Realtime Computing for Mechanical Engineers

Introduction and Laboratory

Rico A. R. Picone
Department of Mechanical Engineering
Saint Martin's University

Joseph L. Garbini
Department of Mechanical Engineering
University of Washington

Thursday 7 April, 2022

Preface

The instructional literature includes many excellent texts on the subject of embedded computing. Appropriately, most such texts, and the courses that use them, are aimed primarily at electrical or computer engineering students. By contrast, this text is designed specifically for mechanical engineering senior undergraduates or first year graduate students. The instruction builds on the typical background of an ME senior, and it focuses on applications of dynamic motion control and digital signal processing that are of particular interest to mechanical engineers.

This course of study is also unique in that learning is organized around a sequence of nine hands-on experiments. Building one upon the other, the experiments proceed from basic interface concepts, through discrete control organization and timing issues, and ultimately to multi-tasked automatic control. Practical information (specific to the lab hardware), as well as, general analytical concepts (applicable to any computing environment) are introduced as necessary to support each experiment.

For students, the text can best be thought of as a journey during which they learn to incorporate (embed) computers into mechanical systems as functional components. They will comprehensively explore issues of user interaction, embedded program organization, timing control, and interface hardware. Along the way, they will learn the process of designing and implementing embedded applications, and become acquainted with the architecture and resources of a modern real-time development environment.

This book evolved during several decades of teaching embedded computing to mechanical engineering students as part of a senior-year curriculum in mechatronics. The goals of the text are first, to provide the embedded computing component for that mechatronics program, culminating in senior capstone design projects. And second, to provide students with a comprehensive foundation in modern embedded computing that can be used and extended throughout an engineering career.

The vehicle for the lab experiments is the National Instruments myRIO-1900 embedded device, programmed in C from an Eclipse development environment on the student's laptop. This compact system features a two-core Xilinx Z-7010 processor, with an appropriate array of interfaces managed by its associated field programmable gate array, and a real-time linux operating system. Having taught this material (over the years) with at least five other microcomputers, we believe that the myRIO is an ideal platform, both for instruction and as the basis of prototypes in the senior capstone course.

Why C? Of course, the myRIO is also readily programmed in LabVIEW. Why have we chosen C as the programming language for this text? While we are big fans of LabVIEW, and use it in many other courses, in this text we seek to give the student a broad exposure to the concept of an integrated development environment (IDE), useful across a wide array of real-time target microcomputers and applications. An engineer working in this field will eventually become familiar with a variety of languages and development environments. But, C/C++ is the best place for a student to begin, and remains today the lingua franca of embedded computing.

This text is intended for senior mechanical engineering undergraduates. We assume that students will have at least the usual first courses in differential equations, newtonian dynamics, and time/frequency domain analysis of dynamic systems. Background in the subjects of automatic control, instrumentation, and signal processing is useful and will help motivate the material, but is not essential.

Knowledge of computer programming is assumed. However, no specific a priori familiarity with the C programming language is required. By degrees, the student will be introduced to C as he progresses through the lab exercises. We suggest that the student rely on a companion text on C as a reference. Several are listed in the bibliography. The text also recommends that students use MATLAB to evaluate experimental results.

An additional point: Although the book is presented for the classroom, there is no reason why a determined student couldn't master the text and its experiments through individual self-study.

Some ancillary hardware is necessary for the experiments: a keypad and LCD display, an oscilloscope, a brushless DC motor and inertial load with incremental encoder, an appropriate motor amplifier, and an analog signal generator. Although many forms of this equipment can be used, specific suggestions are detailed in Appendix X.

Before we begin, let's briefly chart this journey into embedded computing by outlining the nine core experiments.

Lab Exercise 00 In this first lab, students set up their development environment and insure that it works properly with the myRIO targets. They exercise debugging techniques, and become familiar with structure of an embedded C program.

Lab Exercises 01, 02, and 03 During this sequence of labs, students learn the elements of C, and build a user interface, including the low level drivers for a keypad and LCD display.

Lab Exercise 04 Students implement a finite state machine, and explore elementary timing elementary embedded computing timing.

Lab Exercise 05 External interrupts are introduced, and their potential advantages and limitations are explored.

Lab Exercise 06 Students develop a general-purpose linear discrete dynamic system as the basis of digital signal processing and control.

Lab Exercise 07 Combining elements of all the previous labs, students implement and analyze a multi-tasked closed-loop velocity control.

Lab Exercise 08 In the final lab, students design, implement, analyze a closed-loop position control. Trajectory planning is also examined.

In each of the final three lab exercises (06, 07, and 08), students compare the performance of the digital implementations with continuous system models.

Finally, the authors wish to acknowledge the many former students, teaching assistants, and colleagues that have influenced this work, including Professors Jens E. Jorgensen, Peter L. Balise, and William R. Murray.

Part I

Orientation

00

Getting started

00.1 Introduction to embedded computing

Embedded computers comprise the vast majority of computers, yet are perhaps the least familiar to the uninitiated. A computer in a kitchen appliance, smart thermostat, or pacemaker is embedded—that is, it has most of the following characteristics: it

- performs a specific, limited function;
- performs its function in real-time;
- is small;
- is inexpensive;
- is low-power;
- is ruggedly packaged.

The central processing unit (CPU) of any computer is the electronic circuitry that performs a computer's most basic instructions, such as arithmetic, logic, and input/output. A CPU is typically an integrated circuit (IC, i.e. "microchip" or just "chip"), which is made of a single silicon chip, the size of a human fingernail, transformed into a circuit containing billions of elements, such as transistors, diodes, and capacitors, by a process in which the semiconductor material silicon is selectively doped with impurities, locally changing its conductivity. An IC CPU is called a microprocessor.

Many embedded computers are microcontrollers (μC), which are integrated circuits that include CPUs, memory, and input/output peripherals—classes of computing components that will be described in this chapter. Images of a microprocessor and a microcontroller are found in Figure 00.1.

The instructions carried out by a CPU are rather basic, yet combinations of them can be vastly more complex. This is analogous to words, perhaps relatively simple, alone, being combined to form complex phrases, sentences,

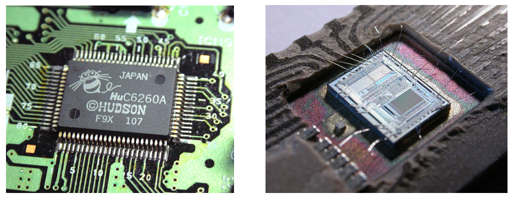


Figure 00.1: (left) a Hudson HuC5260A microprocessor (Baz1521, 2018) and (right) an Intel 8742 microcontroller—including a 12 MHz CPU, 128 B RAM, 2048 B EPROM, and input/output peripherals—broken open (Sameli, 2018).

and books. In fact, this is why a CPU's instructions are said to form a machine language—literally just numbers with predefined meanings, often represented in binary. These languages are very cumbersome for humans to write useful software with, as we will see, so higher level languages are developed. The first level above machine language is called assembly language, which typically assigns a more descriptive mnemonic to represent each instruction ("opcodes"). A list of languages by level is given in Table 00.1. A program is a sequence of instructions that performs some task.

Higher-level languages such as C, Python, and Matlab are called programming languages. They have the important quality of microprocessor independence—that is, they can be written for multiple processors, then translated into lower-level, processor-specific languages. There are two archetypical ways this translation occurs for a program:

compilation When a program is compiled, it is converted en masse into machine code before it is processed. This is often considered to be optimal for performance because each high-level function of the program is converted directly to processor-specific instructions.

interpretation When a program is interpreted, its high-level functions have

Table 00.1: categories of programming languages descending from high-level to low-level.

type	examples
graphical	LabVIEW, Simulink
scripting	Bash, Perl, BASIC
typically interpreted	MATLAB, Python, Ruby
typically compiled	C, C++, Java
assembly	symbolic codes
machine	numerical codes

been pre-translated to machine code such that it can be directly processed "on the fly" without compilation. This is often considered more convenient and easier to implement than compilation.

Most modern programming languages provide both options.[1] Due to typically stringent performance and memory requirements, embedded programs often use compilation. The C programming language is the language-of-choice for embedded computer programming. It can be thought of as being as close as possible to machine language without being processor-specific. It deals with numbers and arithmetic and memory addresses, which is what a processor does, also. That is, it is a low-level, general-purpose language. A limitation of the core language is that it lacks functions for basic operations like reading or writing to a file. The C standard library augments this functionality.

1. Previously, it was common to refer to a specific language as "compiled" or "translated," but this terminology is increasingly obsolete.

An advantage of its compactness is its relatively small size, a key advantage for embedded computing. Other key advantages of C for embedded computing include its ubiquity (C compilers are available for most processors), speed (a result of a small language and good compilers), and energy efficiency (in that executing fewer cycles requires less power). Although the vast majority of embedded computer programming is in C, recently languages such as Python (which is, itself, written in C) have captured a small sliver of the embedded market (Altera, 2018).

00.2 Embedded control of mechanical systems

Feedback control is a powerful idea: a desired system state is compared to a measurement of its actual state and some actuation is applied such that the difference between the desired and the actual state (the error) is minimized. This concept has been realized in many ways since around 300 BC, when the Greeks used a float valve to regulate the flow of water to a relatively constant rate, allowing them to measure time. The 20th Century, especially its latter half, saw a mathematization and vast improvement of control. Electronic circuits began to perform the key "logic" function: for example, "if $\mathtt{a}$ is measured, then do $\mathtt{b}$." Previously, the mechanical organization of the system had to to perform the logic, as in the flow-regulation example illustrated in Figure 00.1.

As circuits have evolved into increasingly powerful microprocessors connected to sensors and actuators, the complexity of implementable logic has grown drastically. But one constraint is persistent: decisions about how the control system should respond with its actuators, given the system's current state, must happen in real-time—that is, now! (Rather, as close to "now" as possible.) Real-time computing for control must not only be fast, but reliably so; that is, the programmer must be able to direct timing. Although computing power has improved steadily, real-time computing remains among the greatest challenges for control systems engineers.

Most feedback control is instantiated with embedded computers because sensors and actuators must be nearby to increase reliability and decrease lag. Therefore, designing controllers (that is, embedded computers and peripherals used for control) requires an understanding of embedded computing hardware and its programming.

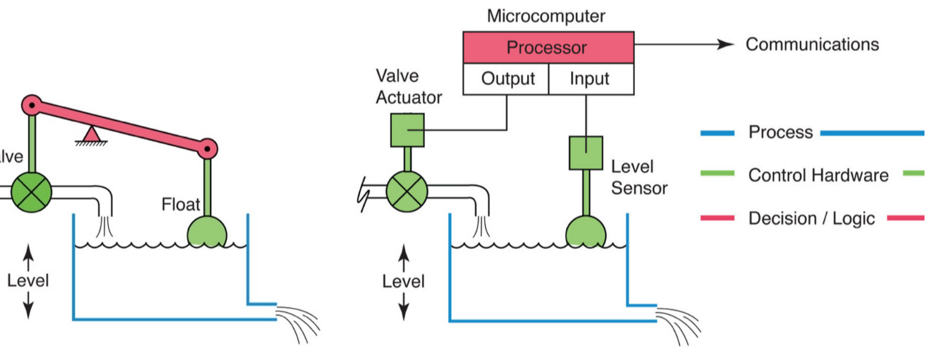


Figure 00.1: level control with (left) a mechanical and with (right) an embedded computer.

00.3 Computer architectures

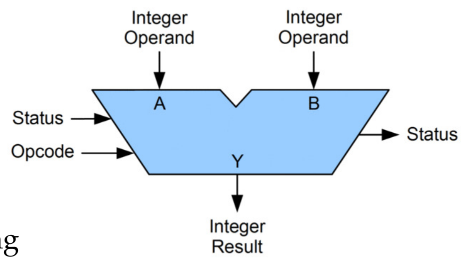


Figure 00.1: ALU block (Lamberson, 2018).

If we descend from high-level programming languages like Python to low-level languages like C, then continue to descend, considerations become more hardware-specific. Computer architectures are descriptions of this down to a fairly concrete level, stopping somewhere above the actual physics of the whole thing.
Embedded computer programming contacts the computer architecture much more than does personal computer programming. In fact, the C language gives us access to different aspects of it.
Consider the four primary components of a computer that follow.[2]

2. This is sometimes called the von Neumann architecture.

central processing unit (CPU) The CPU consists of a control unit (brain) and a datapath (brawn). The control unit receives machine code instructions from memory and controls the other components to perform corresponding tasks. The datapath consists of functional units such as registers (the fastest, smallest-capacity memory), buses (intra-computer communication systems), and arithmetic logic units (ALU) that can perform very basic arithmetic or logical operations (see Figure 00.1 for an illustration).

memory Memory stores data (e.g. numbers, files, etc.) and instructions. Numerous forms exist, but the primary variable of interest is that of speed of access (read/write). Faster tends to mean less capacity and more expense. Permanence is another consideration: some types of memory, called volatile, lose their state when they lose power, while others, called non-volatile, do not. These considerations lead to complex optimization when, as is typically the case, different types of memory are used within the same system. Roughly speaking, data or instructions that are most likely to be read soon are moved to faster, smaller, more expensive memory.

In addition to speed, volatility, and cost there is another important aspect of memory: its rewritability. This divides memory into two categories:

read–only–memory (ROM) of which the state "cannot" be altered, only detected, by the CPU and

read/write memory (RAM) that can be both read and written-to by the CPU.

There are several types of ROM, described as follows.

ROM is masked ROM that is programmed at manufacturing and cannot be altered. It is for mass production and is cheapest.

PROM is field-programmable; that is, it can be programmed after manufacturing, typically only once.

EPROM is erasable (usually with UV light) and reprogrammable after manufacturing, but has a short number of erases: on the order of 100 times.

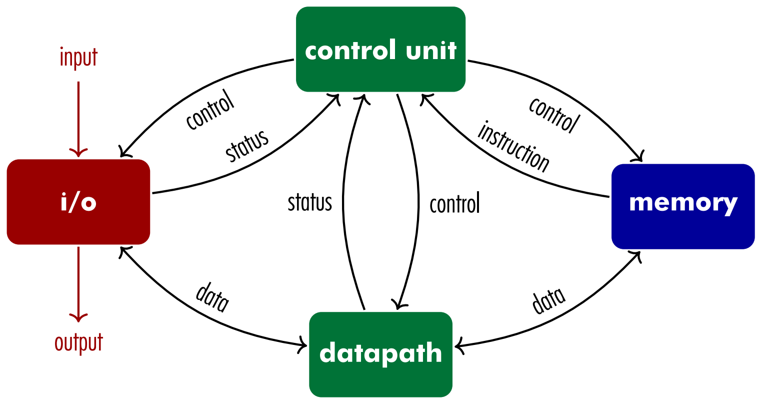


Figure 00.2: high-level schematic of the four primary components of a computer:
🟩 CPU (control unit and datapath), 🟦 memory, 🟥 input, and 🟥 output.

EEPROM is electrically-erasable and non-volatile. It is often called flash memory.

input Inputs are data from devices other than the computer, such as keyboards, sensors, and remotely communicated commands (via, say, the Internet).

output Outputs are data sent to devices other than the computer, such as displays, printers, and actuators (e.g. a motor).

00.4 Numeral systems

The following is a myth in the good sense of the term.

The myth of Niles and Pepper

Niles was an unusual boy with great hair, living in a time before number systems. Quantities were familiar, but symbolic representations of them were not. Niles lived in a grove of trees. A grove on the other side of the hill was home to his friend Pepper, a sassy, no-nonsense girl. One day Niles and Pepper were walking together and, as children do, began arguing about whose grove had more trees. If the argument had been about who had more skipping stones, they could have simply matched up stone-for-stone to discover who had more. But this was impractical with the trees. Niles had an insight:

> We can represent each tree by a drawing and match these to determine who has more.

It went something like this.

Niles: ↟↟↟↟↟↟↟↟↟↟↟↟↟

Pepper: ↟↟↟↟↟↟↟↟↟↟↟↟↟↟↟↟

Not to be discouraged, Niles proposed they compare, instead, the number of trees on each's entire side of the hill. However, there were many more trees, so Pepper suggested they simply draw the symbol ⊤ to represent each tree, to save time. The results were no more-satisfying to Niles.

Niles: ⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤

Pepper: ⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤⊤

Niles pushed on: let's include the neighboring hill on each side. With so many more trees, Pepper suggested a shorthand notation.

> We can compactly represent the number of trees with two symbols $\bigcirc$ and $|$ used in combination.

She explained it to Niles by counting up:

- $\ = \bigcirc$
- $\top = |$
- $\top\top = |\bigcirc$
- $\top\top\top = ||$
- $\top\top\top\top = | \bigcirc \bigcirc$
- $\top\top\top\top\top = | \bigcirc |$
- $\top\top\top\top\top\top = ||\bigcirc$
- $\top\top\top\top\top\top\top = |||.$

That is, each position could take on two symbols, but the position of each symbol denoted its "weight." The far-right symbol represented either a lack $\bigcirc$ or presence $|$ of a single tree. The next symbol to the left was the "overflow" from the first symbol, and therefore represented the number of pairs of trees. The next symbol to the left was the next overflow, representing pairs of pairs of trees.
What fun! They started counting and Pepper immediately recognized a process improvement:

> If we use more symbols, we can represent numbers even more-compactly.

Pepper suggested a symbol for each digit of their hands:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9. \tag{1}$$

Now they could count on their fingers:

- $\ = 0$
- $\top = 1$
- $\top\top = 2$
- $\top\top\top = 3$
- $\top\top\top\top = 4$

- $\top\top\top\top\top = 5$
- $\top\top\top\top\top\top = 6$
- $\top\top\top\top\top\top\top = 7$
- $\top\top\top\top\top\top\top\top = 8$
- $\top\top\top\top\top\top\top\top\top = 9$
- $\top\top\top\top\top\top\top\top\top\top = 10$.

That is, the second symbol now represented a group of ten of the group to the right: the rightmost, the number of trees; to its left, the number of tens of trees; to its left, the number of tens of tens (hundreds); to its left, the number of tens of hundreds (thousands); etc.
Pepper still had more trees.

Positional numeral systems

Niles and Pepper, when they represented a tree by $\top$, created a very simple numeral system: a way of representing quantities with symbols called numerals. Once they recognized the value of including multiple symbols ($\bigcirc$ and $|$, at first) and endowed the position of each numeral with significance, the system became a positional numeral system. The number of numerals used is called the system's base: two for the system with $\bigcirc$ and $|$ and ten for that with 0–9. Base-2 systems are called binary, and typically use the symbols 0 and 1 instead of $\bigcirc$ and $|$. Base-10 systems are those with ten numerals, the most common of which is called the Hindu-Arabic numeral system and uses the Arabic numerals 0-9.
Let's consider the meaning of the Arabic number 937. It means 9 hundreds, 3 tens, and 7 ones. A corresponding arithmetic representation is

$$9 \times 10^2 + 3 \times 10^1 + 7 \times 10^0.$$

Similarly, the binary number 1011 has the meaning 1 pair of pairs of pairs of ones, 0 pair of pairs of ones, 1 pair of ones, 1 one. Let's convert this binary representation to base-10:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11.$$

This highlights an important ambiguity: how can we tell in which numeral system 11 is written? We cannot, so we must either rely on context, explicitly state, or add subscripts, as in

$$1011_2 = 11_{10}. \qquad (2)$$

As a convention, we restrict interpretations of numeral system-denoting subscripts to base-10. Now we introduce nuanced versions of the above numeral systems.

