.

Checkpoint 5.2: The maximum latency is 20 ms, because the switch will be recognized at the next interrupt. The minimum latency is 0, and the latencies are uniformly distributed from 0 to 20, so the average is 10 ms.

Checkpoint 6.1: At 60 Hz, f/f_s is 1/6.

$$\textbf{Gain} = 0.5\sqrt{\{1+\cos(6\pi/6)\}^2+\{\sin(6\pi/6)\}^2} = \sqrt{\{1-1\}^2+\{0\}^2} = 0$$

Checkpoint 6.2: If the gain is larger than one, amplification occurs. For example, if the gain is 1.2, if you put in a sinusoidal wave with amplitude 100, then the output of the filter will be a sinusoidal wave with amplitude 120. This is important because a filtered signal from an 8-bit ADC will not fit into an 8-bit variable.

Checkpoint 6.3: The Q is much higher for the IIR filter. This means it rejects just 60 Hz, and passes most of the other frequencies. This greatly improved performance comes with only a modest increase the computational complexity. The additional computation is 2 multiplies and a subtraction. The performance for the IIR filter is superior.

Checkpoint 6.4: First, sum all the positive terms, 76050. The largest positive value will be if the ADC values for the positive terms are 4095 and the ADC values for the negative terms is 0. 76050*4095 is less than 2^{31}. Next, sum all the negative terms, -76048. The largest negative value will be if the ADC values for the negative terms are 4095 and the ADC values for the positive terms is 0. -76048*4095 is greater than -2^{31}. The input is bounded from 0 to 4095 because the data comes from the 12-bit ADC. The largest gain in this filter is 5, the fixed-point coefficient is 16384. 4095*5*16384 will fit in the 32-bit signed intermediate result, **sum**.

Checkpoint 6.5: Because of the linear phase the *h(n)* filter coefficients are symmetric. Notice that *h(k)* equals *h(*50-*k)*. For example, $4\cdot x(n)+ 4\cdot x(n-50)$ can be replaced with $4\cdot(x(n)+x(n-50))$. In general, $h(k)\cdot x(n-k)+ h(50-k)\cdot x(n-50-k)$ can be replaced with $h(k)\cdot(x(n-k)+x(n-50-k))$, saving 25 multiplies.

Checkpoint 7.1: Both refer to the speed of communication. Latency is the response time to a question and bandwidth is the information transfer rate.

Checkpoint 7.2: If we do not meet the latency requirement, that data is lost. If it happens every time the system doesn't work. If it happens occasionally, it will run slow because we will have to wait for the disk to spin around one revolution and try it again.

Checkpoint 7.3: A portion of the sound is lost, and it will sound like a skip. We may also hear a

click because the waveform is discontinuous. It is firm real time because it causes an error that can be perceived but the effect is harmless and does not significantly alter the quality of the experience.

Checkpoint 7.4: The system runs slow, because the transmitter will timeout and try to resend the packets.

Checkpoint 7.5: The bidirectional driver has three possibilities, determined by two control pins. An example of this type of logic is the 74HC245. It can drive data left to right, making the left input and right output. It can drive data right to left, making the right input and left output. The third possibility is that the device can be off, driving neither the left nor the right. This is a noninverting driver, so the output equals the input.

Checkpoint 7.6: Substitute the four bidirectional data bus drivers with four unidirectional tristate drivers. All four data bus drivers operate in the direction of the simplex transfer (left to right). The bank-switched memory looks like a write-only memory to the computer and a read-only memory to the I/O hardware.

Checkpoint 7.7: The maximum latency for cycle steal DMA is one bus cycle, assume there is only one DMA channel active. If there is more than one DMA channel operating, one DMA request may have to wait for another.

Checkpoint 7.8: On some systems the latency is only one bus cycle. On others it may be 2 or 3 bus cycles. In all cases it is very short.

Checkpoint 7.9: On most systems, the instruction must finish, so the latency will be the maximum instruction length. In all cases burst DMA has a longer latency than cycle steal.

Checkpoint 8.1: On average, each file wastes ½ n bytes. Since this is inside the file, this wasted space is classified as internal fragmentation.

Checkpoint 8.2: The best way to cut the wood is obviously at one end or the other, generating the 2-meter piece and leaving 8 meters free. If you were to cut at the 4-meter and 6-meter spots, you would indeed have the 2-meter piece as needed, but this cutting would leave you two 4-meter leftover pieces. The largest available piece now is 4 meters, but the total amount free would be 8 meters. This condition is classified as external fragmentation.