Decimal numeral system

Representing non-integer numbers is done with a radix point, often ".". The decimal numeral system is the Hindu-Arabic system extended to include non-integer numbers. Digits (decimal numerals) increasingly right of the radix point (called a decimal point in the decimal system) represent tenths, hundredths, thousands, etc. For instance, the decimal 2.73 would be

$$2 \times 10^0 + 7 \times 10^{-1} + 3 \times 10^{-2}.$$

Hexadecimal numeral system

The hexadecimal numeral system extends the decimal system with an additional six numerals, borrowed from the beginning of the Latin alphabet, to have a total of sixteen numerals:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}.$$

As we will see, this base-16 system provides a convenient way to represent the contents of computer memory.
It is conventional to begin hexadecimal numbers with the prefix "0x" as in 0x9A78 0D38.

Converting to and from decimal

Converting from a base-b number x_b with digits $x_n x_{n-1} \cdots x_0 . x_{-1} x_{-2} \cdots x_{-m}$ to decimal is straightforward. Represent each numeral in base-10, then use the formula

$$x_{10} = \sum_{i=-m}^{n} x_i b^i. \tag{3}$$

For instance, if $x_2 = 1010.01$,

$$\begin{aligned} x_{10} &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \\ &= 10.25. \end{aligned}$$

Similarly, if $x_{16} = \mathrm{B8.F}$,

$$\begin{aligned} x_{10} &= 11 \times 16^1 + 8 \times 16^0 + 15 \times 16^{-1} \\ &= 184.9375. \end{aligned}$$

Converting from decimal into a base-b numeral system can be accomplished by the following procedure.

- For the integer part of the number, successively divide by the base b_{10}, represented in base-10. The remainder x_b, represented in base-b, of each step is the base-b numeral in that position, from right-to-left.
- For the decimal part of the number, successively multiplying by the base b_{10}. The overflow x_{-b} of 1_{10} and above, at each step, is the corresponding base-b numeral in that position, from left-to-right.

Note that division and multiplication in the conversion process are the usual base-10 versions. Technically, this process can be used for converting between other numeral systems, but it is not recommended due to our unfamiliarity with division and multiplication in these numeral systems.

Example 00.4 −1 **re: decimal to binary and hex**

1. Convert 14_{10} to binary.
2. Convert 14_{10} to hexadecimal.
3. Convert 421.73_{10} to binary.

1. The following table shows the division.

b/divisor	dividend/quotient	remainder
	14	
2	7	0
2	3	1
2	1	1
2	0	1

   Therefore, $14_{10} = 1110_2$.
2. There is no need to divide because $14_{10} \leqslant 15_{10}$, the number of hex numerals. Therefore, $14_{10} = \mathsf{E}_{16}$.
3. For the integer part, the following table shows the division.

b/divisor	dividend/quotient	remainder
	421	
2	210	1
2	105	0
2	52	1
2	26	0
2	13	0
2	6	1
2	3	0
2	1	1
2	0	1

   Therefore, $421_{10} = 110100101_2$. Now for the number right of the decimal point.

b/factor	factor/product	overflow
	0.73	
2	.46	1
2	.92	0
2	.84	1
2	.68	1
2	.36	1
2	.72	0
2	.44	1
2	.88	0
2	.76	1
2	.52	1
2	.04	1
2	.08	0
2	.16	0
2	.32	0
2	.64	0
2	.28	1
2	.56	0
2	.12	1
2	.24	0
2	.48	0
2	.96	0
2	.92	

This last row is identical to the second row. Therefore, an infinite loop will occur and $421.73_{10} = 110100101.1\overline{01110101110000101000}_2$. This has profound implications: an exact decimal value of 421.73 must be rounded to be stored as binary. This introduces rounding error even when it might appear we know the number, exactly. (Note that this is for floating point representations of the decimal number, which is common. Alternatively, five integers could represent the decimal, exactly.) See Lecture 01.1 for exactly this (BCD).

Converting between hex and binary

Binary numbers can be easily converted hexadecimal and vice-versa. In Lecture 01.1 these conversions will be motivated. There are $2^4 = 16$ unique four-numeral binary numbers and 16 hex characters (which is no coincidence). This allows us to write each grouping of four binary numerals, called a nibble, as a single hex character. It is often easiest to convert each nibble to base-10, then (trivially) to hex. For instance, 1101_2 is

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13_{10}.$$

The thirteenth hex numeral is D.
Similarly, one can convert a hex numeral to a nibble by converting it first to decimal, then to binary.

Signed binary numeral system

The signed binary numeral system, often called the two's complement numeral system, is used to represent both positive and negative numbers in binary form. When encountering a signed binary number, first consider the leftmost numeral: if it is 0, the number it represents is positive or zero and the usual binary-to-decimal conversion holds; if it is 1, the number it represents is negative and must undergo the two's complement operation before the usual binary-to-decimal conversion holds for its negation.
The two's complement operation can be performed by flipping all the bits ($0 \to 1$ and $1 \to 0$) and adding 1. For instance,

$$1101\,0110 \xrightarrow{\text{flip bits}} 0010\,1001 \xrightarrow{\text{add one}} 0010\,1010.$$

This result can be converted to decimal in the usual way: $0010\,1010_2 = 42_{10}$. Therefore $1101\,0110_2 = -42_{10}$.

Of course, this means that an n-numeral in two's complement binary stores not (as would unsigned binary)

$$0, 1, \cdots 2^n - 1$$

but

$$-2^{n-1} + 1, -2^{n-1} + 2, \cdots - 1, 0, 1 \cdots 2^{n-1}.$$

In both unsigned and (two's complement) signed binary, however, a total of 2^n numbers are represented by n binary numerals.

00.5 Binary and hexadecimal arithmetic

In order to perform arithmetic operations on binary and hexadecimal numbers, a straightforward method is to convert the numbers to decimal, operate arithmetically in the usual way, then convert the result back to binary or hex.
However, arithmetic with all numbers represented by positional numeral systems can be performed in a familiar manner. We demonstrate this, by example, with binary, but this method also applies for hexadecimal arithmetic.

Example 00.5 −1 **re: binary summation**

Sum 1110_2 and 1100_2.

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 0\,1110 \\ +\,0\,1100 \\ \hline 1\,1010 \end{array}$$

Example 00.5 −2 **re: binary subtraction**

Subtract 1010_2 from 1100_2.

$$\begin{array}{r} 1\overset{0}{\cancel{1}}00 \\ -\,1010 \\ \hline 0010 \end{array}$$

Example 00.5 −3 **re: binary multiplication**

Multiply 1100_2 and 1010_2.

$$\begin{array}{r} 1100 \\ \times\,1010 \\ \hline 0000 \\ 1\,100 \\ 00\,00 \\ 110\,0 \\ \hline 111\,1000 \end{array}$$

00.6 Exploring C–building a sandbox

When beginning with any programming language, it helps to have a sort of "sandbox" in which one can play. If one follows the instructions in Resource 7 to set up the same environment used in the lab to compile for the myRIO, one can develop C programs in Eclipse at home. Even without connecting to a myRIO, one can compile (Eclipse calls this build) a myRIO program to check for syntax and other errors.

Taking it one step further, one can install another C compiler made for the development computer. This compiler may differ in certain aspects, but typically these differences are insignificant.

A good compiler to use for this purpose is the free GNU C compiler GCC. It is available on most platforms.

It is important to note that the compiler used in the class is provided by C/C++ Development Tools for NI Linux Real-Time. It is based on GCC, but may have different functionality.

Installing and using GCC on Windows

Download minGW from this link. Run the installer and include the GUI interface. Open the interface and, under the `Basic Setup` tab, check the boxes for `mingw32-base` and `mingw-gcc-g++`. Then select menu item Installation > Update Catalog.

Probably, there is no need to, but if the following step fails, try adding the minGW installation directory's `bin` to the system `PATH` environment variable.

Restart Eclipse. Select menu item File > New > C Project. Select from the left Project Type menu Executable > Hello World ANSI C Project. From the right Toolchains menu choose minGW GCC. Name the project (say)

`my_project` and select Finish.
In the C/C++ Perspective, under the `Project Explorer`, select
`my_project`. Build it by selecting menu item Project › Build Project. Run it by selecting menu item Run › Run As › 1 Local C/C++ Application.
If everything is working, you should see the "hello world" message display in the console. You can now edit the project, build, and run at will!
Of course, device driver functions like `fgets_keypad` still won't work in this environment because our system isn't connected to this hardware. However, many analogous functions can be substituted, like `fgets`.

Installing and using GCC on macOS

For those using macOS, the following instructions will help installing and using GCC. The following instructions assume you are using a terminal (e.g. Terminal.app) to execute the commands.
First, install the package manager Homebrew. Homebrew is great for getting and maintaining other software, too! It will install GCC with the following command.

```
brew install gcc
```

Now, just write a C program! Let's say you have `hi.c` in the current directory with contents as follows.

```
#include "stdio.h"

int main()
{
  printf("Hello World\n");
}
```

Compile this with GCC using the following command.

```
gcc -o hi hi.c
```

This compiles `hi.c` to the output executable file `hi` in the working directory. Now, try it out!

```
./hi # => prints Hello World to terminal
```

Alternatively, the Eclipse IDE can be used to write and debug GCC-compiled programs. The configuration is analogous to that for a Windows machine, described above.

00.exe Exercises for Chapter 00

Consider the binary numbers of Fig. exe.1.

```
0100 1011
1001 1000
0001 0001
1111 0110
```

Figure exe.1: four 8-bit binary numbers.

Unsigned binary

In Exercise 00., interpret these values as unsigned binary numbers.

Exercise 00.1

Convert the binary numbers of Fig. exe.1 to octal.

Exercise 00.2

Convert the binary numbers of Fig. exe.1 to hexadecimal.

Exercise 00.3

Convert the binary numbers of Fig. exe.1 to decimal.

Exercise 00.4

As a check on your calculations, convert the octal numbers from Exercise 00. to decimal.

Exercise 00.5

As a check on your calculations, convert the hexadecimal numbers from Exercise 00. to decimal.

Exercise 00.6

As a check on your calculations, convert the decimal numbers from Exercise 00. to octal.

Exercise 00.7

As a check on your calculations, convert the decimal numbers from Exercise 00. to hexadecimal.

Exercise 00.8

Add the first and second binary numbers of Fig. exe.1. Convert the sum to decimal, and compare to the sum of the decimal numbers obtained in Exercise 00..

Exercise 00.9

Add the third and fourth binary numbers of Fig. exe.1. Convert the sum to decimal, and compare to the sum of the decimal numbers obtained in Exercise 00..

Signed binary

In Exercise 00. interpret the same four bytes from Fig. exe.1 as signed binary numbers expressed in 8-bit, two's complement.

Exercise 00.10

Determine the decimal equivalents of each (signed) binary number of Fig. exe.1.

Exercise 00.11

Add the first and second (signed) binary numbers of Fig. exe.1. Convert the sum to decimal, and compare to the sum of the decimal numbers obtained in Exercise 00..

Exercise 00.12

Add the third and fourth (signed) binary numbers of Fig. exe.1. Convert the sum to decimal, and compare to the sum of the decimal numbers obtained in Exercise 00..

00.L Lab Exercise: Getting started

Objective

The objective of this exercise is to get acquainted with the following.

1. The Eclipse IDE and debugger.
2. Editing, building, loading, and running a program on the target computer.
3. Setting break points and single-stepping through a running program.
4. Displaying register and memory contents.

Pre-laboratory preparation

Read Resource 2 and Resource 7, following the instructions on your personal computer, preferably a laptop your can bring to lab. Pay particular attention to procedures for editing, building, and debugging. The following laboratory procedure is a tutorial that will allow you to become familiar with the Eclipse IDE.

Laboratory procedure

Perform this procedure in the lab, either on your laptop or a lab computer, connected to a myRIO. Launch Eclipse, Open the `main.c` file of your myLab0 project. Edit the file to include the C code in Figure L.1. Substitute your name for `<your name>` (12 characters max). Note the use of <tab>s and indenting.
Look carefully at this program. The `main` program loops four times, calling `sumsq` each time. What values do you expect in the `x` array after the program has executed?
Use the build command to compile and link your program. Errors and Warnings are shown in the Console pane. Correct any errors and re-build.

Run→Run Configurations the program. Select your `myLab0` project. Click Run. The results will be printed on the LCD display.

Using the debugger

In the following, you may find it useful to refer to the outline of important debugger commands in the Eclipse IDE for myRIO Notes. No program may be running on the target myRIO when you start the debugger.
Select Run→Debug Configurations. Select your `myLab0` project. Click Debug. The Debug perspective opens, showing the source code for the function `main`.
At this point, the execution has been suspended at the line highlighted in the source code.
Notice that the values of the program variables are displayed in the Variables pane anytime that execution is suspended. What are the current values of `i` and the array `x`? Notice also that the values of the processor registers are shown in the Registers pane.

```
/* Lab #0 - <your name> */

/* includes */
#include "stdio.h"
#include "MyRio.h"
#include "me477.h"

/* prototypes */
int  sumsq(int x);    /* sum of squares */

/* definitions */
#define   N   4       /* number of loops */

int main(int argc, char **argv)  {
    NiFpga_Status status;
    static    int    x[10];  /* total */
    static    int    i;      /* index */

    status = MyRio_Open(); /* Open  NiFpga.*/
    if (MyRio_IsNotSuccess(status)) return status;

    printf_lcd("\fHello, <your name>\n\n");
    for (i=0; i<N; i++) {
        x[i] = sumsq(i);
        printf_lcd("%d, ",x[i]);
    }
    status = MyRio_Close(); /* Close NiFpga. */
    return status;
}
int  sumsq(int x) {
    static int    y=4;

    y = y + x*x;
    return y;
}
```

Figure L.1: C code for Lab 00

Executing the Program—You are about to execute your program from the debugger. Use the ▶ icon to resume execution of your program. The only indication that your program has executed will be that your name and the values of the x-array should be displayed on the LCD display attached to the myRIO. Are the results consistent with your understanding of the program? Explain.
Running to a Breakpoint—A "breakpoint" is an address in memory where we would like the processor to stop while we examine or modify the state of the myRIO and/or memory. The target processor runs continuously unless it is stopped at a breakpoint.
Now let's re-execute the program until the execution arrives at a specified breakpoint. Start the debugger again from Debug Configurations....
Suppose that we want to continue execution (from the beginning) and determine the values of x and i just before the first C statement inside the "`for`" loop executes for the first time. Set a breakpoint at the "`x[i]=sumsq(i);`" statement by double-clicking on the marker bar next to that source code line.
A new breakpoint marker appears on the marker bar, directly to the left of the line where

you added the breakpoint. Also, the new breakpoint appears in the Breakpoints pane.
Run to the breakpoint using the icon.
The text window should now show execution suspended at that line. The Variables pane should now display the new values of `x` and `i`. The window marks in yellow values that have changed. Are they what you expected?
Finally, (assuming that execution has stopped at the first "`for`" loop iteration), cause the debugger to execute the loop one more time using the icon. The values of `x` and `i` should be updated in the Variables pane. Are they what you expected? Try it again. …And again. …And again! Watch the progress of the program on the LCD display.
Single-Stepping—The debugger can also step through the execution of the program in three ways:

1. "Step Over" Execute the current line, including any routines, and proceed to the next statement.
2. "Step Into" Execute the current line, following execution inside a routine.
3. "Step Return" Execute to the end of the current routine, then follow execution to the routine's caller.

"Step Over" single steps to the next sequential C statement, but executes through functions, and out of branches and loops before pausing. For example, if the next line of code is a call to a function, pressing will cause the entire function to be executed and debugger will pause at the line of code following the function call.
To begin the stepping process the target must be suspended.
Terminate execution 🟥, and then restart the debugging. Execution will be suspended at the beginning of the program.

Now, single step from this point using 'Step Over" repeatedly. Notice that step corresponds to a single line of C code. Notice also that the current values in the Variables pane change as you step. Watch the progress of the program on the LCD display. Eventually, execution exits through the `return` statement. Restart the debugging again. This time run to your breakpoint at "`x[i]=sumsq(i);`". Now, use "Step Into" to follow the execution into `sumsq`. Continue stepping using until execution exits back to `main`.
Quitting—You may terminate the debugging at anytime using terminate ■.
Feel free to repeat these procedures and to try other commands.

Resource R1 High−level embedded system

The Embedded Computing Laboratory (ECL) at Saint Martin's University is a space dedicated to teaching embedded computing in electromechanical systems. It is hosted by the Robotics Laboratory and developed in collaboration with Prof. Joe Garbini of the Department of Mechanical Engineering at the University of Washington (UW), to whom credit for much of the design is owed.

The following description is of the apparatuses at ECL, which are similar to those at the UW. The primary differences are that each lab has a different set of motors and the UW uses a custom analog amplifier to drive the motor whereas the ECL uses pulse-width modulation.

The developers of the following content distribute it in the hopes that others will find it educational and perhaps useful as a template for similar laboratories. Furthermore, we hope students will be able to reference it when they want to design their own embedded systems.

ECL has four identical systems for student use. Each system consists of four subsystems:

1. an embedded computer and development environment subsystem consisting of a National Instruments myRIO microcontroller, a personal computer, and the Eclipse IDE;
2. a user interface hardware subsystem consisting of a keypad, LCD display, and associated circuit boards;
3. a motor driver subsystem consisting of a dc power supply and a pulse-width modulation motor driver circuit board and
4. a motor and mechanical apparatus subsystem consisting of a flywheel supported by bearings and coupled to the shaft of a dc servomotor (with encoder for

feedback).

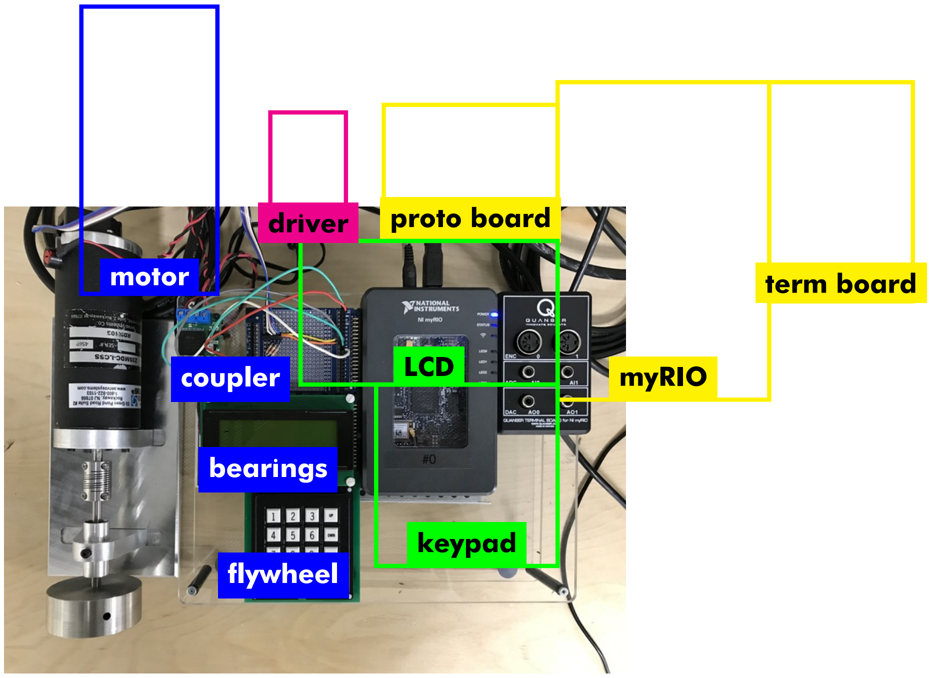


Figure 00.2: top view of most of the ECL apparatus.

Each of these is described in detail in the following sections. Together, they allow a student to program the microcontroller (in the C programming language) to instantiate completely embedded control of the motor speed and position, which are set by the user through the keypad and LCD display.

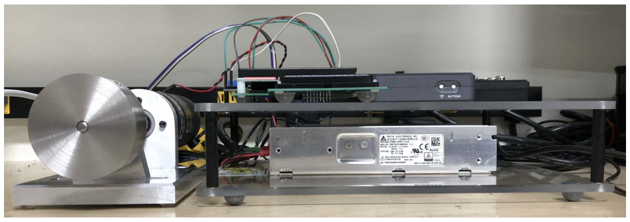


Figure 00.3: front view of most of the ECL apparatus.

Resource R2 Embedded computer and development environment subsystem

The development system is a powerful and convenient tool for embedded computing applications. As shown below, the development system consists of a personal computer, connected via a USB cable to target computer.

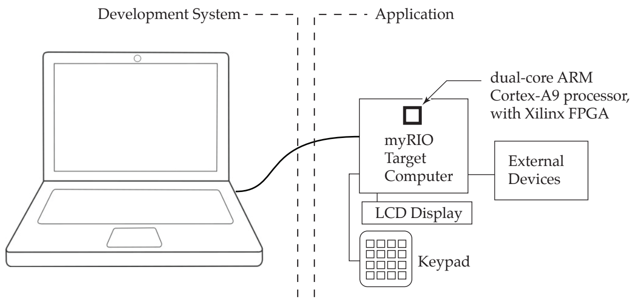


Figure 00.4:

During the development of an embedded computing application, the development system communicates with the real-time Linux operating system of the myRIO target computer. The development environment includes an integrated set of hardware and software tools that help to debug a microcomputer design by allowing you to watch your program execute, as well as to stop it and inspect system variables. As you will see, it allows you to monitor and control the target computer, without interfering with its timing.
Once hardware and software development is completed, the development system is disconnected from the target system. In the final application, the target program resides in ROM on the target computer.

Getting Started with CDT

Eclipse is an integrated development environment (IDE). We will use Eclipse through

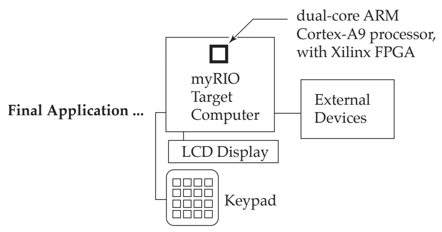


Figure 00.5:

its C Development Tool (CDT) to create, edit, build, deploy, and debug C language projects for the myRIO target computer. Within Eclipse all of your projects are organized into a single workspace on your computer. Each project, along with all of its necessary resources, are stored in a named project folder.
The outline below describes the basic functions of the IDE in preparing a C program for subsequent loading and execution on the myRIO remote system. Additional features are described in the Help menu.

Box 00.1 is CDT set up?

If the development PC has not yet been set up on the development computer, follow the procedure of Resource 7 to do so, before continuing.

Begin by starting the Eclipse IDE application.

C/C++ Perspective

Enter the C/C++ Perspective by selecting that button in the upper right.

The Project The ME 477 C Support for myRIO archive that you imported into your Eclipse workspace when you set up the CDT contains a template project for each of the nine laboratory exercises this quarter. They are listed in the right pane of the C/C++ perspective. Open a project by double clicking on its folder.

Each C program consists of a collection of functions, one of which must be called `main{}`, and is executed first. For large projects, additional functions are often in separate files. However, the organization of the assignments in this class is such that all of the functions for a single assignment can be conveniently stored in `main.c` along with `main{}`.

Run and Debug Configurations Among other things, Run and Debug Configurations specify how the project will be stored on the remote target. Configurations for all ME 477 laboratory exercises were loaded into your workspace in steps 4 and 5 of Part 1 of the C Development Tool Setup documentation—see Resource 7.

Building the Project Building the project consists of compiling your C source code into object modules, and linking them with other resources. Many coding errors can be found during the building process. Since building does not require that the development system be connected to the target, time spent in the lab is minimized. Before building, save any edit changes in the source code (`ctrl-s`). Either right click the project and use Build Project, or select and use Build Project from the Project pull down menu. Errors and warnings are displayed in the console menu in the bottom pane.

During each build, the CDT automatically re-compiles any file that has been edited (and saved). The build operation creates an output file in project's Debug folder.

Running the Project The project must build without errors before it can be run. The first time a project is run, pull down the Run menu and select Run

Configurations…. In the Run Configurations window, select the Run Configuration of your project. Then click Run.

The first run after a connection, you may be asked to login. Use User ID: `admin` and Password: `me477`.

Recently run projects may be conveniently run from the pull down menu under the run icon ▶▾.

A project will not run if a project is already running.

Barring execution problems, the project runs until `main{}` terminates.

Debug Perspective

Enter the Debug Perspective by selecting that button in the upper right. The Debug perspective lets you manage the debugging or running of a program. You can control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.

Debugging the Project The project must build without errors before it can be debugged. The first time a project is debugged, pull down the Run menu and select Debug Configurations…. In the Debug Configurations window, select the configuration of your project. Then click Debug.

After the first debug, the project may be conveniently selected for debugging by pulling down menu under the debug icon ▾.

A project may not be debugged if a project is already running.

Breakpoints A breakpoint suspends the execution of a program at the location

where the breakpoint is set. To set a line breakpoint, right-click in the marker bar area on the left side of an editor beside the line where you want the program to be suspended, then choose Toggle Breakpoint. You can also double-click on the marker bar next to the source code line. A new breakpoint marker appears on the marker bar, directly to the left of the line where you added the breakpoint. Also, the new breakpoint appears in the Breakpoints view list.
Once set, a breakpoint can be enabled and disabled by right-clicking on its icon or by right-clicking on its description in the Breakpoints view.

- When a breakpoint is enabled, it causes the program to suspend whenever it is hit. Enabled breakpoints are indicated with a blue enabled breakpoint circle.
- Enabled breakpoints that are successfully installed are indicated with a checkmark overlay.
- When a breakpoint is disabled, it will not affect the execution of the program. Disabled breakpoints are indicated with a white disabled breakpoint circle.

Debug view toolbar commands

The Debug perspective also drives the C/C++ Editor. As you step through your program, the C/C++ Editor highlights the location of the execution pointer.

Resume Select the Resume command to resume execution of the currently suspended debug target.

Suspend Select the Suspend command to halt execution of the currently selected thread in a debug target.

Terminate Ends the selected debug session and/or process. The impact of this action depends on the type of the item selected in the Debug view.

Step Over Select to execute the current line, including any routines, and proceed to the next statement.

Step Into Select to execute the current line, following execution inside a routine.

Step Return Select to continue execution to the end of the current routine, then follow execution to the routine's caller.

Debug information

Variables You can view information about the variables in a selected stack frame in the Variables view. When execution stops, the changed values are by default highlighted in red. Like the other debug-related views, the Variables view does not refresh as you run your executable. A refresh occurs when execution stops.

Expressions An expression is a snippet of code that can be evaluated to produce a result. The context for an expression depends on the particular debug model. Some expressions may need to be evaluated at a specific location in the program so that the variables can be referenced. You can view information about expressions in the Expressions view.

Registers You can view information about the registers in a selected stack frame. Values that have changed are highlighted in the Registers view when the program stops.

Memory You can inspect and change memory.

Disassembly You can view disassembled code mixed with source information.

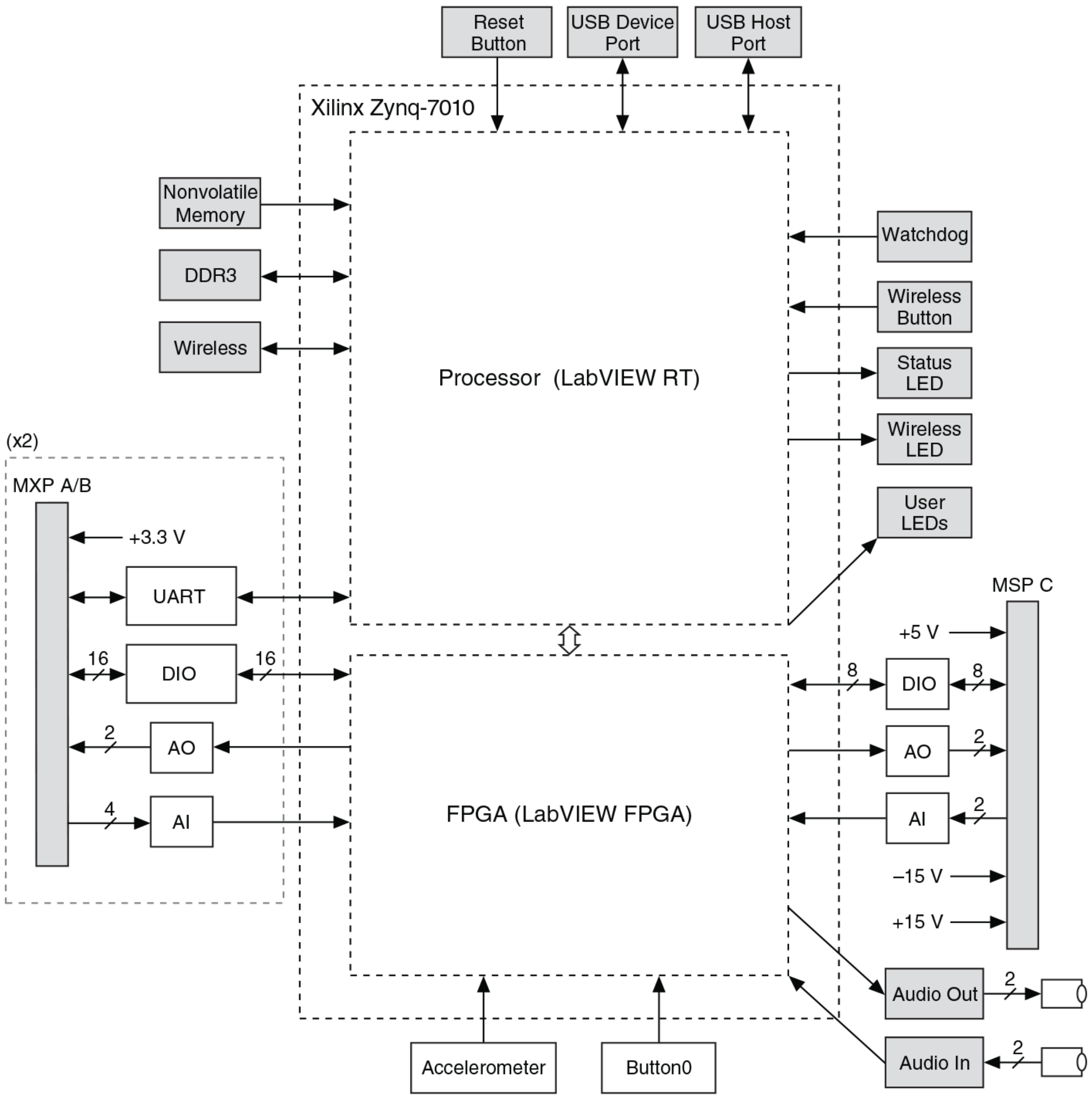


Figure 00.6: myRIO-1900 Hardware Block Diagram (source: Instruments (2013))

The system on a chip

The NI myRIO is centered around a Xilinx Z-7010 system on a chip (SoC): a dual-core Coretex A-9 CPU, memory, I/O inerfaces, and an Artix-7 fully programmable gate array (FPGA). The Z-7010 datasheet[3] is available here. These are powerful SoCs. The Coretex A-9 CPUs have 667 MHz clocks, have single- and double-precision vector float point units, and include NEON extensions (Xilinx, 2017). These processors use the ARMv7-A instruction set architecture (ISA) (ARM, 2012, 2014). The Coretex-A9 Reference Manual and Programmer's Guide are available here.

3. Xilinx, 2017.

Resource R3 User interface hardware subsystem

Resource R4 Motor driver subsystem

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia.

Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa. Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Resource R5 Motor and mechanical apparatus subsystem

Motor

Mechanical apparatus

The motor hanger, shaft, shaft hanger, ball bearings, and retainer rings are best purchased from a single mechanical supplier (for tolerance matching); we have chosen the following WM Berg parts (catalog pages linked):

1. motor hanger (blank): BC7-1C (needs holes drilled),
2. shaft (1/4 in diameter, 4 in length): s4-40,
3. shaft hanger (supports shaft through bearings): BC17-12C,
4. ball bearings (2): B1-9,
5. retainer rings (2): Q4-62, and
6. split cylinder shaft coupler: CO41S-2.

The split cylinder shaft coupler is a flexible coupler that helps with inevitable shaft misalignment.
The flywheel is the only custom-machined mechanical part. It is 304 stainless steel, one inch thick and 2.5 in diameter with a 0.25 in hole in the center for the shaft.
The Association of Electrical Equipment and Medical Imaging Manufacturers NEMA defines standards that many motor manufacturers use for mounting geometry.

Resource R6 Sourcing and costs

Resource R7 Setting up the C Development Tool for myRIO

Box 00.2 setting up a lab PC or one's own laptop?

For configuring your own laptop, complete all the steps, below.
For configuring a lab PC, complete steps 4 and 5 of Part A, then complete all remaining parts (Part B – Part F).

Box 00.3 myRIO connected?

Parts A, B, and C can be performed without connecting your laptop to one of the lab myRIOs. For Parts D, E, and F, a myRIO connection is required.

Do Parts A, B, and C just once, in the order shown.

Part A: Setting up the software environment

Follow these instructions to set up the C Development Tool for myRIO. Clicking on hyperlinks opens the appropriate websites in your browser.

1. Download and install LabVIEW 2015 myRIO Toolkit. (2700 Mb)

   For Native Windows 8 or 10:
   Download myRIOToolkit2015. Mount disk image. Then run `setup.exe`.

   For Native Windows 7: Download from myRIOToolkit2015. You may need a means of mounting the `.iso` disk image. For example, use Virtual CloneDrive to mount the `.iso` disk image files as a virtual CD-ROM drive. Then run `setup.exe`.

For Virtual Windows 7, 8, or 10 under Parallels:

Download myRIOToolkit2015 under OS X. From Parallels, Devices > CD/DVD > Connect Image... and mount the disk image.
Then run `setup.exe`.

2. Install Java. Visit the Java website GetJava to download Java. (17 Mb) Use Internet Explorer, not Microsoft Edge.
3. Install the C/C++ Development Tools for NI Linux Real-Time 2014, Eclipse Edition. Visit this link Eclipse2014 to download and install Eclipse. (260 Mb)
4. Project templates have been prepared for each of the ME 477 laboratory exercises. Visit the ME 477 website Resources Page to download the `ME477 myRIO support 2018` archive. Remember where you put this archive, but do not unzip. (2 Mb)
5. Eclipse uses Launch Configurations to specify how the project will be deployed and run on the myRIO.
Visit the ME 477 website Resources Page to download the `ME 477 Launch Configurations` archive. Unzip into folder `LaunchConfig477` (40 kb). Remember where you put the folder.
6. Add the compiler path to the system environment variables.

   a. Visit the ME 477 website Resources Page.
   `64-bit compiler path` file, select and copy with ctrl + C the contents.
   b. In the Windows Control Panel, select System and Security > System > Advanced system settings to display the `System Properties` dialog box.
   c. Click Environment Variables to display the `Environment Variables` dialog box.

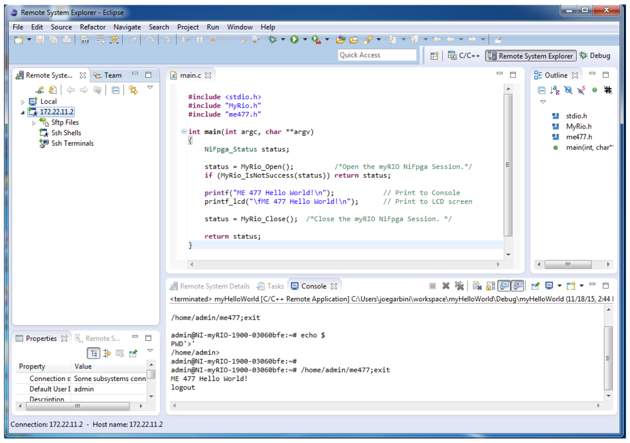
   d. Select `PATH` in the `User variables` group box and click Edit.
      If `PATH` does not exist, click New to create it.
   e. Click New and paste with ctrl + V the compiler path to the end of `Variable value` (separated by the `;` character). Be certain that there are no extra spaces in the path.

7. Click OK to close the dialog box and save changes.

Part B: Define a connection to the myRIO

Complete the following steps to define a connection in Eclipse from your laptop to the myRIO target.

1. Launch Eclipse, specify a workspace, and click OK to display the C/C++ perspective (default).
   Two other perspective views, Remote Systems Explorer and Debug, will also be useful. To make these available, select Window › Open Perspective › Other to display the `Open Perspective` dialog box.
   Then select `Remote Systems Explorer` and click OK to display the Remote Systems Explorer perspective. Repeat this process to display the Debug perspective. Buttons for all three perspectives should appear and can be used at any time to switch perspectives.
2. Open the Remote Systems Explorer perspective to display the `Remote Systems` pane at left.
3. Click the `Define a connection to remote system` icon to display the `New Connection` dialog box.
4. Select General › SSH Only.



```
#include <stdio.h>
#include "MyRio.h"
#include "me477.h"

int main(int argc, char **argv)
{
    NiFpga_Status status;

    status = MyRio_Open();          /*Open the myRIO NiFpga Session.*/
    if (MyRio_IsNotSuccess(status)) return status;

    printf("ME 477 Hello World!\n");            // Print to Console
    printf_lcd("\fME 477 Hello World!\n");      // Print to LCD screen

    status = MyRio_Close();  /*Close the myRIO NiFpga Session. */

    return status;
}
```

```
<terminated> myHelloWorld [C/C++ Remote Application] C:\Users\joegarbini\workspace\myHelloWorld\Debug\myHelloWorld (11/18/15, 2:44 

/home/admin/me477;exit

admin@NI-myRIO-1900-03060bfe:~# echo $
PWD'>'
/home/admin>
admin@NI-myRIO-1900-03060bfe:~#
admin@NI-myRIO-1900-03060bfe:~# /home/admin/me477;exit
ME 477 Hello World!
logout
```

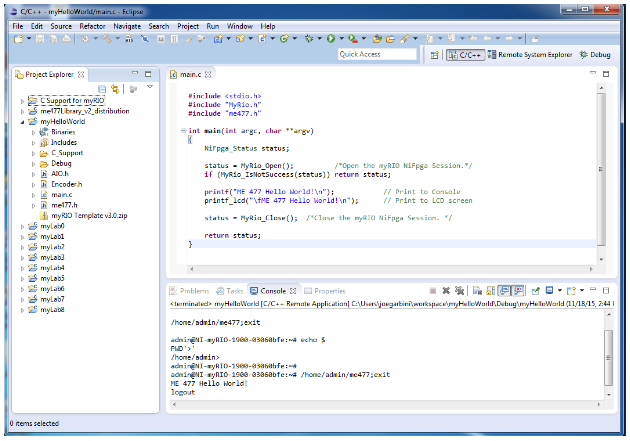
Figure 00.7: `Remote Systems Explorer` with myRIO connection successfully defined.

5. Enter the IP address `172.22.11.2` in the `Host name` textbox and click Finish. Your target displays in the `Remote Systems` tab in the `Remote System Explorer` pane, as shown in Figure 00.7.