Checkpoint 8.3: The largest contiguous part of the disk is 8 blocks. So the largest new file can have 8*512 bytes of data (4096 bytes). This is less than the available 16 free blocks, therefore there is external fragmentation.

Checkpoint 8.4: First fit would put the file in block 1 (block 0 has the directory). Best fit would put the file in block 10, because it is the smallest free space that is big enough. Worst fit would put it in block 14, because it is the largest free space.

Checkpoint 8.5: A gibibyte is 2^{30} bytes. Each sector is 2^{12} bytes, so there are 2^{18} sectors. So you need 2^{18} bits in the table, one for each sector. There are 2^3 bits in a byte, so the table should be 2^{15} (32768) bytes long.

Checkpoint 8.6: 2 Gibibytes is 2^{31} bytes. 512 bytes is 2^9 bytes. 31-9 = 22, so it would take 22 bits to store the block number.

Checkpoint 8.7: 2 Gibibytes is 2^{31} bytes. 32k bytes is 2^{15} bytes. 31-15 = 16, so it would take 16 bits to store the block number.

Checkpoint 8.8: There are 16 free blocks, they can all be linked together to create one new file. This means there is no external fragmentation.

Checkpoint 8.9: There are many answers. One answer is you could store a byte count in the directory. Another answer is you could store a byte count in each block.
Checkpoint 8.10: 16+9=25. 2^{25} is 32 Mebibytes, which is the largest possible disk.
Checkpoint 8.11: There are $2^{31}/2^{10}=2^{21}$ blocks, so the 21-bit block address will be stored as a 32-bit number. One can store 1024/4=256 index entries in one 1024-byte block. So the maximum file size is $256*1024 = 2^8*2^{10} = 2^{18} = 256$ kibibytes. You can increase the block size or store the index in multiple blocks.
Checkpoint 8.12: There are 15 free blocks, and they can create an index table using all the free blocks to create one new file. This means there is no external fragmentation.
Checkpoint 8.13: There are 15 free blocks, they can create FAT using all the free blocks to create one new file. Each block is 512 bytes, so the largest file is 15 time 512 bytes; there is no external fragmentation.
Checkpoint 8.14: Each directory entry now requires 10 bytes. You could have 50 files, leaving some space for the free space management.
Checkpoint 8.15: Change the 1024 to 4096.

Checkpoint 9.1: Most people communicate in half-duplex. Normally, when we are talking, the sound of our voice overwhelms our ears, so we usually cannot listen while we are talking.
Checkpoint 9.2: Since information is encoded as energy, and data is transferred at a fixed rate, each energy packet will exist for a finite time. Energy per time is power.
Checkpoint 9.3: If the units of a signal x is something like volts or watts, we cannot take the $\log_{10}(x)$, because the units of $\log_{10}(x)$ is not defined. Whenever we use the $\log_{10}$ to calculate the amplitude of a signal, we always perform the logarithm on a value without dimensions. In other words, we always perform the logarithm on a ratio of one signal to another.
Checkpoint 9.4: The performance measure for a storage system is information density in bits/cm^3.
Checkpoint 9.5: With open collector outputs, the low will dominate over HiZ. The signal will be low.

Checkpoint 10.1: The V_{OL} of the 7406 at 40 mA will be 0.7V. This means there will be 4.3V across the coil.
Checkpoint 10.2: If they are too close, then the system can turn on-off-on-off-… very quickly, causing the electromagnetic relays to prematurely fail. If they are too far apart, then the system will oscillate with large positive and negative errors.
Checkpoint 10.3: Every interrupt, the actuator would be increased or decreased, causing a lot of output changes.
Checkpoint 10.4: If the interrupt period were too small, the actuator would be increased to maximum or decreased to minimum, causing it to behave like a bang-bang controller. Basically, the plant would not have time to react to changes in the actuator.
Checkpoint 10.5: The output will saturate. The error increases to a very large positive value or decreases down to a very large negative value.
Checkpoint 10.6: The limit of the discrete integral as Δt goes to zero is the continuous integral.
Checkpoint 10.7: The limit of the discrete derivative as Δt goes to zero is the continuous derivative.

Checkpoint 10.8: Yes. Let **watts** be the units of the actuator output and **RPM** be the units of the sensor input. The units of the lag **L** is **sec**. The units of the rate **R** is **cm/sec**. The units of Δ**U** is **watts**.