Part C: Importing C Support and Launch Configurations

Complete the following steps to import C Support and Launch Configurations to Eclipse.

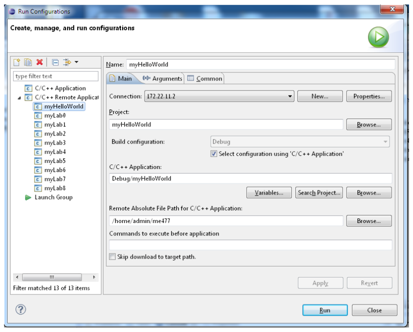
1. From the C/C++ perspective, select File › Import to display the `Import` dialog box.
2. Select General › Existing Projects into Workspace and click Next to display the `Import Projects` page.
3. Select Select archive file, click Browse and select the `ME 477 C Support for myRIO` zip file downloaded in step 4 of Part A.
4. Ensure that all items are checked and click Finish to import `ME 477 C Support for myRIO`. See Figure 00.8.
5. Build (compile) all projects with menu selection Project › Build All.



```
#include <stdio.h>
#include "MyRio.h"
#include "me477.h"

int main(int argc, char **argv)
{
    NiFpga_Status status;

    status = MyRio_Open();          /*Open the myRIO NiFpga Session.*/
    if (MyRio_IsNotSuccess(status)) return status;

    printf("ME 477 Hello World!\n");             // Print to Console
    printf_lcd("\fME 477 Hello World!\n");       // Print to LCD screen

    status = MyRio_Close();  /*Close the myRIO NiFpga Session. */

    return status;
}
```

```
<terminated> myHelloWorld [C/C++ Remote Application] C:\Users\joegarbini\workspace\myHelloWorld\Debug\myHelloWorld (11/18/15, 2:44

/home/admin/me477;exit

admin@NI-myRIO-1900-03060bfe:~# echo $
PWD'>'
/home/admin>
admin@NI-myRIO-1900-03060bfe:~#
admin@NI-myRIO-1900-03060bfe:~# /home/admin/me477;exit
ME 477 Hello World!
logout
```

Figure 00.8: ME 477 Project Templates.



Name: myHelloWorld
Connection: 172.22.11.2
Project: myHelloWorld
Build configuration: Debug
[✓] Select configuration using 'C/C++ Application'
C/C++ Application: Debug/myHelloWorld
Remote Absolute File Path for C/C++ Application: /home/admin/me477
Commands to execute before application:
[ ] Skip download to target path.

Figure 00.9: Launch Configurations.

6. Again, from the C/C++ perspective, select File ⟩ Import to display the `Import` dialog box.
7. Select menu item Run/Debug ⟩ Launch Configurations and click Next to display the `Import Launch Configurations` page.
8. Click Browse and select the `LaunchConfig477` folder that you downloaded in step 5 of Part A.
9. Ensure that all items are checked and click Finish. To check that the import of the Launch Configurations was successful, select menu item Run ⟩ Run Configurations... and compare the dialog with Figure 00.9.

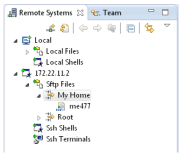


Figure 00.10: the `Remote Systems` tab should appear like this once a connection is established successfully.

Box 00.4 myRIO connected?

Your laptop must be connected through a USB cable to one of the myRIOs to perform Parts 4, 5, and 6. Each time you connect, a `myRIO USB Monitor` dialog box will appear indicating myRIO IP Address `172.22.11.2`. Always select Do Nothing.

Part D: Connect to the myRIO target

Complete the following steps to establish a connection between Eclipse and the myRIO target.

1. In the `Remote Systems` pane, right-click the target and select Connect from the shortcut menu to display the `Enter Password` dialog box.
2. Enter the user ID: `admin` and password: <UW: `me477` | SMU: leave blank> and click OK.
3. Click OK in the `Info` dialog box.
4. If the `Keyboard Interactive authentication` dialog box appears, leave the password blank, and click OK. As shown in Figure 00.10, green arrow appears on the target icon when the myRIO is connected.

Part E: Running the myHelloWorld project

In Parts 5 and 6 you will run and debug a project. Here, the `myHelloWorld` project is used

as example.
Eclipse uses a "Run Configuration" to specify how the project will be deployed and run on the myRIO. Run Configurations for ME 477 projects were downloaded in step 5 of Part A.
Complete the following steps to run the `myHelloWorld` example project.

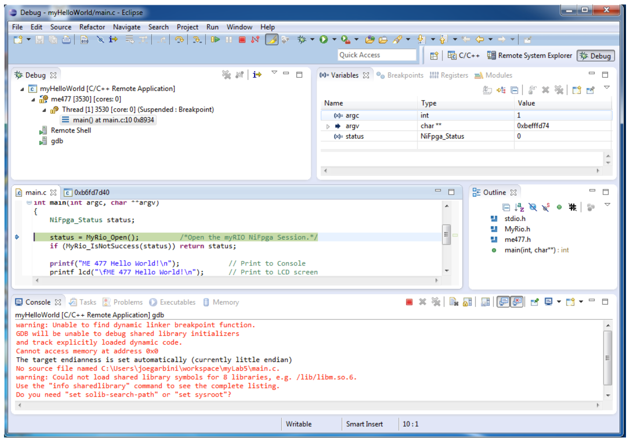
1. In Eclipse, switch to the C/C++ perspective.
2. You can view and edit the C source code by double clicking on the `myHelloWorld` project in the left pane, and then double clicking on `main.c`.
3. In the `Project Explorer` pane, right-click the `myHelloWorld` project, and select Build Project from the shortcut menu to build the project. Any build errors will be noted in the `Problems` pane.
4. Right-click the `myHelloWorld` project and select Run As › Run Configurations to display the `Run Configurations` dialog box.
5. Select the `myHelloWorld` project in the left pane.
6. Click Run. The project runs on the myRIO target. You can find the result in the `Console` pane, and on the LCD screen.

Part F: Debugging the myHelloWorld project

Similarly, Eclipse uses a "Debug Configuration" to specify how the program will be debugged on the myRIO. Once the Debug Configuration for a project is set up, debugging the program requires just a single click.
Complete the following steps to set up the Debug Configuration for the `myHelloWorld` project. These include building, deploying, and debugging the project.

1. In Eclipse, switch to the C/C++ perspective.

2. In the `Project Explorer` pane, right-click the `myHelloWorld` project and select Debug As › Debug Configurations to display the `Debug Configurations` dialog box.
3. Select the `myHelloWorld` project in the left pane.
4. Click Debug. The project runs on the myRIO target within the debugger. Some warnings may appear in the Console pane. Under normal circumstances, these warnings are not a problem. You can find the debug tools on the toolbar of Eclipse. There will be more about this in the first laboratory exercise.
5. For now, try setting a breakpoint at the `printf()` statement by double-clicking in the margin at left of that statement. A blue dot with a small checkmark ● should appear in the margin. The blue dot indicates that the breakpoint is enabled, and the checkmark indicates that the breakpoint is installed.
   If you `resume` (green arrow) from the beginning of the program, execution should pause at the breakpoint, as shown in Figure 00.11.



```
myHelloWorld [C/C++ Remote Application]
  me477 [3530] [cores: 0]
    Thread [1] 3530 [core: 0] (Suspended : Breakpoint)
      main() at main.c:10 0x8934
  Remote Shell
  gdb
```

Name	Type	Value
argc	int	1
argv	char **	0xbefffd74
status	NiFpga_Status	0

```
int main(int argc, char **argv)
{
    NiFpga_Status status;

    status = MyRio_Open();            /*Open the myRIO NiFpga Session.*/
    if (MyRio_IsNotSuccess(status)) return status;

    printf("ME 477 Hello World!\n");            // Print to Console
    printf_lcd("\fME 477 Hello World!\n");      // Print to LCD screen
```

```
myHelloWorld [C/C++ Remote Application] gdb
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
Cannot access memory at address 0x0
The target endianness is set automatically (currently little endian)
No source file named C:\Users\joegarbini\workspace\myLab5\main.c.
warning: Could not load shared library symbols for 8 libraries, e.g. /lib/libm.so.6.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
```

Figure 00.11: debugging and stopped at a breakpoint.

Resource R8 Suggested reading

The classic C programming language text Kernighan and Ritchie (1988), co-authored by Dennis Ritchie, who developed the language at AT&T Bell Laboratories between 1969 and 1973.

For computer hardware and software concepts, Patterson and Hennessy (2016) is a good introduction.

Part II

The User Interface

01

Computing principles, myRIO C programming, and high−level io drivers

01.1 Memory

Computer memory is a collection of bistable devices—so they can represent only, say 0 or a 1 in each bit—organized as bytes: collections of 8 binary digits or bits. There are $2^8 = 256$ unique bytes. In more modern systems, each byte (n.b. not bit) of memory has a unique address—an identifying code. An important aspect of the C programming language is that it can deal directly with these memory addresses, a relatively low-level functionality.

Memory is not content-specific. It can be used to represent numbers (integers, floating point, signed numbers, etc.), codes (character codes, numeral codes, etc.), and instructions. We must keep track of the meaning of its contents. For instance, a single bit could represent the state of the union: 1 could mean "covfefe" and 0, "dumpsterfire." A less exciting example with two bits representing four directions:

Things you can store in memory

Pure binary numbers

Non-negative integers of different magnitudes can be stored as pure binary in memory. Here is an example using one byte or two nibbles:

$$\begin{aligned}
\texttt{0000 0000}_2 &= 0_{10} \\
\texttt{0000 0001}_2 &= 1_{10} \\
&\vdots \\
\texttt{1111 1110}_2 &= 254_{10} \\
\texttt{1111 1111}_2 &= 255_{10}.
\end{aligned}$$

So the non-negative integers we can store in one byte are 0–255, of which there are $2^8 = 256$.

But we can use more than one byte to store a non-negative integer in pure binary. If multiple bytes are representing a number, the byte that occurs first (in terms of address) in memory is called the most significant byte (MSB), and the byte that occurs last is called the least significant byte (LSB). The MSB is usually represented as being to the left of the other bytes, and the LSB is typically represented as being to the right.

Here is a list of the total number of possible non-negative integers that can be stored in n bits (formula: 2^n) for typical values of n:

$$\begin{aligned} 2^8 &= 256 \\ 2^{16} &= 65,536 \\ 2^{24} &= 16,777,216 \\ 2^{32} &= 4,294,967,296. \end{aligned}$$

8-bit two's complement signed binary

How can a negative number be stored in memory? A single byte can store 256 unique pieces of information. For decimal numbers, this can range 0 to 255 or (say) -128 to 127.

A very convenient binary representation is called two's complement. A number x has two's complement in n bits of $(2^n - x)_2$; that is, the number of unique numbers representable minus the number, represented in binary. For instance, the 8-bit two's complement of `0110 1000` is[1]

$$\begin{array}{r} 1\,0000\,0000 \\ -\,0110\,1000 \\ \hline \end{array}$$

1. The first borrow might seem strange, but it's simply $10_2 - 01_2 = 2_{10} - 1_{10} = 1_{10} = 01_2$.

Below are listed some 8-bit two's complement decimal interpretations of binary numbers.

$$\begin{aligned}
\texttt{0000 0000}_2 &= 0 \\
\texttt{0000 0001}_2 &= 1 \\
&\vdots \\
\texttt{0111 1111}_2 &= 127 \\
\texttt{1000 0000}_2 &= -128 \\
\texttt{1000 0001}_2 &= -127 \\
&\vdots \\
\texttt{1111 1110}_2 &= -2 \\
\texttt{1111 1111}_2 &= -1
\end{aligned}$$

As if in Pac-Man, starting from the middle and exiting screen-right, only to appear screen-left—counting "up" loops one back down to negative numbers. Note that positive two's complements are the same as their pure binary counterparts.

There are two more-convenient ways to find the two's complement:

1. switching all bits ($\texttt{0} \mapsto \texttt{1}$ and $\texttt{1} \mapsto \texttt{0}$), then adding 1 or
2. starting from the right, copying all bits through the first 1 encountered, then switching all thereafter.

Both methods can be seen to always hold from the subtraction definition.

The two's complement of the two's complement of x is x; that is, it is its own inverse.

Example 01.1 −1 **re: two's complement**

Find the two's complement of 0000 0101.

If a binary number is interpreted as a two's complement binary number, it is negative if its most significant bit (msbit) is 1.

Binary coded decimal (BCD)

A binary coded decimal (BCD) represents each decimal digit with a nibble, so a series of nibbles can represent a decimal number. This leads to slightly less-dense storage, but is still useful for high-precision computation.

Example 01.1 –2   **re: BCD for rounding error**

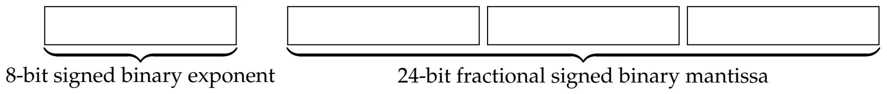
Recall that the number 421.73 had an infinitely long binary representation in Example 00.4 - 1. Represent this number in BCD. Let there be an "implied" decimal point, as some encodings define, between the third and fourth nibbles.

Floating point

Floating point numbers can represent very large or very small numbers with limited space. It is for computer memory what scientific notation is for a small piece of paper: that is, it represents a number as a mantissa[2] x and an exponent n; that is, $x2^n$, where we have used the conventional base of 2.

2. The mantissa is also called the significand or coefficient.

Consider the following illustration of a 32-bit (four-byte) floating point representation.


8-bit signed binary exponent
24-bit fractional signed binary mantissa

We would interpret this as, for instance,

$$\underbrace{\cdot 1011\cdots}_{\text{24-bit mantissa}} \times 2^{\underbrace{1011\ 0110}_{\text{8-bit exponent}}}. \tag{1}$$

Character codes

In addition to numbers, memory can store character codes: encoded alphabetic, special symbols, emojis, etc.
The most common character code is the American Standard Code for Information Interchange (ASCII). It's a 7-bit code, so there are 128 unique character codes.
It leaves the eighth bit of a byte, "bit seven," the parity bit, to be checked for transmission errors. It works as follows. Set (`1`) or reset (`0`) before transmission such that the total number of set (`1`) bits is either even or odd. If the system is using even parity, an even number of bits are set; or if it's using odd parity, an odd number of bits are set.
For instance, under odd parity, if the byte `1100 1101` is sent and the byte `1100 0101` is received, with its even number of set bits, the receiving system knows there has been a transmission error.

Instructions

Instructions are codes that direct the operation of a microprocessor. The myRIO has an ARM Cortex-A9 processor with 32-bit instructions.

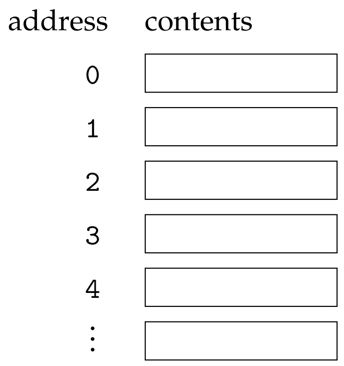
Example 01.1 –3 **re: memory interpretation**

Suppose the following is stored in a byte of memory: `1101 0101` or `D5`. How might this be interpreted?

Memory organization

In memory, bits are grouped into bytes of eight bits. Each byte is often considered as two nibbles, the contents of each represented by a hexadecimal numeral. For instance, a byte might be represented as follows.

Each byte is given a unique positive integer address distinct from its contents.



When storing a multi-byte number, we use the bigendian convention: the MSB is stored at the lower address. The littleendian convention stores the MSB at the higher address.

01.2 Processing

A CPU has an abstract model, called an instruction set architecture (ISA), that typically describes how the processor interacts with memory, input, output, and instructions. A popular architecture for personal computers is the x86 ISA. For mobile and embedded computers, however, the ARM ISA is ubiquitous.[3]

3. Another popular embedded architecture is the MIPS architecture.

The ARM ISA is a reduced instruction set computing architecture (RISC architecture), which means its instructions are less complex than those of a complex instruction set computing architecture (CISC architecture), such as x86. RISC architectures are often used in embedded computers.

The Embedded Computing Lab's embedded computers (on NI myRIO 1900 boards—see Resource 1) use the ARM architecture. Specifically, the system on a chip (SoC) Xilinx Z-7010's Coretex-A9 (dual) CPUs use the ARMv7-A ISA (see Resource 2).

Although the focus of this chapter is this architecture, many of the concepts apply more broadly, to CPUs with different ISAs.

01.3 A CPU programming model

A central processing unit (CPU) has three functions that are repeated endlessly:

1. fetch an instruction from memory,
2. translate the instruction, and
3. execute it.

Typically, the control unit of a CPU fetches instructions from memory and translates them. It then sends to the datapath of the CPU to be processed. It does this by means of registers, which are small, special purpose units of memory in the datapath.

Core ARM registers

A developer of an embedded system with a given CPU must understand it at the "application-level," which is distinguished from the "system-level" of the operating system. An application-level "view" of the ARM processor registers has thirteen general-purpose 32-bit registers named R0–R12 and three special-purpose registers named SP, LR, and PC (also called R13–R15) (ARM, 2014, p. A2-45).
The stack pointer SP (R13) register contains the memory address of, and therefore points to, the top of the active stack. A stack holds data, such as "automatic variables," temporarily. We'll talk more about stacks, later.
The return link LR (R14) register is used, for instance, to store the current memory address of the calling program during a subroutine call.
The program counter PC (R15) register contains the memory address of the current instruction plus eight (bytes)—that is, of two instructions-from-now.[4]
The general-purpose registers typically hold data, such as `int`s, `double`s, and `char`s.

4. For an interesting discussion of "why the offset," see this informative SO answer.

Other ARM registers

The 32-bit application status register (APSR) stores the program's last-executed instruction return status in flags:

- N: negative condition (e.g. two's complement negative MSbit),
- Z: zero condition (e.g. equal from comparison),
- C: carry condition (e.g. unsigned overflow from addition),
- V: overflow condition (e.g. signed overflow from addition), and
- Q: overflow or saturation condition (e.g. from DSP)

encoded as single bits. These flags can be tested by the next instruction for conditional execution. A nibble of the APSR stores the GE: greater-than or equal flag.
The execution state registers allows special instruction sets, such as Thumb, to be executed; contains special Thumb instructions; and sets the register endianness mapping (big-endian or little-endian).
The Xilinx Z-7010 Coretex-A9 has the optional ARMv7-A vector floating-point unit VFPv3 ISA extension, which enables high-performance and efficiency of floating-point arithmetic. The extension has its own, dedicated extension registers.

Types of instructions

Below are some examples of the types of instructions a CPU might encounter:

- load or store (to/from CPU registers),
- transfers (between registers),
- move (memory→memory),
- set/reset bits,
- shift/rotate,

- arithmetic (add, subtract, multiply, divide, negate),
- logic (ands, ors, etc.),
- conditional branches and jumps,
- unconditional branches and jumps, and
- subroutines.

Addressing modes

Addressing modes specify how the CPU is to calculate the memory address for a load or a store operation. For the ARMv7-A ISA, the address is composed of two parts: a base register value and an offset (ARM, 2014, p. A4-176). The base register can be any core ARM register. The offset must have one of the following three formats.

Immediate An unsigned number, it can be summed with (or subtracted from) the value of the base register.

Register A value from a core ARM register other than PC.

Scaled register A shifted value from a core ARM register other than PC summed with (or subtracted from) the value of the base register.

These lead to the following three addressing modes:

Offset The offset is summed with (or subtracted from) the base register, forming the memory address.

Pre−indexed Same as "Offset," followed by the new address is then assigned to the base register.