Proportional	$\mathbf{K_P} = 1.2\ \Delta\mathbf{U}/(\mathbf{L}*\mathbf{R})$	**watts/(sec*(RPM sec)) = watts/ RPM**
Integral	$\mathbf{K_I} = 0.5\ \mathbf{K_P}\ /\mathbf{L}$	**watts/(RPM-sec)**
Derivative	$\mathbf{K_D} = 0.5\ \mathbf{K_P}\ \mathbf{L}$	**(watts-sec)/RPM**

Checkpoint 10.9: E = X*-X, so the error is very negative, causing the P term to be very negative, making U=100. This removes power and gravity will force it down.

Checkpoint 10.10: SlowDown=WayTooFast+SpeedingUp*LittleBitFast=50+(40*60)=50

> The true engineering experience occurs not with your eyes and ears, but rather with your fingers and elbows. In other words, engineering education does not happen by listening in class or reading a book; rather it happens by designing under the watchful eyes of a patient mentor. So, go build something today, then show it to someone you respect!

Reference Material

Vector address	Number	IRQ	ISR name in **Startup.s**	NVIC	Priority bits
0x00000038	14	-2	**PendSV_Handler**	**NVIC_SYS_PRI3_R**	23 – 21
0x0000003C	15	-1	**SysTick_Handler**	**NVIC_SYS_PRI3_R**	31 – 29
0x00000040	16	0	**GPIOPortA_Handler**	**NVIC_PRI0_R**	7 – 5
0x00000044	17	1	**GPIOPortB_Handler**	**NVIC_PRI0_R**	15 – 13
0x00000048	18	2	**GPIOPortC_Handler**	**NVIC_PRI0_R**	23 – 21
0x0000004C	19	3	**GPIOPortD_Handler**	**NVIC_PRI0_R**	31 – 29
0x00000050	20	4	**GPIOPortE_Handler**	**NVIC_PRI1_R**	7 – 5
0x00000054	21	5	**UART0_Handler**	**NVIC_PRI1_R**	15 – 13
0x00000058	22	6	**UART1_Handler**	**NVIC_PRI1_R**	23 – 21
0x0000005C	23	7	**SSI0_Handler**	**NVIC_PRI1_R**	31 – 29
0x00000060	24	8	**I2C0_Handler**	**NVIC_PRI2_R**	7 – 5
0x00000064	25	9	**PWMFault_Handler**	**NVIC_PRI2_R**	15 – 13
0x00000068	26	10	**PWM0_Handler**	**NVIC_PRI2_R**	23 – 21
0x0000006C	27	11	**PWM1_Handler**	**NVIC_PRI2_R**	31 – 29
0x00000070	28	12	**PWM2_Handler**	**NVIC_PRI3_R**	7 – 5
0x00000074	29	13	**Quadrature0_Handler**	**NVIC_PRI3_R**	15 – 13
0x00000078	30	14	**ADC0_Handler**	**NVIC_PRI3_R**	23 – 21
0x0000007C	31	15	**ADC1_Handler**	**NVIC_PRI3_R**	31 – 29
0x00000080	32	16	**ADC2_Handler**	**NVIC_PRI4_R**	7 – 5
0x00000084	33	17	**ADC3_Handler**	**NVIC_PRI4_R**	15 – 13
0x00000088	34	18	**WDT_Handler**	**NVIC_PRI4_R**	23 – 21
0x0000008C	35	19	**Timer0A_Handler**	**NVIC_PRI4_R**	31 – 29
0x00000090	36	20	**Timer0B_Handler**	**NVIC_PRI5_R**	7 – 5
0x00000094	37	21	**Timer1A_Handler**	**NVIC_PRI5_R**	15 – 13
0x00000098	38	22	**Timer1B_Handler**	**NVIC_PRI5_R**	23 – 21
0x0000009C	39	23	**Timer2A_Handler**	**NVIC_PRI5_R**	31 – 29
0x000000A0	40	24	**Timer2B_Handler**	**NVIC_PRI6_R**	7 – 5
0x000000A4	41	25	**Comp0_Handler**	**NVIC_PRI6_R**	15 – 13
0x000000A8	42	26	**Comp1_Handler**	**NVIC_PRI6_R**	23 – 21
0x000000AC	43	27	**Comp2_Handler**	**NVIC_PRI6_R**	31 – 29
0x000000B0	44	28	**SysCtl_Handler**	**NVIC_PRI7_R**	7 – 5
0x000000B4	45	29	**FlashCtl_Handler**	**NVIC_PRI7_R**	15 – 13
0x000000B8	46	30	**GPIOPortF_Handler**	**NVIC_PRI7_R**	23 – 21
0x000000BC	47	31	**GPIOPortG_Handler**	**NVIC_PRI7_R**	31 – 29
0x000000C0	48	32	**GPIOPortH_Handler**	**NVIC_PRI8_R**	7 – 5
0x000000C4	49	33	**UART2_Handler**	**NVIC_PRI8_R**	15 – 13
0x000000C8	50	34	**SSI1_Handler**	**NVIC_PRI8_R**	23 – 21
0x000000CC	51	35	**Timer3A_Handler**	**NVIC_PRI8_R**	31 – 29
0x000000D0	52	36	**Timer3B_Handler**	**NVIC_PRI9_R**	7 – 5
0x000000D4	53	37	**I2C1_Handler**	**NVIC_PRI9_R**	15 – 13
0x000000D8	54	38	**Quadrature1_Handler**	**NVIC_PRI9_R**	23 – 21
0x000000DC	55	39	**CAN0_Handler**	**NVIC_PRI9_R**	31 – 29
0x000000E0	56	40	**CAN1_Handler**	**NVIC_PRI10_R**	7 – 5
0x000000E4	57	41	**CAN2_Handler**	**NVIC_PRI10_R**	15 – 13
0x000000E8	58	42	**Ethernet_Handler**	**NVIC_PRI10_R**	23 – 21
0x000000EC	59	43	**Hibernate_Handler**	**NVIC_PRI10_R**	31 – 29
0x000000F0	60	44	**USB0_Handler**	**NVIC_PRI11_R**	7 – 5

0x000000F4	61	45	**PWM3_Handler**	**NVIC_PRI11_R**	15 – 13
0x000000F8	62	46	**uDMA_Handler**	**NVIC_PRI11_R**	23 – 21
0x000000FC	63	47	**uDMA_Error**	**NVIC_PRI11_R**	31 – 29

Table 2.6. Some of the interrupt vectors for the TM4C.

Memory access instructions

```
LDR   Rd, [Rn]       ; load 32-bit number at [Rn] to Rd
LDR   Rd, [Rn,#off] ; load 32-bit number at [Rn+off] to Rd
LDR   Rd, [Rn,#off]! ; load 32-bit number at [Rn+off] to Rd, preindex
LDR   Rd, [Rn],#off  ; load 32-bit number at [Rn] to Rd, postindex
LDRT  Rd, [Rn,#off] ; load 32-bit number unprivileged
LDR   Rd, =value     ; set Rd equal to any 32-bit value (PC rel)
LDRH  Rd, [Rn]       ; load unsigned 16-bit at [Rn] to Rd
LDRH  Rd, [Rn,#off] ; load unsigned 16-bit at [Rn+off] to Rd
LDRH  Rd, [Rn,#off]! ; load unsigned 16-bit at [Rn+off] to Rd, pre
LDRH  Rd, [Rn],#off ; load unsigned 16-bit at [Rn] to Rd, postindex
LDRHT  Rd, [Rn,#off] ; load unsigned 16-bit unprivileged
LDRSH Rd, [Rn]       ; load signed 16-bit at [Rn] to Rd
LDRSH Rd, [Rn,#off] ; load signed 16-bit at [Rn+off] to Rd
LDRSH Rd, [Rn,#off]! ; load signed 16-bit at [Rn+off] to Rd, pre
LDRSH Rd, [Rn],#off ; load signed 16-bit at [Rn] to Rd, postindex
LDRSHT Rd, [Rn,#off] ; load signed 16-bit unprivileged
LDRB  Rd, [Rn]       ; load unsigned 8-bit at [Rn] to Rd
LDRB  Rd, [Rn,#off] ; load unsigned 8-bit at [Rn+off] to Rd
LDRB  Rd, [Rn,#off]! ; load unsigned 8-bit at [Rn+off] to Rd, pre
LDRB  Rd, [Rn],#off ; load unsigned 8-bit at [Rn] to Rd, postindex
LDRBT  Rd, [Rn,#off] ; load unsigned 8-bit unprivileged
LDRSB Rd, [Rn]       ; load signed 8-bit at [Rn] to Rd
LDRSB Rd, [Rn,#off] ; load signed 8-bit at [Rn+off] to Rd
LDRSB Rd, [Rn,#off]! ; load signed 8-bit at [Rn+off] to Rd, pre
LDRSB Rd, [Rn],#off ; load signed 8-bit at [Rn] to Rd, postindex
LDRSBT Rd, [Rn,#off] ; load signed 8-bit unprivileged
LDRD Rd,Rd2,[Rn,#off] ; load 64-bit at [Rn+off] to Rd,Rd2
LDRD Rd,Rd2,[Rn,#off]!; load 64-bit at [Rn+off] to Rd,Rd2,pre
LDRD Rd,Rd2,[Rn],#off ; load 64-bit at [Rn] to Rd,Rd2, postindex
LDMFD  Rn{!}, Reglist ; load reg from list at Rn(inc), !update Rn
LDMIA  Rn{!}, Reglist ; load reg from list at Rn(inc), !update Rn
LDMDB  Rn{!}, Reglist ; load reg from list at Rn(dec), !update Rn
STMIA  Rn{!}, Reglist ; store reg from list to Rn(inc), !update Rn
STMFD  Rn{!}, Reglist ; store reg from list to Rn(dec), !update Rn
STMDB  Rn{!}, Reglist ; store reg from list to Rn(dec), !update Rn
STR   Rt, [Rn]       ; store 32-bit Rt to [Rn]
STR  Rt, [Rn,#off] ; store 32-bit Rt to [Rn+off]
STR  Rt, [Rn,#off]! ; store 32-bit Rt to [Rn+off], pre
STR  Rt, [Rn],#off ; store 32-bit Rt to [Rn], postindex
```