Post−indexed The memory address is the value of the base register. Then the base register is offset.

01.exe Exercises for Chapter 01

01.L Lab Exercise: Introduction to myRIO C programming and high−level io drivers

Objectives

In this exercise you will gain experience with:

1. C programming for myRIO.
2. The beginning of a device driver for the keypad/LCD.
3. On-line debugging techniques.

Introduction

In Lab Exercises 01, 02 and 03, we will write several functions that will allow a user to interact with the program through the keypad and LCD screen. Below is an outline of the functional dependencies and corresponding Lab Exercises. Functions provided by the `me477` library, core C, or the standard C library will be overwritten by those we write, which are shown in `green`.

- `double_in` (Lab 01) prompts LCD and returns keypad double ← this lab!
  - `fgets_keypad` (Lab 02) gets string from keypad
    - `getchar_keypad` (Lab 02) gets char from keypad
      - `getkey` (Lab 03) gets char from keypad
      - `putchar_lcd` (Lab 03) prints char to LCD
  - `printf_lcd` (Lab 01) prints string to LCD ← this lab!
    - `putchar_lcd` (Lab 03) prints char to LCD
    - `vsnprintf` (Lab 01) assigns to formatted string
  - `sscanf` (Lab 01) converts ASCII to binary
  - `strstr` (Lab 01) find string in string
  - `strpbrk` (Lab 01) find member in string

It is important to note that these functions are already available in `me477` library, so when we write our own version of a function, it supersedes the library version. This allows us to depend on the lower-level functions without writing them, first.

In this Lab Exercise, in addition to the `main` program, you will write `double_in` and `printf_lcd`. At this point, you are expected to have only an elementary knowledge of C, but you should become familiar with the procedures, such as debugging, that you will need in the future.

Pre-laboratory preparation

Complete the following and make sure your functions compile before running them while connected to the lab hardware.

Part #1 User input: writing the function *double_in*

Very often in an interaction between a computer and a user, a message or "prompt" is written on the LCD display and the user is expected to respond by entering an appropriate decimal number through the keypad. In this laboratory exercise you will write a C function, called `double_in`, to perform the complete keypad/LCD procedure.
This function will be used here, and in later exercises, to obtain numerical information through interaction with the terminal. It should execute the following steps each time it is called.

1. A user prompt (a string of ASCII characters) is written on Line-1 of the LCD display. A pointer to the string corresponding to this prompt is the only argument of the `double_in` function.
2. A floating point number is accepted from the keypad in response to the prompt. If an error occurs in the input string, the display is cleared, an error message is written on Line-2 of the display, and the prompt is issued again on the first line. The number is entered as a string of ASCII characters that may include the decimal digits 0 - 9, a decimal point, and a minus sign, and is terminated by ENTR.
3. The entered string is interpreted as a floating point number.
4. The floating point number (C data type `double`) is returned from `double_in` function to the calling program.

The prototype of the `double_in` function is

```
double  double_in(char *prompt);
```

For example, a call to `double_in` might be:

```
vel = double_in("Enter Velocity:  ");
```

The variable `vel` would be assigned the value entered.

The LCD interaction would look like:

```
Enter Velocity:  -50.75

```

Or, if an error occurs: (e.g. user enters: `-50..75`)

```
Enter Velocity: _
Bad Key.  Try Again.
```

Allow for four possible user errors:

Error Type	Error Message Displayed on Line-2
No digits are entered (e.g. ENTR only)	Short. Try Again.
↑ or ↓	Bad Key. Try Again.
"−" other than first character (e.g."−−")	Bad Key. Try Again.
".." double decimal point	Bad Key. Try Again.

Our goal here is that the user must enter a valid number before the `double_in` function can exit. Notice that the errors are detected in the string that the user enters.
Here is a possible strategy for `double_in`:
Begin by using the `printf_lcd` function (which we will also write in this exercise) to display the prompt on the LCD screen. Then,

1. Use `fgets_keypad` (get string) to obtain the string from the keypad. Its prototype is:

   ```
   char * fgets_keypad(char *buf, int buflen);
   ```

   When `fgets_keypad` is called, as in `fgets_keypad(string, 40)`, it assigns the characters from the keypad to the `string` variable, which should have been declared to be a character array, like `static char string[40]`. However, if ENTR is pressed, `fgets_keypad(string, 40)` returns `NULL` (instead of writing to `string`). So if you defined `flag = fgets_keypad(string, 40)`, if ENTR is pressed `flag == NULL` should be true.

2. Use `strpbrk` (string pointer break) to detect ↑ or ↓. Note: ↑ is returned by `fgets_keypad` as the ASCII character `[` and ↓ as `]`.
3. Use `strpbrk` to detect minus signs `-` beyond the first character.
4. Use `strstr` to detect double decimal points (i.e. `..`).
5. Use `sscanf` (scan formatted from string) to perform the ASCII-string-to-double conversion. Hint: because `sscanf` is converting to a variable of type `double`, you need to use the format `%lf` (long float).

Note: `printf_lcd` and `fgets_keypad` work like the standard C functions `printf` and `fgets`, and are linked to your program from `me477` library.
Write a main program that tests your `double_in` function by calling it twice from the `main` program, assigning each result to a different variable. Then, as a check, print the values of both variables on the console using `printf`. See Algorithm L.1 for `main` pseudocode and Algorithm L.2 for `double_in` pseudocode.

Part #2 Display on LCD: writing the function *`printf_lcd`*

Our second task is to write the `printf_lcd` function used by `double_in`. The C function

Algorithm L.1 `main` pseudocode

```
function main
    declare double variable vel for velocity
    open connection to myRIO and check for success
    call double_in and assign output to vel
    print vel to LCD with printf_lcd
    close myRIO connection and return its status
end function
```

Algorithm L.2 `double_in` pseudocode

```
function double_in(p)                        ▷ p is prompt pointer
    declare variables
    clear LCD display                        ▷ use printf_lcd
    c ← 1                                    ▷ initialize stop check
    while c == 1 do
        print p to LCD                       ▷ use printf_lcd
        f ← fgets_keypad(s,␣)                ▷ get string s and out flag f
        if f == NULL then
            print "Short. Try again." to LCD ▷ use printf_lcd
        else if s does not pass bad key checks then
            print "Bad key. Try again."      ▷ use printf_lcd
        else
            c ← 0                            ▷ set stop condition
            sscanf(s,"%lf",&v)               ▷ convert s to double and assign to v
        end if
    end while
    return v
end function
```

`printf` prints to the standard output device, in our case the Console pane of the Eclipse IDE. We want `printf_lcd` to operate exactly as `printf`, except that it will print to the LCD screen. Refer to your C text. To do this, we want `printf_lcd` to accept a format string with a variable number of arguments. Therefore, the prototype for `printf_lcd` is

```
int printf_lcd(const char *format, ...);
```

where `format` is a string specifying how to interpret the data, and the ellipsis (`...`) represents the variable list of arguments specifying data to print. The return value is an `int` equal to the number of characters written if successful or a negative value if an error occurred.
For example,

```
n = printf_lcd("\fa = %f, b = %f", a, b);
```

Here is a suggested strategy for `printf_lcd`:

- Use the C function `vsnprintf` to write the data to a C string.
- Then use the LCD driver function `putchar_lcd` to successively write each character in the string to the LCD display. Note: It is strongly suggested that you use an incremented pointer to access the string, rather than an array index. See Lec. 01.L for more guidance on `putchar_lcd`.

The C function `vsnprintf` writes formatted data from the variable argument list to a buffer (the string) of a specified size.
The tricky part is passing the variable argument list of `printf_lcd` to `vsnprintf`. Here is an example fragment of code. From your C text, study the data type `va_list`, and the C macros `va_start` and `va_end` to see how this works.

```
int printf_lcd(char *format, ...) {
    va_list args;
    va_start(args, format);
    n = vsnprintf(string, 80, format, args);
    va_end(args);
}
```

As usual, you must allocate storage for the C `string` of length 80.

The `main` program, the `double_in` function, and the `printf_lcd` function should all be in the same file: `main.c`. Be sure to *`#include`* the header files `me477.h`, `<stdio.h>`, `<stdarg.h>`, and `<string.h>` in the code.

Once you have defined `printf_lcd` within your `main.c`, your code will supersede the version in the `me477` library. See Algorithm L.3 for pseudocode for `printf_lcd`.

Algorithm L.3 `printf_lcd` pseudocode

```
function printf_lcd($f$, $v$)    ▷ f is string format, v is variable data to print
    declare variables
    start parse args with va_list, va_start
    $n \leftarrow$ vsnprintf($S$,80,$f$,args)    ▷ S is the string char length 80
    finish parse args with va_end
    if $n < 0$ then    ▷ test for conversion error
        return $n$
    end if
    initialize $s$    ▷ s points to start of S
    while dereferenced $s$ is not 0 do    ▷ check if S is done
        putchar_lcd(dereferenced $s$ with postfix increment)
    end while
    return $n$
end function
```

Laboratory procedure

Debug and test your C program. As necessary, use breakpoints and single-stepping to find errors.

Guidance

This section provides guidance on several aspects of the Lab Exercise, above.

Background on *putchar_lcd*

The C function `putchar_lcd` places the single character corresponding to its argument on the LCD screen. Its prototype is

```
int  putchar_lcd(int c);
```

where both the input parameter and the returned value are the character to be sent to the display. A character constant is an integer, written as one character within single quotes, such as `'x'`.
For example, calls to `putchar` might be:

```
ch = putchar('m');
putchar('\n');
```

To write both parts of your program you also need to know how the escape sequences used in the `putchar_lcd` function affect the LCD screen. This concerns the important matter of I/O (input/output), which we will consider in detail later. For now the following table explains the escape sequences:

Escape Sequence	Function
`\f`	Clear Display
`\b`	Move cursor left one space
`\v`	Move cursor to the start of line-1
`\n`	Move cursor to the start of the next line

Dissecting a C program

This lab requires the use of several aspects of the C programming language. In this section, some of that is outlined, but a C textbook such as Kernighan and Ritchie (1988) is required for sufficient understanding.

We begin by writing a simple C program that sums loop indices and proceed unpack its meaning.

```c
/* include libraries */
#include "stdio.h"

/* declare function prototypes */
int sum(int x); /* sum */

/* define external/global variables */
#define N 5 /* number of loops */

/* define functions */
int main(int argc, char *argv[]) {
    static int x[10]; /* total */
    static int i; /* index */
    for (i=0; i < N; i++) {
        x[i] = sum(i);
        printf("%d",x[i]);
        if (i < N-1) {
            printf(",");
        }
        else {
            printf(".");
        }
    }
    return 0;
}

int sum(int x) {
    static int y=0; /* initialize y */
    y = y + x;
    return y;
}
```

```
0,1,3,6,10.
```

C programs consist of variables and functions. Variables are defined via an assignment statement, the most common operator is = as when our program assigns the first value of the variable `i` in the expression `i = 0`, which could also be written `i=0`—spaces are added for clarity.

Our program has two functions: `main` and `sum`. Whenever a C program is executed, it begins with a function named `main`. Every program must have one. If we don't need to pass any arguments to our program, in its definition, the argument can be empty, as in:

```c
int main() { /* statements */ }
```

If we need to pass arguments—say, from the command-line—there is a specific method described in detail by Kernighan and Ritchie (1988, p. 114) that would have as its definiton:

```c
int main(int argc, char *argv[]) { /* statements */ }
```

For our program, we don't use the arguments, so either is valid.
The `int`s before the definitions of `main` and `sum` declare that these functions return data type `int` for integer. Although it is not strictly required in every instance, it is considered best practice to always precede a definition with its return data type.
Most C programs load external libraries with pre-compiled functions. The most popular libraries are from the C standard library. For i/o functions like the `printf` we use here to print to the console, the `stdio.h` header file must be *`#include`*d, as shown at the top of our program.
We'll include the header file `me477.h`, which includes compiled versions of the functions we'll be writing over the next few Lab Exercises.
Best practice is to declare prototypes for each function (we often skip `main`, which always has the same prototype), which, for our `sum`, looks like:

```c
int sum(int x);
```

Here we're declaring that `sum` is a function with a single integer argument, which we'll call `x` inside the function, that returns an integer to the calling function. These declarations should be before `main`. The function definition can occur either before or after `main`, but we adopt the convention of defining functions after `main`. External or global variables are those defined above `main`. These variables are defined once and can be accessed by every function that declares it with `extern`. A similar, but distinct object is the symbolic constant, defined by `#define`, as with `N` in our program. A difference is that symbolic constants need not be declared within a function. We conventionally capitalize symbolic constants.

Line 12 shows the declaration of variable `x` to be an array with `10` elements. This preallocates a block of memory for `x`. Most variables inside a function are automatic: they are not retained between function calls. However, often in embedded computing we will be using pointers to specific addresses in memory at which a variable can be found. The safest way to use pointers is to declare the variable to be `static`, as in Lines 12, 13, and 28. A very important consequence of this declaration is that the variable's value is retained between function calls. For instance, in `sum`, we initialize `y` to be a static integer (`0`), then add the argument `x` to it are return the sum, which has overwritten `y`. Each successive call, the old value of `y` is retained, so on the second call, third call, for which `x` is `2` and the old `y` is `1`, the returned `y` is `3`.

Line 14 is the beginning of a `for` loop. We highlight two syntactical nuances. First, there are the three flow control components in the statement `(`initialize`;`condition`;`increment`)`. The initialize statement is executed first and only once. The condition statement returns a boolean (actually just an `integer`) of `1` for true and `0` for false. If the condition is true, the statements between the following braces are executed. Afterwards, the increment statement is evaluated and the loop returns to evaluate the

condition ….
The second syntactical consideration is that the braces `{}` should enclose the looped block. Although a single statement need not be enclosed, multiple statements must, and therefore we adopt the convention of always enclosing loop statements in braces.
The `if`/`else` execution control keywords are straightforward and are not expanded upon, here.
Finally, `main`, like any function, should return to the calling function (for `main`, calling program) some value, which, for most functions, can be of any data type, but for `main` is a status code as an `int`eger. The `return` keyword defines the return status, in our program, simply `0`. Conventionally, this signifies to the calling program that our program has run successfully. Nonzero `main` return values are used to signify different error codes, which should be documented for your program.

Execution control

As we saw in the example above, C has the usual execution control statements, which include `while`, `for`, `if`, `else`, and `else if`. This Lab Exercise should familiarize you with several of these.

C data types

C has only a few core data types:

- `char`s are single byte characters;
- `int`s are integers, the size of which is machine-dependent;
- `float`s are single-precision floating-point numbers, the size of which is machine-dependent; and
- `double`s are double-precision floating-point numbers, the size of which is machine-dependent.

Typically, a `float` is 32-bit and a `double` is 64-bit. There are also qualifiers such as `short` and `long`, which compilers typically take to mean “fewer” bytes for the specified representation or “greater,” respectively. Arrays are just lists of values. When declaring an array, one specifies the data type of each element and the number of elements, as in `double x[10];`, which is an array of ten `double`s. Accessing element `n` of an array `x` is done with the syntax `x[n]`. It is important to note that the first index of an array is `0` in C.

Pointers

Pointers are a key concept in C. A pointer is variable that is assigned not a value, but a memory address. To get some variable `x`’s value address, one uses the address operator `&`, like `&x`. In order to assign this to a pointer variable, the variable must be declared as a pointer to a specific data type. For `x` an integer, a pointer to it can be declared with `int *p;` and assigned with `p = &x`. To access the value to which a pointer `p` points, the dereferencing operator `*` can be used, as in `*p`.
Consider the following example.

```c
#include "stdio.h"

int main() {
    static int x = 1;
    static int *p = &x;
    printf("%d\n",x);  /* value */
    printf("%p\n",&x); /* address */
    printf("%p\n",p);  /* pointer */
    printf("%d\n",*p); /* deref'd pointer */
    return 0;
}
```

```
1
0x100693018
0x100693018
1
```

An array variable, say an integer array of length 10 declared by `int z[10];`, is just a pointer to the first value in the array. An array name is a constant pointer, so it cannot be reassigned (e.g. if `p_a` is an array, this is invalid: `p_b = p_a;`).

Cast operator

A cast operator on an expression to type type is `(type)` expression. It represents the expression in the new type in accordance with certain rules. It does not affect any definitions in the original expression; rather, it returns a new expression. Suppose you have the following:

```c
static int a; // 2-byte
static long int b; // 4-byte
b = 3;
a = (int) b; // cast and assign to a
```

The casting of a four-byte `long int` to a two-byte `int` means there is a potential for truncation because four bytes can represent more integers.
When casting to an `int` from a `float` or `double`, beware that truncation does not round in the usual sense: it simply drops the fractional part. It is preferable to use the function `round` provided by the standard library header file `math.h`.

Incrementing and decrementing

For `int x = 0`, instead of writing `x = x + 1` to increment `x`, we can write either `++x` or `x++`. The former is called a prefix operator and the latter postfix, both of which increment `x`, but they are interpreted differently in an expression:

- `++x` increments `x`, then uses it in the expression in which it appears (e.g. `n = ++x` assigns `1` to `x`, then `1` to `n`) and

- `x++` uses `x` in the expression in which it appears, then increments it (e.g. `n = x++` assigns `0` to `n`, then `1` to `x`).

The decrement operator `--` also has pre- and postfix versions, but subtracts one instead of adding.
The next example shows how pointers—not just `int`s—can be incremented. They can also be decremented. Incrementing a pointer moves it not to the next address, but to the next piece of data in memory, skipping the necessary number of bytes.

Operator precedence and associativity

See Lec. 02.2 for a table of operator precedence and associativity. The following example shows some interesting precedence and associativity interactions among operators `*` and `++` and parentheses `()`.

```c
#include "stdio.h"

int main() {
    static int x = 5;
    static int *p = &x;
    printf("(int) p   => %d\n",(int) p);
    printf("(int) p++ => %d\n",(int) p++);
    x = 5; p = &x;
    printf("(int) ++p => %d\n",(int) ++p);
    x = 5; p = &x;
    printf("++*p      => %d\n",++*p);
    x = 5; p = &x;
    printf("++(*p)    => %d\n",++(*p));
    x = 5; p = &x;
    printf("++*(p)    => %d\n",++*(p));
    x = 5; p = &x;
    printf("*p++      => %d\n",*p++);
    x = 5; p = &x;
    printf("(*p)++    => %d\n",(*p)++);
    x = 5; p = &x;
    printf("*(p)++    => %d\n",*(p)++);
    x = 5; p = &x;
    printf("*++p      => %d\n",*++p);
    x = 5; p = &x;
    printf("*(++p)    => %d\n",*(++p));
    return 0;
}
```

```
(int) p   => 81195032
(int) p++ => 81195032
(int) ++p => 81195036
++*p      => 6
++(*p)    => 6
++*(p)    => 6
*p++      => 5
(*p)++    => 5
*(p)++    => 5
*++p      => 0
*(++p)    => 0
```

Strings

Strings are arrays of `char`s, terminated by a `NULL` (which is a pointer that casts to `0`). For instance, the string `"HELLO"` could be represented in memory (with corresponding ASCII codes) as follows.

"H"
"E"
"L"
"L"
"O"
NULL

Function argument passing

All function arguments in C are passed “by value”: the function receives its arguments through temporary local variables called automatic variables (see Lec. 01.L for more about automatic and global variables). When it’s necessary to pass back an argument with a changed value, the caller can provide the function with the argument address via a pointer, and the function must access the value through the pointer. A potential alternative is a global `extern` variable.