STRT Rt, [Rn,#off] ; store 32-bit Rt to [Rn+off] unprivileged
STRH Rt, [Rn] ; store least sig. 16-bit Rt to [Rn]
STRH Rt, [Rn,#off] ; store least sig. 16-bit Rt to [Rn+off]
STRH Rt, [Rn,#off]! ; store least sig. 16-bit Rt to [Rn+off], pre
STRH Rt, [Rn],#off ; store least sig. 16-bit Rt to [Rn], postindex
STRHT Rt, [Rn,#off] ; store least sig. 16-bit unprivileged
STRB Rt, [Rn] ; store least sig. 8-bit Rt to [Rn]
STRB Rt, [Rn,#off] ; store least sig. 8-bit Rt to [Rn+off]
STRB Rt, [Rn,#off]! ; store least sig. 8-bit Rt to [Rn+off],pre
STRB Rt, [Rn],#off ; store least sig. 8-bit Rt to [Rn], postindex
STRBT Rt, [Rn,#off] ; store least sig. unprivileged
STRD Rd,Rd2,[Rn,#off] ; store 64-bit Rd,Rd2 to [Rn+off]
STRD Rd,Rd2,[Rn,#off]!; store 64-bit Rd,Rd2 to [Rn+off], pre
STRD Rd,Rd2,[Rn],#off ; store 64-bit Rd,Rd2 to [Rn], postindex
PUSH Reglist ; push 32-bit registers onto stack
POP Reglist ; pop 32-bit numbers from stack into registers
ADR Rd, label ; set Rd equal to the address at label
MOV{S} Rd, <op2> ; set Rd equal to op2
MOV Rd, #im16 ; set Rd equal to im16, im16 is 0 to 65535
MOVT Rd, #im16 ; set Rd bits 31-16 equal to im16
MVN{S} Rd, <op2> ; set Rd equal to -op2

Branch instructions

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

```
  CBZ Rn,label         ; branch if Rn zero
  IT{x{y{z}}}cond      ; if then block with x,y,z T(true) or F(false)
  TBB [Rn, Rm]         ; table branch byte
  TBH [Rn, Rm, LSL #1] ; table branch halfword
```

Mutual exclusive instructions

```
  CLREX                        ; clear exclusive
  LDREX{cond}  Rt,[Rn{,#offset}]    ; load 32-bit exclusive
  STREX{cond}  Rd,Rt,[Rn{,#offset}] ; store 32-bit exclusive
  LDREXB{cond} Rt,[Rn]             ; load 8-bit exclusive
  STREXB{cond} Rd,Rt,[Rn]          ; store 8-bit exclusive
  LDREXH{cond} Rt,[Rn]             ; load 16-bit exclusive
  STREXH{cond} Rd,Rt,[Rn]          ; store 16-bit exclusive
```