Literal of a *`long`*

For the compiler to recognize a literal number as a `long`, it must have an `L` suffix. For instance, if `val` is a `long` variable and you want to compare it to 32767:

```
if(val > 32767L) { /* validated! */ }
```

NULL detection

The following program gives some insight into detecting a returned `NULL`.

```
#include "stdio.h"

int main() {
    printf("%p\n",NULL); /* print as pointer */
    printf("%d\n",(int) NULL); /* cast to int */
    if (NULL == 0) {
        printf("this works\n");
    }
    if (NULL == 0x0) {
        printf("this works, too!\n");
    }
    if (NULL == NULL) {
        printf("so does this!");
    }
    return 0;
}
```

```
0x0
0
this works
this works, too!
so does this!
```

Hex numbers—signed

In addition to the specifically C-related topics, above, the following is useful for the first assignment.
We can change the sign of a signed binary by taking the two's complement.
To put a negative hexadecimal number into a signed hexadecimal form, take the sixteen's complement. Steps:

- take fifteen's complement and
- add 1.

Example Lab 01–1 **re: Signed hexadecimal**

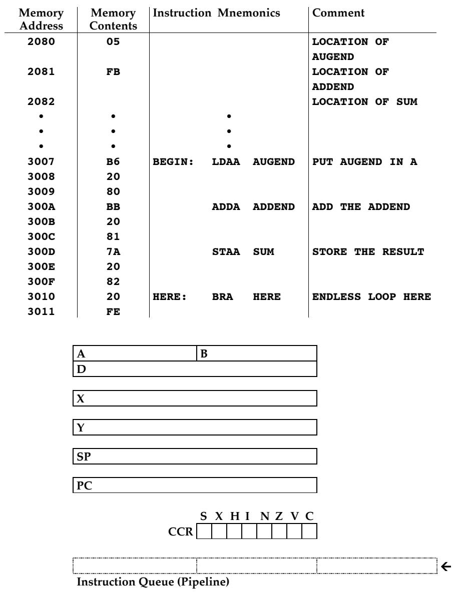
Convert 3A to -3A.

02

Exploring C and mid–level io

02.1 A paper computer

Consider the following graphic. It is an example of how a program running on a Motorola 68HC12 microcontroller might proceed at the memory/register/assembly language level. The HC12 registers are different than the ARM registers discussed in Lec. 01.3 , but there are some differences. For instance, the HC12's condition code register (CCR) is akin to the ARM application status register (APSR). Begin with the program counter (PC) at memory address 3007 and follow the corresonding instructions, annotating the registers as appropriate.



Memory Address	Memory Contents	Instruction Mnemonics	Comment
2080	05		LOCATION OF AUGEND
2081	FB		LOCATION OF ADDEND
2082			LOCATION OF SUM
•	•	•	
•	•	•	
•	•	•	
3007	B6	BEGIN: LDAA AUGEND	PUT AUGEND IN A
3008	20		
3009	80		
300A	BB	ADDA ADDEND	ADD THE ADDEND
300B	20		
300C	81		
300D	7A	STAA SUM	STORE THE RESULT
300E	20		
300F	82		
3010	20	HERE: BRA HERE	ENDLESS LOOP HERE
3011	FE		

Table 02.1: C operator precedence and associativity.

Operator	Description	Associativity
`()`	Parentheses (grouping)	left-to-right
`[]`	Brackets (array subscript)	
`.`	Member selection via object name	
`->`	Member selection via pointer	
`++ --`	Postfix increment/decrement (see Note 1)	
`++ --`	Prefix increment/decrement	right-to-left
`+ -`	Unary plus/minus	
`! ~`	Logical negation/bitwise complement	
`(type)`	Cast (change type)	
`*`	Dereference	
`&`	Address	
`sizeof`	Determine size in bytes	
`* / %`	Multiplication/division/modulus	left-to-right
`+ -`	Addition/subtraction	left-to-right
`<< >>`	Bitwise shift left, Bitwise shift right	left-to-right
`< <=`	Relational less than/less than or equal to	left-to-right
`> >=`	Relational greater than/greater than or equal to	
`== !=`	Relational is equal to/is not equal to	left-to-right
`&`	Bitwise AND	left-to-right
`^`	Bitwise exclusive OR	left-to-right
`|`	Bitwise inclusive OR	left-to-right
`&&`	Logical AND	left-to-right
`||`	Logical OR	left-to-right
`?:`	Ternary conditional	right-to-left
`=`	Assignment	right-to-left
`+= -=`	Addition/subtraction assignment	
`*= /=`	Multiplication/division assignment	
`%= &=`	Modulus/bitwise AND assignment	
`^= |=`	Bitwise exclusive/inclusive OR assignment	
`<<= >>=`	Bitwise shift left/right assignment	
`,`	Comma (separate expressions)	left-to-right

02.2 Exploring C–operator precedence and associativity

Table 02.1 lists all C operators in order of their precedence (highest to lowest). Operators within the same box have equal precedence. Note 1—Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement `y = x * z++;` the current value of `z` is used to evaluate the expression (i.e., `z++` evaluates to `z`) and `z` only incremented after all else is done.

Operator precedence

When an expression contains two or more operators, normal operator precedence rules are applied to determine the order of evaluation. If two operators have different levels of precedence, the operator with the highest precedence is evaluated first. For example, multiplication is of higher precedence than addition, so the expression `2+3*4` is evaluated as

```
3 * 4 // = 12
2 + 12 // = 14
```

The evaluation order can be explicitly controlled using parentheses; e.g., `(2+3)*4` is evaluated as

```
2 + 3 // = 5
5 * 4 // = 20
```

Operators in Table 02.1 are grouped from highest to lowest precedence.

Operator associativity

If two operators in an expression have the same precedence level, they are evaluated from left to right or right to left depending on their associativity. For example, addition's associativity is left-to-right, so the expression `2+3+4` is evaluated as `(2+3)+4`. In contrast, the assign operator's associativity is right-to-left; so the expression `x=y=z` is evaluated as `x=(y=z)`.

02.3 Exploring C–compile–time integral constants

Often, we want to define a symbol that has a single integral value—an integer—throughout our program. Fortunately, C lets us do that many ways. Unfortunately, it can be hard to choose among them.

The primary ways are `#define`s (macros), `enum`s (enumerations), and `const int`s. When choosing among them, our primary concerns are code readability, debuggability, and compile-time optimization.

Box 02.1 `static` is not constant

Several times thus far, including in the listing of Fig. L.1, we have declared `static int`s. These are variables not constants, as their name might suggest. Rather, they retain their value between function calls—but that value can be changed within the function (and the new value retained).

The last of these means a compiler (or preprocessor before the compiler) can replace each instance of the symbol with its constant value (since it never chances). There are subtle differences in how each compiler works, but most of the time all three of our options yield replaced compile-time constants. However, `#define`s are the best guarantee (because it actually happens before compilation, via preprocessing), `enum`s a close second, and `const int`s a respectable third.

In terms of debuggability, the rankings are probably best reversed; that is, in decreasing debuggability: `const int`s, `enum`s, and `#define`s. Macros (`#define`s) are most difficult because the compiler can't usually give useful error codes related to them (since the compiler typically knows nothing of them due to

preprocessing).
Readability is rather subjective, but `enum`s are typically considered strong in this regard, especially with its automatic enumeration of symbols.
A way to demonstrate this is to show the same example, written these three ways. Let's define an integral value to each day of the week, then write a script that prints a value.

```
#include <stdio.h>
enum day {
    sunday, monday, tuesday, wednesday,
    thursday, friday, saturday
};
enum day today = monday;
enum day checkout = friday;

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

```
#include <stdio.h>
#define sunday 0
#define monday 1
#define tuesday 2
#define wednesday 3
#define thursday 4
#define friday 5
#define saturday 6
#define today monday
#define checkout friday

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

```
#include <stdio.h>
const int sunday = 0;
const int monday = 1;
const int tuesday = 2;
const int wednesday = 3;
const int thursday = 4;
const int friday = 5;
const int saturday = 6;
const int today = monday;
const int checkout = friday;

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

Preference among these three options is hotly debated, but it seems `enum`s are the most readable and the "just right" option in terms of reliable compile-time integral constant replacement and debuggability.
It is important to remember that `#define`s can be used for much more than integer replacement: function-like macros, for instance, are very useful.

02.4 Exploring C–pointers

Assigning to a pointee

The function `fgets_keypad`, the source for which is shown in the introduction to Lab Exercise 02, was used in Lab Exercise 01. Recall that in `double_in` we supplied as arguments to `fgets_keypad` a character array (pointer) and its length. Instead of returning the string, the function wrote to the character array it was supplied—but remember: inside a C function arguments are assigned automatic variables. How does `fgets_keypad` assign to the array when it knows only a pointer to its first element? The secret sauce is to assign through a dereferenced pointer. Examine the source for `fgets_keypad` or consider the following example.

```c
#include <stdio.h>
void foo(int * p);

int main() {
    static int x = 0;
    static int * p = &x;
    printf("before: %d\n",*p);
    foo(p);
    printf("after: %d",*p);
    return 0;
}

void foo(int * p) {
    *p = 3;
}
```

```
before: 0
after: 3
```

Note that, while this sort of structure is rare among higher-level programming languages, it is quite common in C. For instance, `fgets` and `gets` have this same feature.

02.exe Exercises for Chapter 02

02.L Lab Exercise: Keypad mid–level primitives

Objectives

In this exercise you will gain experience with:

1. Code requirements for character I/O of a custom embedded computing application.
2. On-line debugging techniques.

Introduction

In Lab Exercise 01, we implemented a general-purpose function `double_in` that prompts the user to enter a floating-point value on the keypad, and returns the result to the calling program. That function calls the C functions `printf_lcd` and `fgets_keypad`. These functions, in turn, call other lower-level C library functions according to the following hierarchy. Functions provided by the `me477` library, core C, or the standard C library will be overwritten by those we write, which are shown in `green`.

- `double_in` (Lab 01) prompts LCD and returns keypad double
  - `fgets_keypad` (Lab 02) gets string from keypad ← this lab!
    - `getchar_keypad` (Lab 02) gets char from keypad ← this lab!
      - `getkey` (Lab 03) gets char from keypad
      - `putchar_lcd` (Lab 03) prints char to LCD
  - `printf_lcd` (Lab 01) prints string to LCD
    - `putchar_lcd` (Lab 03) prints char to LCD
    - `vsnprintf` (Lab 01) assigns to formatted string
  - `sscanf` (Lab 01) converts ASCII to binary
  - `strstr` (Lab 01) find string in string
  - `strpbrk` (Lab 01) find member in string

Continuing down the hierarchy, `fgets_keypad` gets a string from the keypad. Due to time constraints, we will not write it ourselves; instead, we will use the `me477` library version. For reference and understanding, its source code is displayed in the following listing.

```c
char *fgets_keypad(char *buf, int buflen) {
  char *bufend;
  char *p;
  int c;

  p = buf; // buffer pointer
  bufend = buf + buflen - 1; // last address in buffer
  while (p < bufend) { // one exit condition
    c = getchar_keypad(); // get char from char array
    if (c == EOF) // another exit condition
      break; // break while loop
    *p++ = c; // write to buffer, increment pointer
  }
  if(p == buf) return NULL; // just ENTR
  *p = '\0'; // write last character (NULL)
  return buf;
}
```

This function gets one keypad character at a

time from the buffered `getchar_keypad` and writes them to the character array `buf` via the pointer provided as an argument of the function. In this lab exercise, you will write the lower-level `getchar_keypad` function. This function acquires a single character from the keypad. It must function identically to the standard C function `getchar` that performs the same operations for the standard I/O device (the console). You should review the `getchar` function in your C textbook.
In Lab Exercise 03, you will write the lowest-level I/O functions `getkey` and `putchar_lcd`.

Pre-laboratory preparation

Write the following functions and compile (and debug) them before running them while connected to lab hardware.

Writing the buffered function *`getchar_keypad`*

The prototype of the `getchar_keypad` function should be as follows.

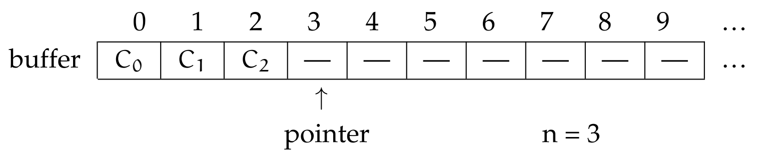
```
int getchar_keypad(void) // void means no args
```

Each time `getchar_keypad` is called it returns a single character from the keypad; and it returns EOF (defined in `stdio.h`) when it encounters its representation of ENTR. In the example below `getchar_keypad` is used to obtain a string of characters until `EOF` is reached. The characters are stored sequentially in a buffer pointed to by `point`.

```
while ( (ch=getchar_keypad()) != EOF ) {
  *point++ = ch;
}
```

There are two types of `getchar` functions in C. The first type, called an unbuffered `getchar`, simply returns the character to the calling

program immediately after each keystroke. The second type, called a buffered `getchar`, collects the characters entered by the user in a temporary buffer. Pressing ENTR causes the block of characters to be made available to the calling program. You will write a buffered `getchar_keypad` for the keypad.
The advantage of the buffered `getchar` is that the user can edit the characters in the buffer using the ← key in the usual manner, before they are sent to the calling program. There is no possibility of editing with the unbuffered `getchar`.
You might wonder how a function designed to return only a single character could edit the whole buffer. This is accomplished by a simple and elegant means inside `getchar_keypad`. The key idea is to use a statically declared character buffer. In this way, the characters remain in the buffer in between calls to `getchar_keypad`. You will also need to statically declare a pointer to the buffer, and a variable (e.g. `n`) to keep count of the number of characters in the buffer. A schematic of the buffer, pointer, and count variable is shown, below.



Here's how the buffering scheme should work. Whenever `getchar_keypad` is called either the buffer is empty or the buffer contains one or more characters.
The first time `getchar_keypad` is called, the buffer is empty, the count is zero (`n==0`), and the pointer is at the beginning of the buffer. The function enters a loop, filling the buffer and

displaying the characters, one keystroke at a time, until the ENTR key is pressed.
Each time through the loop, it checks if the buffer is full. If it's not, it completes the following tasks:

1. enter the current character into the buffer at the pointer's pointee,
2. increment the pointer,
3. increment the character count, and
4. print the character to the LCD.

After ENTR is pressed, the buffer pointer is set back to the beginning of the buffer, and the first character (alone) is returned to the calling program.
On subsequent calls to `getchar_keypad` the buffer is not empty. For each call, the pointer is incremented, the count is decremented, and the character pointed to is returned to the calling program. This continues until the last character in the buffer is returned, and the pointer is returned to the beginning of the buffer. Once the buffer is empty, the next call to `getchar_keypad` begins the filling process again. Note: `getchar_keypad` should return `EOF` to represent the ENTR key.
Putting these ideas together, algorithm pseudocode (so far) for a buffered `getchar_keypad` might look like that of Algorithm L.1, with

- `n` is the number of characters in the buffer,
- `buf` is a character array, of length `buf_len + 2`,
- `p` is a pointer that points to the location in the buffer where the next character will be put or taken, and
- `chg` is the current character from `getkey`.

Now, suppose that the ← is pressed while characters are being entered. The deleted character is effectively "removed" from the

Algorithm L.1 buffered `getchar_keypad` pseudocode

```
function getchar_keypad
    if n is 0 then                                    ▷ empty buffer!
        point p to start of buf
        while the chg is not ENTR do
            if n < buf_len then
                assign what getkey returns to chg
                assign chg to buf at p
                increment p
                increment n
                print chg to LCD with putchar_lcd
            end if
        end while
        point p to start of buf
    end if
    if n > 1 then                 ▷ more than one character in buffer
        decrement n
        return *p++               ▷ return the pointee then increment
    else if n is 1 then           ▷ one character in buffer
        decrement n
        return EOF
    end if
end function
```

buffer by decrementing both the buffer pointer `p` and the counter `n`. The deleted character is removed from the display by moving the cursor left one space, printing a space, and moving the cursor left one space again. What should happen if $\boxed{\leftarrow}$ is pressed before any characters have been entered (`n==0`)? Modify the pseudo code above (and your program) to include this "delete" functionality.

Writing the *main* function

Write a main function that tests your `getchar_keypad`. It should collect at least two separate strings using `fgets_keypad` (which calls `getchar_keypad`).

Table L.1: (left) keypad key codes and (right) `putchar_lcd` escape sequences.

key	decimal code	symbol
←	8	DEL
ENTR	10	ENT
-	45	
.	46	
0 – 9	48 – 57	
UP	91	UP
DWN	93	DN

esc seq	function
\f	clear display
\b	cursor left, 1 space
\v	cursor to start of Line-1
\n	cursor to start of Line-2

Background

To accomplish its task `getchar_keypad` must read characters from the keypad. The `getkey` function returns a single key code for each keystroke. Its prototype is as follows.

```
char getkey(void);
```

A call to `getkey` might be: `key = getkey();` Corresponding to each of the 16 keys of the keypad, the key code is shown in Table L.1. The symbols are `#define`d in the header file `me477.h`.
In addition to getting keys, `getchar_keypad` must be able to print characters `-`, `.`, and decimal digits to the LCD screen. The `me477` library function `putchar_lcd` should be used. Its prototype is as follows.

```
int putchar_lcd(int c);
```

Both the input parameter and the returned value are the character to be sent to the display. The following are some examples of calls to `putchar_lcd`.

```
ch = putchar_lcd('m');
putchar_lcd('\n');
```

It prints the character corresponding to its argument on the LCD screen.

The `putchar_lcd` function uses the same escape sequences, as shown in Table L.1, as `printf_lcd`, which we wrote in Lab Exercise 01.

Laboratory Procedure

Test and debug your program.

Guidance

The following guidance is provided for this week's lab exercise.

Compile-time integral constants

Often, we want to define a symbol that has a single integral value—an integer—throughout our program. Fortunately, C lets us do that many ways. Unfortunately, it can be hard to choose among them.
The primary ways are `#define`s (macros), `enum`s (enumerations), and `const int`s. When choosing among them, our primary concerns are code readability, debuggability, and compile-time optimization.
The last of these means a compiler (or preprocessor before the compiler) can replace each instance of the symbol with its constant value (since it never chances). There are subtle differences in how each compiler works, but most of the time all three of our options yield replaced compile-time constants. However, `#define`s are the best guarantee (because it actually happens before compilation, via preprocessing), `enum`s a close second, and `const int`s a respectable third.
In terms of debuggability, the rankings are probably best reversed; that is, in decreasing debuggability: `const int`s, `enum`s, and `#define`s. Macros (`#define`s) are most difficult because the compiler can't usually give useful error codes related to them (since the compiler

typically knows nothing of them due to preprocessing).
Readability is rather subjective, but `enum`s are typically considered strong in this regard, especially with its automatic enumeration of symbols.
A way to demonstrate this is to show the same example, written these three ways. Let's define an integral value to each day of the week, then write a script that prints a value.

```
#include <stdio.h>
enum day {
    sunday, monday, tuesday, wednesday,
    thursday, friday, saturday
};
enum day today = monday;
enum day checkout = friday;

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

```
#include <stdio.h>
#define sunday 0
#define monday 1
#define tuesday 2
#define wednesday 3
#define thursday 4
#define friday 5
#define saturday 6
#define today monday
#define checkout friday

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

```
#include <stdio.h>
const int sunday = 0;
const int monday = 1;
const int tuesday = 2;
const int wednesday = 3;
const int thursday = 4;
const int friday = 5;
const int saturday = 6;
const int today = monday;
const int checkout = friday;

int main() {
    printf("Checkout in %d days.", checkout-today);
    return 0;
}
```

```
Checkout in 4 days.
```

Preference among these three options is hotly debated, but it seems `enum`s are the most readable and the "just right" option in terms of reliable compile-time integral constant replacement and debuggability.
It is important to remember that `#define`s can be used for much more than integer replacement: function-like macros, for instance, are very useful.

Assigning to a pointee

The function `fgets_keypad`, the source for which is shown in the introduction to this lab, was used in Lab Exercise 01. Recall that in `double_in` we supplied as arguments to `fgets_keypad` a character array (pointer) and its length. Instead of returning the string, the function wrote to the character array it was supplied—but remember: inside a C function arguments are assigned automatic variables. How does `fgets_keypad` assign to the array when it knows only a pointer to its first element? The secret sauce is to assign through a dereferenced pointer. Examine the source for `fgets_keypad` or consider the following example.