Miscellaneous instructions

```
  BKPT   #imm     ; execute breakpoint, debug state 0 to 255
  CPSIE F         ; clear faultmask F=0
  CPSIE I         ; enable interrupts  (I=0)
  CPSID F         ; set faultmask F=1
  CPSID I         ; disable interrupts (I=1)
  DMB             ; data memory barrier, memory access to finish
  DSB             ; data synchronization barrier, instructions to finish
  ISB             ; instruction synchronization barrier, finish pipeline
  MRS Rd,SpecReg  ; move special register to Rd
  MSR Rd,SpecReg  ; move Rd to special register
  NOP             ; no operation
  SEV             ; Send Event
  SVC #im8        ; supervisor call (0 to 255)
  WFE             ; wait for event
  WFI             ; wait for interrupt
```

Logical instructions

```
  AND{S} {Rd,} Rn, <op2> ; Rd=Rn&op2    (op2 is 32 bits)
  BFC  Rd,#lsb,#width    ; clear bits in Rn
  BFI  Rd,Rn,#lsb,#width ; bit field insert, Rn into Rd
  ORR{S} {Rd,} Rn, <op2> ; Rd=Rn|op2    (op2 is 32 bits)
  EOR{S} {Rd,} Rn, <op2> ; Rd=Rn^op2    (op2 is 32 bits)
  BIC{S} {Rd,} Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)
  ORN{S} {Rd,} Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)
  TST    Rn, <op2>       ; Rn&op2    (op2 is 32 bits)
  TEQ    Rn, <op2>       ; Rn^op2    (op2 is 32 bits)
  LSR{S} Rd, Rm, Rs      ; logical shift right Rd=Rm>>Rs  (unsigned)
  LSR{S} Rd, Rm, #n      ; logical shift right Rd=Rm>>n   (unsigned)
```

```
ASR{S} Rd, Rm, Rs      ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S} Rd, Rm, #n      ; arithmetic shift right Rd=Rm>>n (signed)
LSL{S} Rd, Rm, Rs      ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n      ; shift left Rd=Rm<<n  (signed, unsigned)
REV    Rd, Rn          ; Reverse byte order in a word
REV16  Rd, Rn          ; Reverse byte order in each halfword
REVSH  Rd, Rn          ; Reverse byte order in the bottom halfword,
                       ; and sign extends to 32 bits
RBIT  Rd, Rn           ; Reverse the bit order in a 32-bit word
SBFX Rd,Rn,#lsb,#width ; signed bit field and extract
UBFX Rd,Rn,#lsb,#width ; unsigned bit field and extract
SXTB {Rd,}Rm{,ROR #n}  ; Sign extend byte
SXTH {Rd,}Rm{,ROR #n}  ; Sign extend halfword
UXTB {Rd,}Rm{,ROR #n}  ; Zero extend byte
UXTH {Rd,}Rm{,ROR #n}  ; Zero extend halfword
```

Arithmetic instructions

```
ADD{S} {Rd,} Rn, <op2> ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12 ; Rd = Rn + im12, im12 is 0 to 4095
CLZ    Rd, Rm          ; Rd = number of leading zeros in Rm
SUB{S} {Rd,} Rn, <op2> ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12 ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2> ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12 ; Rd = im12 – Rn
CMP    Rn, <op2>       ; Rn – op2      sets the NZVC bits
CMN    Rn, <op2>       ; Rn - (-op2)   sets the NZVC bits
MUL{S} {Rd,} Rn, Rm    ; Rd = Rn * Rm       signed or unsigned
MLA    Rd, Rn, Rm, Ra  ; Rd = Ra + Rn*Rm    signed or unsigned
MLS    Rd, Rn, Rm, Ra  ; Rd = Ra - Rn*Rm    signed or unsigned
UDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         unsigned
SDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         signed
UMULL  RdLo,RdHi,Rn,Rm ; Unsigned long multiply 32by32 into 64
UMLAL  RdLo,RdHi,Rn,Rm ; Unsigned long multiply, with accumulate
SMULL  RdLo,RdHi,Rn,Rm ; Signed long multiply 32by32 into 64
SMLAL  RdLo,RdHi,Rn,Rm ; Signed long multiply, with accumulate
SSAT  Rd,#n,Rm{,shift #s} ; signed saturation to n bits
USAT  Rd,#n,Rm{,shift #s} ; unsigned saturation to n bits
```