```
#include <stdio.h>
void foo(int * p);

int main() {
    static int x = 0;
    static int * p = &x;
    printf("before: %d\n",*p);
    foo(p);
    printf("after: %d",*p);
    return 0;
}

void foo(int * p) {
    *p = 3;
}
```

```
before: 0
after: 3
```

Note that, while this sort of structure is rare among higher-level programming languages, it is quite common in C. For instance, `fgets` and `gets` have this same feature.

03

Digital communication and low–level io

03.1 Digital communication

Digital signals convey information via a communication channel: a wired or wireless means of transmitting and receiving electromagnetic signals over some distance. In embedded computing, most digital communication channels are wires and buses. An example of a bus is the familiar "slot" on a PC motherboard, as shown in Fig. 03.1.

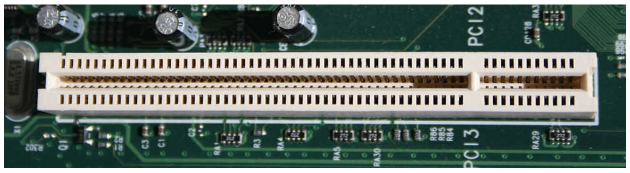


Figure 03.1: a 32-bit PCI bus (Jonathan Zander).

There are two primary divisions of digital communication: serial/parallel and synchronous/asynchronous. These divisions are explored in the following sections. Afterward, common communication protocols are described.

Serial and parallel communication

Communication can be in serial or parallel, with the former taking place sequentially over a single channel and the latter over multiple channels, as shown in Fig. 03.2. Serial communication transfers each bit of information at a time and has significant advantages for "long-haul" communication, since only a single channel is required to span the distance. Parallel communication transfers several bits in parallel, which can be faster than serial communication, but has the disadvantages of clock skew (in synchronous parallel communication, arrival of supposedly simultaneous bits can be "skewed" in time) and serialization/deserialization (converting parallel-to-series and vice versa).

However, recently improved serial communication speed has given it the advantage, even over short distances; for instance, the parallel PCI bus of Fig. 03.1 has been largely replaced by the serial PCI Express bus.

Figure 03.2: (left) serial communication and (right) parallel communication.

Synchronous and asynchronous communication

Synchronos communication is that for which a common, external "clock" times both TX output and RX input. The clock (usually a digital signal itself) signifies when the signal at the RX is valid and should therefore be read.
Asynchronous communication encodes the starting and stopping information in the bitstream itself: the RX detects a "start bit"; waits a predetermined amount of time; reads the next (often seven or eight) bits as "data" at a predetermined constant rate called the bitrate or baud rate; often reads the parity bit; and finally the "stop bit."
The parity bit is used for error checking, for which there are two general forms: even and odd parity. In even parity, the TX sends a parity bit of `0` if the number of ones in the data is even, and a `1` when it is odd. Even parity takes the opposite approach, with evens getting `1` and odds getting `0`. This lets the RX check the parity of the received data. Of course, if an even number of bits are incorrect, checking the parity bit will not be sufficient for error detection.

Standards

Communication standards define signal, electrical, connector, cable, and other characteristics that can be adopted across an entire industry. For instance, USB, Ethernet (IEEE 802.3), and RS-232 are such standards for serial communications. The Institute of Electrical and Electronics Engineers (IEEE) defines communication protocols for many wired communications and the International Organization for Standardization (ISO) defines several others.

03.2 Universal asynchronous receivers–transmitters

Universal asynchronous receivers-transmitters (UARTs) are hardware devices that allow microcontroller CPUs to asynchronously, serially communicate with other devices of the microcontroller or peripheral devices.
Often, the data input to a TX UART arrives in parallel but must transmitted in serial. This is achieved via a shift register operating in parallel-in, serial-out (PISO) mode. Consider the byte `1101 0001` into a four-bit shift register. Table 03.1 shows the register contents at each step of transmission.

Table 03.1: a four-bit shift register operating in PISO mode transmitting the byte `1101 0001` in serial.

registers					output
0	0	0	1	→	
1	0	0	0	→	1
0	1	0	0	→	0
1	0	1	0	→	0
1	1	0	1	→	0
0	1	1	0	→	1
0	0	1	1	→	0
0	0	0	1	→	1
0	0	0	0	→	1

In the corresponding RX UART, the opposite process called serial-in, parallel-out (SIPO) is also performed with a shift register in SIPO mode.
A UART can transmit and receive data with different rates, parity bits, stop bits, etc., and therefore must be properly configured. If a peripheral device requires a certain serial communication configuration, the UART transmitter controlled by a microcontroller's CPU must match this configuration.
The myRIO microcontroller has a configurable UART interface that will be used in Lab Exercise 03 to transmit data to the LCD display.

03.3 Digital signals

A digital signal is a continuous signal transmitted with the assumption that the receiver will interpret it as representing a finite number of values. Typically, only two values are represented: binary `1` or `0`, also called Boolean true ($\top$) and false ($\bot$). Digital signals are the signals of digital circuits, of which CPUs, memory reading/writing devices, and I/O devices are made.

Why would one want to discard the potentially infinite amount of information resolution in a continuous signal? Primarily because (1) often our transmission (TX) has a limited number of potential values, especially when dealing with data stored as computer memory bits, and (2) noise: offsets (biases) and random signals added to the transmitted signal through a number of mechanisms. With noise, we lose significant resolution, and a tradeoff emerges between voltage resolution and fidelity. Throughout the history of digital electronics, the tendency has been to sacrifice voltage resolution—settling for binary encoding—for time resolution: digital electronics can send and receive digital signals that switch between `0` and `1` at blazing speeds.

A number of digital signal standards have been developed,[1] with the complementary metal-oxide-semiconductor (CMOS) standard being the most popular, but several others remain in use, including the transistor-transistor logic (TTL) standard, which is now described. All these standards are similar, so describing one is sufficient for our purposes.

1. For a thorough description and history of standards, see (Horowitz and Hill, 2015).

With reference to Figure 03.1, the TTL standard defines interpretations for voltage level ranges for both transmission output and reception input. Note that the output ranges are stricter than the input ranges. This accounts for noise added to a signal between transmission and

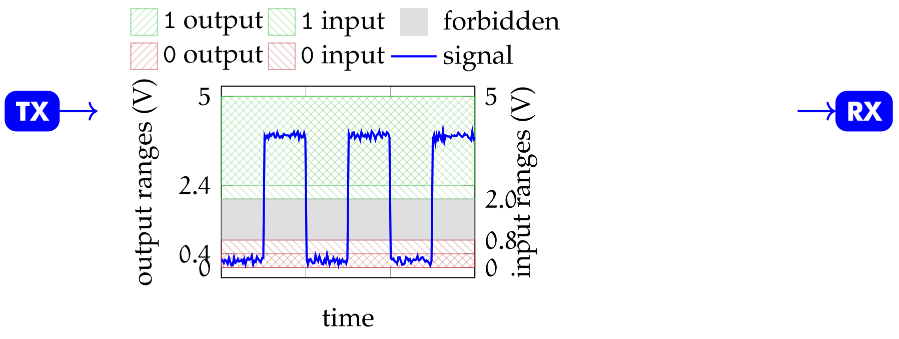


Figure 03.1: standard TTL signal transmission (TX) and reception (RX) voltage levels. Note the 0.4 V noise margin. Typically, values greater than 5 V can be received as 1 and values less than 0 V as 0.

reception, and is called the noise margin. From the figure, what is the noise margin for a TTL signal?

Most digital circuits can function properly with signals greater than their maximum defined voltage and those less than their minimum (typically 0 V). Most TTL circuits will "interpret" signals greater than 5 V as 1 and those less than 0 V as 0.

In addition to voltage specifications, the TTL standard includes current output and input ranges. Furthermore, it specifies the maximum time for a TTL-compliant device to switch between 0 and 1.

In Lecture 05.3 , some building blocks of digital circuits are described. How analog signals are converted to digital and vice versa are explored in Lecture 06.1 .

03.4 Exploring C–structures

C structures are used to group information that belongs together. The quintessential example is the tuple: coordinates that define a point.[2] The following example shows some of the syntax.

2. We follow Kernighan and Ritchie (1988, p. 129), where structures are introduced via a double (2-tuple).

```
#include <stdio.h>

int main() {
    struct point { // declare point
        double x;
        double y;
    };
    struct point pt1 = {1.2,4.5}; // declare instance
    struct point pt2; // another instance
    pt2.x = 2*pt1.x; // assign to second instance x
    pt2.y = 3*pt1.y; // assign to second instance y
    printf("pt2 = {%f,%f}",pt2.x,pt2.y);
}
```

```
pt2 = {2.400000,13.500000}
```

The first declaration `struct point { ... }` shows that two `double` types of members that are grouped into a `struct`ure with structure tag `point`. The structure tag allows us to re-use this template for further `struct`ure declarations, as with `pt1` and `pt2`—two instances of `point`. Although, in this case, the two members are of the same type (`double`), they need not be.
An instance of a `struct`ure can be assigned at declaration, as with `pt1`, or it can be assigned after declaration, as with `pt2`. The members of an instance are accessed and written-to via the name defined in the initial declaration, as in `pt2.x` and `pt2.y`.
C `struct`ures can also be nested. For instance, a line segment can be defined by two points, as shown in the following snippet, which could be interpolated into the previous `main` function.

```
struct segment { // declare segment
    struct point pt1;
    struct point pt2;
} seg1;
```

```
seg1.pt1 = pt1;
seg1.pt2 = pt2;
printf("seg1 is from {%f,%f} to {%f,%f}",
    seg1.pt1.x,seg1.pt1.y,
    seg1.pt2.x,seg1.pt2.y
);
```

Note that we can overload the names of `struct`ure members such as `pt1` and `x` without conflict. Furthermore, the syntax that declares `seg1` can be used to declare further `segment`s. A function can be passed as an argument a `struct`ure, or a pointer to it, or each of its members, separately. Similarly, a function can return `struct`ures in any of these ways. Note that `struct`ure tags declared in `main` are available to other functions.

03.exe Exercises for Chapter 03

03.L Lab Exercise: Low−level character io

Objectives

In this exercise you will gain experience with:

1. The keypad and LCD display.
2. Code requirements for character I/O of a custom embedded computing application.
3. On-line debugging techniques.

Introduction

In this lab you will write the lowest-level routines for character I/O for our keypad and LCD display. They are the `putchar_lcd` function and the `getkey` function called from `getchar_keypad` in Lab Exercise 02, as shown in the following function structure.

- `double_in` (Lab 01) prompts LCD and returns keypad double
  - `fgets_keypad` (Lab 02) gets string from keypad
    - `getchar_keypad` (Lab 02) gets char from keypad
      - `getkey` (Lab 03) gets char from keypad ← this lab!
      - `putchar_lcd` (Lab 03) prints char to LCD ← this lab!
  - `printf_lcd` (Lab 01) prints string to LCD
    - `putchar_lcd` (Lab 03) prints char to LCD ← this lab!
    - `vsnprintf` (Lab 01) assigns to formatted string
  - `sscanf` (Lab 01) converts ASCII to binary
  - `strstr` (Lab 01) find string in string
  - `strpbrk` (Lab 01) find member in string

Pre-laboratory preparation

Two functions, in addition to `main`, must be written in the exercise.

Part #1: character output: writing *putchar_lcd*

The function `putchar_lcd` puts a single character on the LCD display. The character may be any in the ASCII code or any of the escape sequences described in Lab Exercise 01 (`\f`, `\v`, `\n`, `\b`). The prototype of the `putchar_lcd` function is

```
int putchar_lcd(int value);
```

where the argument (`value`) is the character to be sent to the display. If the input value is in the range $[0, 255]$ then the returned value is also equal to the input value. If the input value is outside that range then an error is indicated by returning `EOF`.
Your version of `putchar_lcd` will replace that in the `me477` library. Calls to `putchar_lcd` might be

```
ch = putchar_lcd('m'); // or
putchar_lcd('\n');
```

Serial data is sent to the LCD display through a Universal Asynchronous Receiver/Transmitter (UART). Write the `putchar_lcd` to perform four functions:

1. Initialize the UART the first time that `putchar_lcd` is called.
2. Send a character to the display or send a decimal code to the display to implement an escape sequence.
3. Check for the success of the UART write.
4. Return the `EOF` error code, if appropriate. Otherwise, return the character to the calling program.

```
uart.name = "ASRL2::INSTR"; // UART on Connector B
uart.defaultRM = 0;         // def. resource manager
uart.session = 0;           // session reference
status = Uart_Open( &uart,  // port information
                    19200,  // baud rate
                    8,      // no. of data bits
                    Uart_StopBits1_0, // 1 stop bit
                    Uart_ParityNone); // No parity
```

Listing 03.1: initializing the UART.

The UART must be initialized once before any data is passed to the display. It is initialized through the `Uart_Open` function that sets appropriate myRIO control registers to define the operation of the UART. The initialization may be accomplished as shown in Listing 03.1, where `uart` (type: `static MyRio_Uart`) is a port information structure, and the returned value is assigned to `status` (type: `NiFpga_Status`). The macros `Uart_StopBits1_0` and `Uart_ParityNone` are defined in `UART.h`. You must `#include UART.h` in your code.
Perform this UART initialization just once, and immediately return `EOF` from `putchar_lcd` if `status` is less than the `VI_SUCCESS` macro.
Escape sequences, received as the argument of `putchar_lcd`, control the cursor position and the function of the LCD display. They are implemented by sending the escape sequences of Table L.1.
Arguments of `putchar_lcd`, in the range of `0` to `127`, are sent to the display where they are interpreted as the corresponding ASCII characters. Other arguments, in the range 128 to 255 are used for special control functions of this display.
Both escape sequences and ASCII characters are sent to the display using the `Uart_Write` function. A typical call would be as shown in Listing 03.2, where `uart` is the port information structure defined during the initialization, `writeS` (type: `uint8_t`) is an array containing

```
status = Uart_Write( &uart,  // port information
                     writeS, // data array
                     nData); // no. of data codes
```

Listing 03.2: writing to the UART.

the data to be written, and `nData` (type: `size_t`) indicates the number of elements in `writeS`. Again, return `EOF` if `status` is less than the `VI_SUCCESS`. Under normal operation (no errors), return the input character to the calling program.
See Algorithm L.1 for `putchar_lcd` pseudocode.

Part #2: keypad input: writing *getkey*

You will write the `getkey` function, which waits for a key to be depressed on the keypad, and returns the character code corresponding to that key. The prototype of the `getkey` function is

```
char getkey(void);
```

Your version of `getkey` will replace that in the C library. A call to `getkey` might be:

```
key = getkey();
```

The keypad is a matrix of switches. When pressed, each switch uniquely connects a row conductor to a column conductor. The row and column conductors are connected to eight digital I/O channels of connector-B (`DIO-0`–`DIO-7`) of the myRio as shown in Fig. L.1.
Each channel may be programmed to operate as either a digital input or an output. As an output, the channel operates with low output impedance as it asserts either a high or a low voltage at its terminal. Programmed as an input, the channel has high input impedance ("Hi-Z mode") as it detects either a high or a low voltage.

Algorithm L.1 buffered `putchar_lcd` pseudocode

```
function putchar_lcd(c)    ▷ c is ASCII character code
    initialize variables    ▷ include static int iFirst=1
    if iFirst==1 then    ▷ first call!
        initialize UART (Listing 03.1)    ▷ status ← Uart_open(...)
        if status < VI_SUCCESS then
            return EOF
        end if
        iFirst=0
    end if
    n ← 1    ▷ assume n (data points) is 1
    if c == '\f ' then    ▷ clear display, backlight on
        S[0] ← 17    ▷ S is uint8_t array
        S[1] ← 12
        n ← 2    ▷ n actually 2 in this case
    else if c == '\b ' then    ▷ cursor backspace
        S[0] ← 8
    else if c == '\v ' then    ▷ cursor line-0
        S[0] ← 128
    else if c == '\1' then    ▷ cursor line-1
        S[0] ← 148
    else if c == '\2' then    ▷ cursor line-2
        S[0] ← 168
    else if c == '\3' then    ▷ cursor line-3
        S[0] ← 188
    else if c == '\n ' then    ▷ cursor to next line
        S[0] ← 13
    else if c > 255 then    ▷ outside range
        return EOF
    else    ▷ send ascii code
        S[0] ← c cast as uint8_t    ▷ cast syntax (uint8_t) c
    end if
    write S to UART (Listing 03.2)    ▷ status ← Uart_Write(...)
    if status < VI_SUCCESS then
        return EOF
    else
        return c
    end if
end function
```

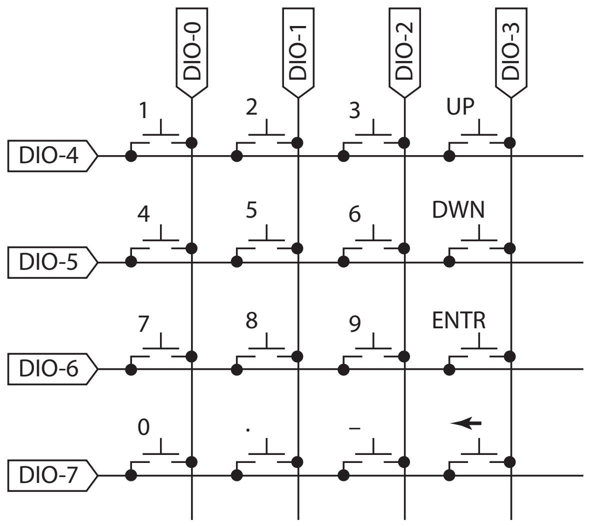


Figure L.1: keypad circuit.

How will we detect if a key is depressed? Briefly, this is accomplished by driving (as output) one column to low voltage (digital false), with the other columns channels in Hi-Z mode. Then, all of the rows are scanned (detected). If a row is found to be low, the key connecting that row to the driven column must be depressed. This procedure is repeated for each column. The entire process is repeated until a key is found.
Essential to this scheme is that a pull-up resistor is connected between each channel and the high voltage.[3] So, unless a row is connected (through a key) to a low-impedance, low-voltage column, it will always read high.

3. The NI myRIO-1900 User Guide and Specifications describes the DIO as having built-in 40 KΩ pull-up resistors to 3.3 V (Instruments, 2013, p. 11).

Strategy A strategy for `getkey` is shown in the pseudocode Algorithm L.2.

Channel initialization The `MyRio_Dio` structure, defined in `DIO.h`, identifies the control registers and the bit to read or write for a channel.