Notes Ra Rd Rm Rn Rt represent 32-bit registers

```
  value    any 32-bit value: signed, unsigned, or address
  {S}      if S is present, instruction will set condition codes
```

#im8 any value from 0 to 255
#im12 any value from 0 to 4095
#im16 any value from 0 to 65535
{Rd,} if Rd is present Rd is destination, otherwise Rn
#n any value from 0 to 31
#off any value from -255 to 4095
label any address within the ROM of the microcontroller
SpecReg APSR,IPSR,EPSR,IEPSR,IAPSR,EAPSR,PSR,MSP,PSP, PRIMASK,BASEPRI,BASEPRI_MAX,FAULTMASK, or CONTROL.
Reglist is a list of registers. E.g., {R1,R3,R12}
op2 the value generated by <op2>

Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**
ADD Rd, Rn, Rm ; op2 = Rm
ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned
ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned
ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed
ADD Rd, Rn, #constant ; op2 = constant , where **X** and **Y** are hexadecimal digits:

- produced by shifting an 8-bit unsigned value left by any number of bits
- in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

Parameter	PN2222 (I_C=150mA) PN2907 (I_C=150mA)	2N2222 (I_C=500mA) 2N2907 (I_C=500mA)	TIP120 (I_C=3A) TIP125 (I_C=3A)
h_{fe}	100	40	1000
V_{BEsat}	0.6	2	2.5 V
V_{CE} at saturation	0.3	1	2 V

Design parameters for the 2N2222 and TIP120.

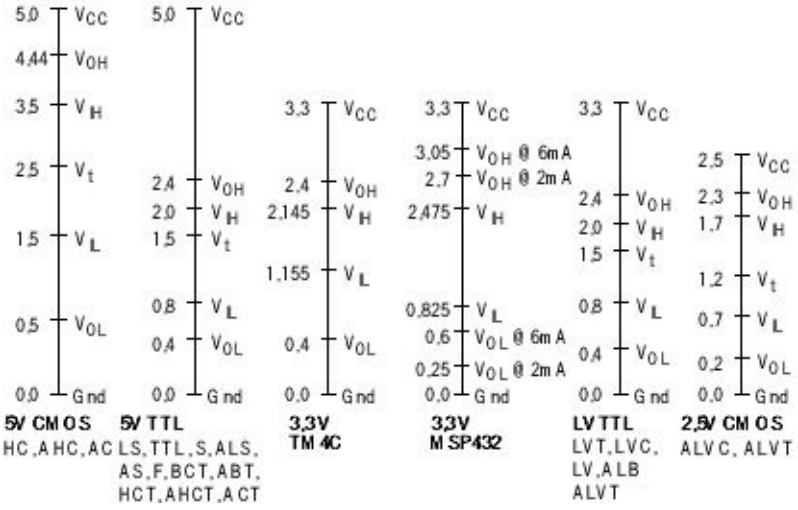
Chip	Current	Comment
L293D	0.6 A	Dual, diodes
L293	1 A	Dual
DRV8848	2 A	Dual, fault status
TPIC0107	3 A	Direction, fault status

L6203	5 A	Dual

H-bridge drivers

Family	*Example*	I_{OH}	I_{OL}	I_{IH}	I_{IL}
Standard TTL	7404	0.4 mA	16 mA	40 µA	1.6 mA
Low Power Schottky	74LS04	0.4 mA	4 mA	20 µA	0.4 mA
High Speed CMOS	74HC04	4 mA	4 mA	1 µA	1 µA
Adv High Speed CMOS	74AHC04	4 mA	4 mA	1 µA	1 µA
MSP432 regular drive	MSP432	6 mA	6 mA	20 nA	20 nA
MSP432 high drive	MSP432	20 mA	20 mA	20 nA	20 nA
TM4C 2mA-drive	TM4C123	2 mA	2 mA	2 µA	2 µA
TM4C 4mA-drive	TM4C123	4 mA	4 mA	2 µA	2 µA
TM4C 8mA-drive	TM4C123	8 mA	8 mA	2 µA	2 µA
TM4C 12mA-drive	TM4C1294	12 mA	12 mA	2 µA	2 µA

The input and output currents of various digital logic families and microcontrollers.



Voltage thresholds for various digital logic families.