```
typedef struct { uint32_t dir;   // direction register
                 uint32_t out;   // output value register
                 uint32_t in;    // input value register
```

Algorithm L.2 `getkey` pseudocode

```
function getkey
    initialize the 8 digital channels
    while a low bit not detected do
        for each column do
            for each column do
                set column to Hi-Z
            end for
            set one column low
            for each row do
                read bit
                if bit is low then
                    break row loop
                end if
            end for
            if bit is low then
                break out of column loop
            end if
        end for
        wait for some msec
    end while
    while row is still down do
        wait for some msec
    end while
    identify key from row, column in table
    return key
end function
```

```
                uint8_t bit;     // Bit to modify
} MyRio_Dio;
```

Declare an array of `MyRio_Dio` structures, one element for each of the 8 necessary channels. In a loop initialize the channels as follows.

```
MyRio_Dio Ch[8];
for (i=0; i<8; i++) {
    Ch[i].dir = DIOB_70DIR;
    Ch[i].out = DIOB_70OUT;
    Ch[i].in = DIOB_70IN;
    Ch[i].bit = i;
}
```

Again, the symbols shown are defined in `DIO.h`.

Channel I/O

Input—Digital channel read function prototype:

```
NiFpga_Bool Dio_ReadBit(MyRio_Dio* channel);
```

For example, a typical call might be:

```
bit = Dio_ReadBit(&Ch[row+4]);
```

Note: In addition to reading the bit, `Dio_ReadBit` sets the channel to Hi-Z mode.
Output—Digital channel write function prototype:

```
void Dio_WriteBit(MyRio_Dio* channel, NiFpga_Bool value);
```

For example, a typical call might be:

```
Dio_WriteBit(&Ch[col], NiFpga_False);
```

The data type `NiFpga_Bool` may take values of either `NiFpga_True` (high), or `NiFpga_False` (low).

Key code The key code returned by `getkey` is determined by the indices of a key code table. The key code table can be stored in a statically declared 4×4 array of characters.

```
char table[4][4] = { {'1','2','3', UP},
                     {'4','5','6', DN},
                     {'7','8','9',ENT},
                     {'0','.','-',DEL}  };
```

For example, if the detected row was `1`, and the column was `2`, then the value of `table[1][2]` is the character `'6'`.
The symbols `UP`, `DN`, `ENT`, `DEL` are defined in `me477.h`.

Wait The x ms time delay will be determined by executing a delay-interval routine. The "wait" function below is suggested. It executes in a small fraction of a second. In next week's lab we will calculate and measure its precise duration.

```
/*----------------------------------------
Function wait
      Purpose:      waits for x ms.
      Parameters:   none
      Returns:      none
*------------------------------------------*/
void wait(void) {
  uint32_t i;

  i = 417000;
  while(i>0){
    i--;
  }
  return;
}
```

Writing the *main* function

Write a main function that tests your versions of putchar_lcd and getkey. It should:

1. Make at least one individual call to each of putchar_lcd and getkey. Be sure to test the value-out-of-range error returned by putchar_lcd.
2. Collect an entire string using fgets_keypad (which automatically calls getkey).
3. Write an entire string using printf_lcd (which automatically calls putchar_lcd). Be sure to test all four escape sequences.

Laboratory Procedure

Test and debug your program.

Part III

Timing, Threads, and Finite State Machines

04

Finite state machine control

Finite state machines model the behavior of an intelligent system as consisting of a finite number of states and transitions thereamong. These models are commonly used in the design of intelligent systems.
This chapter introduces some additional concepts of importance:

- pulse-width modulation (Lec. 04.1),
- the driving of a DC motor (Lec. 04.2), and
- measuring motor position and velocity (Lec. 04.3).

Finally, finite state machines are introduced in Lec. 04.4 . In Lab Exercise 04, we apply a finite state machine model to basic DC motor speed control.

04.1 Pulse−width modulation[1]

1: This lecture appears also in the Mechatronics Laboratory Manual.

1 Pulse-width modulation (PWM) is a technique used to deliver an effectively variable signal to a load (in our case a motor) without a truly variable power source. A pulse of full source amplitude is repeated at a high frequency (e.g. 20 kHz), delivering a signal that is effectively averaged by the load dynamics such that its effects on the load are nearly continuous. The fraction of the period T that the signal is high (on) is called the duty cycle δ. Fig. 04.1 shows a PWM signal $v(t)$ and its average $\bar{v}(t)$ with a few parameter definitions.

2 The mean of any periodic signal with period T can be computed with the integral

$$\bar{v}(t) = \frac{1}{T}\int_0^T v(t),$$

which is easily evaluated for a PWM signal:

$$\bar{v}(t) = \frac{Aw}{T} = A\delta.$$

3 This result shows that if a PWM signal is delivered to a load, such as a DC motor, that is relatively unaffected by high-frequency signals, the effective signal will be simply the product of the source amplitude A and the duty cycle δ. The duty cycle can have values from 0 to 1, so

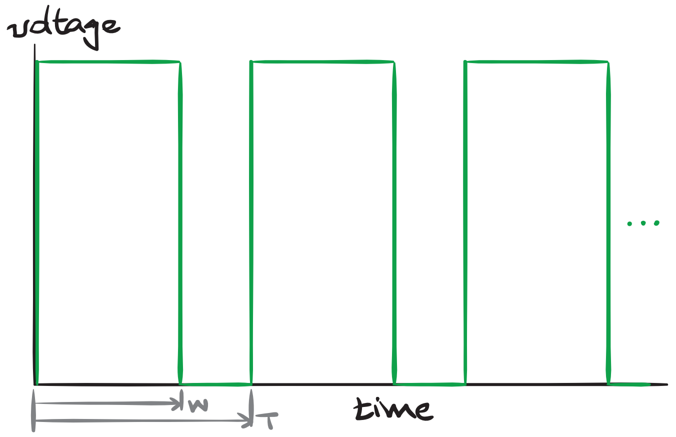


Figure 04.1: a pulse-width modulation (PWM) signal.

the effective DC signal produced varies linearly with δ from 0 to A.

04.2 DC motor driving

1 There are two common methods of driving DC motors: (a) with a digital motor driver and (b) with an analog amplifier. Schematics of both are shown in Fig. 04.1.

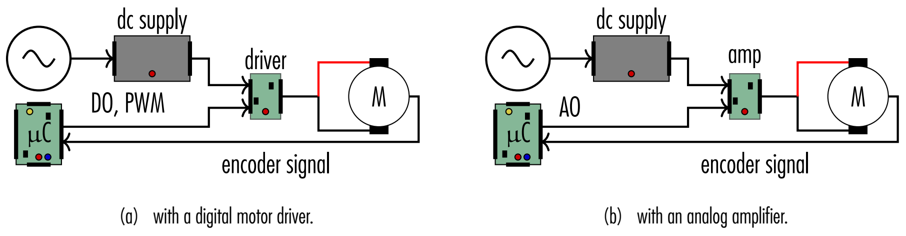


Figure 04.1: two common methods of driving motors.

Digital motor drivers

2 A microcontroller such as the myRIO or Arduino can easily produce a PWM signal, which, as we saw in Lec. 04.1 , can be averaged by a system's dynamics such that varying the duty cycle varies the averaged signal. However, microcontrollers are low-power and cannot drive even small DC motors. Therefore it is common to include a special kind of integrated circuit (IC) that uses the microcontroller's low-power PWM signal to gate a high-power DC source signal for delivery to the motor. These are called digital motor drivers; a common system setup with a motor driver is shown in Fig. 04.1(a). They deliver power from a high-power source in accordance with a PWM signal, and they often include many additional features such as

1. compact forms;
2. forward- and reverse-driving (see Lec. 04.2)
3. protection against reverse voltage, overcurrent, and overheating; and

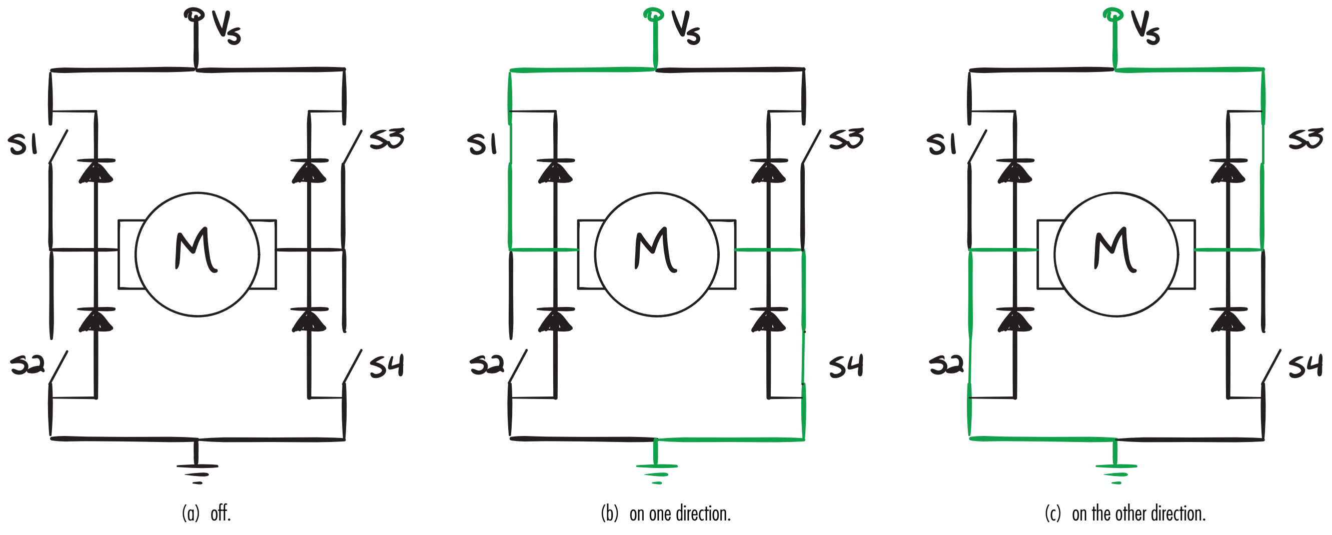
4. output pins that monitor delivered current and voltage.

3 These digital motor drivers are sometimes called class-D or switching amplifiers. Generally, they inexpensive and are quite efficient (around 90 % in some cases), which, in addition to conserving power, adds the capacity of delivering high-power operation or requiring lower heat dissipation (or a "Goldilocks" mixture thereof).

H-bridge circuits

4 We want to drive DC motors at different effective voltages and different directions. An H-bridge circuit allows us to reverse the direction of the PWM signal delivered to the motor. Fig. 04.2 is a diagram of the H-bridge circuit.

5 The switches S1-S4 are typically instantiated with MOSFET transistors. As shown in the figure below, during the high duration of the PWM pulse, either S1 and S4 (Fig. 04.2(b)) or S2 and S3 (Fig. 04.2(c)) are closed and the others are open.



(a) off. (b) on one direction. (c) on the other direction.

Figure 04.2: H-bridge operation.

6 Recall that the electronics side of a DC motor can be modeled as a resistor and inductor in series with an electro-mechanical transformer. The inductance of the windings make it an “inductive” load, which presents the following challenge. We can’t rapidly change the current flow through an inductor without a huge spike in voltage, and the switches do just that, leading to switch damage. Therefore, during the low or “off” duration of the PWM signal, S1-S4 cannot all be simply opened. There are actually a few options for switch positions that allow the current to continue to flow without inductive “kickback.”

7 What’s up with the diodes? Technically, they could be used to deal with the kickback. But since the diodes dissipate power, the proper switching is the primary kickback mitigation technique. However, the diodes ease the transition between switch flips, which are never quite simultaneous.

Analog amplification

8 An alternative to digital motor drivers are analog amplifiers, which require a slightly different setup, shown in Fig. 04.1(b). This setup requires an analog signal from the microcontroller, a digital device. Therefore, the microcontroller performs a process called digital-to-analog conversion (DAC), treated further in Lec. 06.1 and Resource 14. Many microcontrollers have this functionality and can produce analog signals over ranges such as ± 10 V, the range of the myRIO’s `C10` channel analog outputs.

9 An amplifier essentially “adds power” to the microcontroller analog output from an external power source. There are several varieties that can operate as voltage/current-controlled voltage/current sources within a range of

operation. When that range is exceeded, operation typically becomes nonlinear and finally saturates (increased input does not increase output). Saturation is, of course, one of several considerations when designing with amplifiers.

10 A comparison between digital motor drivers and analog amplifiers is given in Table 04.1. For more, see (Collins, 2018).

Table 04.1: comparison between digital motor drivers and linear analog amplifiers.

feature	digital motor driver	analog amplifier
cost	less expensive	more expensive
signal noise	noisy	minimal
audible noise	load	none
low-signal fidelity	poor	good
high-precision control	poor	good
efficiency	90 %	50 %
heat generation	low	high
high-powered	> 100 W	< 100 W
brushed/less dc	good	good
H-bridge	required	not required

The ECL instantiation

11 The Embedded Computing Lab (ECL) has both digital motor drivers and linear analog amplifiers, both of which are commercially available.

Digital motor drivers

12 For a digital motor driver, we use a connectorized printed circuit board (PCB)—the Pololu motor driver carrier:

pololu.com/product/1451
ricopic.one/resources/pololu_VNH5019.pdf.
(manual)

This includes an STMicroelectronics VNH5019 H-bridge motor driver integrated circuit:

`ricopic.one/resources/vnh5019.pdf`.

13 This type of motor driver is commonly found in small-motor applications such as those in an automobile used for adjusting seat, window, and mirror positions.

Analog amplifiers

14 The Copley Controls 412 voltage-controlled current source (or transconductance) amplifiers in the ECL are actually switched amplifiers, internally (so they're relatively efficient and capable of high-power), but function as analog amplifiers. This is a standard type of motor amplifier found in industrial settings. The device manual can be found here:

`ricopic.one/resources/Copley412.pdf`.

15 For the amplifier settings used in the ECL, see Resource 10.

04.3 Measuring motor position and velocity

1 Motor position and angular velocity are best measured by rotational quadrature encoders. Rotational encoders are made from a wheel with alternating dark and light stripes called lines. The encoder we have affixed to the rear shaft—the HP HEDS-5640-A06 with manual

`ricopic.one/resources/encoder_manual.pdf`

—has black lines on clear plastic. A light source either reflects differently off the stripes or, as in our case, passes the light through the clear plastic wheel into a photodiode or is blocked by the black stripes. Each time a stripe passes by, the photodiode detects a "blink," which is passed on to the myRIO via digital channels of the myRIO configured for detecting encoder outputs.

2 The encoder pinout is shown in Fig. 04.1, from the manual.

Quadrature encoders

3 The only issue remaining is that a given "blink" doesn't give one important piece of information: which direction the encoder is rotating. However, a clever technique called quadrature encoding can be used to determine direction. If we offset one of the two

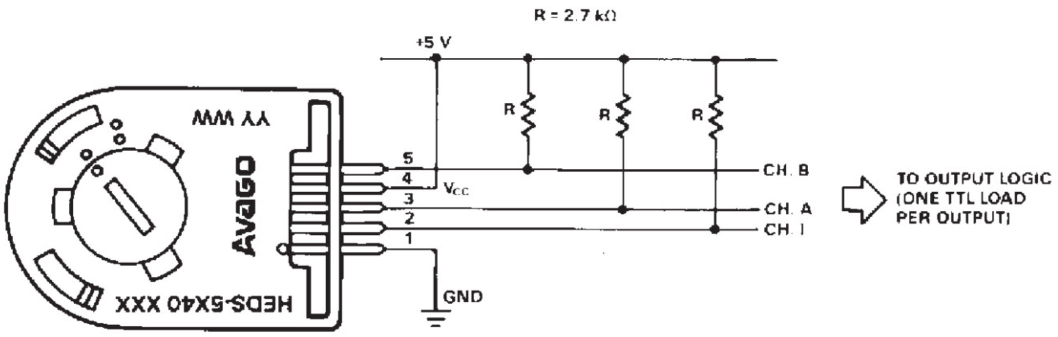


Figure 04.1: the encoder used (source: HEDM-55xx/560x & HEDS-55xx/56xx Data Sheet).

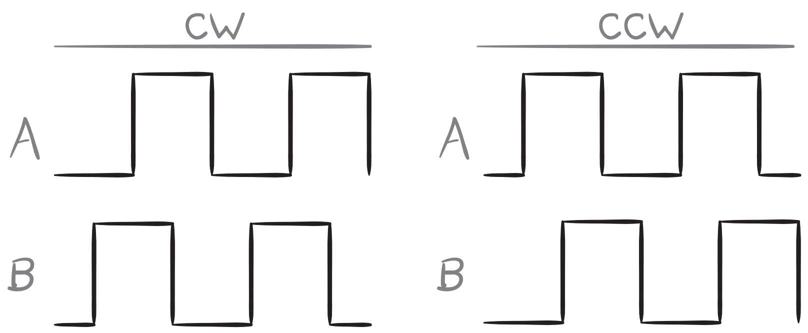


Figure 04.2: quadrature encoding with channels `A` and `B`.

sources/detectors by half of a stripe width, then measure both "channels" `A` and `B`, then the direction can be determined by which channel "leads" the other. For instance, in Fig. 04.2, the encoder output is high when light is detected and low when it is blocked by a stripe. Channel `A` leads `B` when the encoder is rotating clockwise (CW) and `B` leads `A` when it is rotating counter-clockwise (CCW).

4 Note that this also gives us better resolution in that we detect four blinks per line. So a 500 line (CPR) quadrature encoder changes state $4 \times 500 = 2000$ times per revolution.

04.4 Finite state machines

A program that sequences a series of actions, or handles inputs differently depending on what mode it's in, is often implemented as a finite state machine. A state is a condition that defines a prescribed relationship between inputs and outputs, and between inputs and subsequent states. A finite state machine is an algorithm that can be in a finite number of different states. For example, consider the control algorithm for an elevator operating between two floors. The elevator has four possible states:

1. stopped on floor-1,
2. stopped on floor-2,
3. moving up, and
4. moving down.

Inputs include:

1. the buttons that are pushed in the elevator car and on each floor and
2. limit switches indicating that the car has reached each floor.

The outputs are the commands

1. to the lift motor,
2. to the elevator doors, and
3. to the indicator displays in the car and on the floors.

The outputs and the transition from one state to another depend on the current state and inputs. A state machine for which the outputs are functions of both the current state and the inputs is called a Mealy machine. A state machine for which the outputs are functions of only the current state is called a Moore machine. An advantage of using state machines is that the necessary logic can be represented graphically in a state transition diagram. A state transition diagram shows the input/output relationships

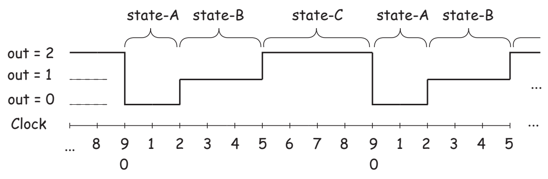


Figure 04.1:

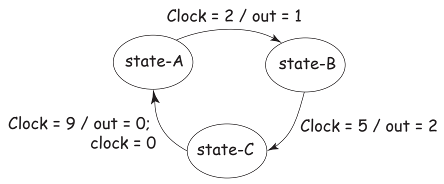


Figure 04.2:

and the conditions for transitions between states. A skeleton of code that implements any state transition diagram can be standardized. Let's examine the state transition diagram for a simple example, and see how it might be coded. This system contains three states (A, B, and C). Its only input is the sequential count of a variable `Clock` (0, 1, 2, ...). Its outputs are a variable `out` and the `Clock` (which the algorithm may reset to 0). The clock increments at a fixed rate. Potential state transitions are evaluated at each clock count.

The state machine operates as follows. The system stays in A until `Clock == 2`, then it sets `out = 1`, and changes to B. It stays in B until `Clock == 5`, then sets `out = 2`, and changes to C. Finally, it stays in C until `Clock == 9`, then sets `out = 0`, resets the clock (`Clock = 0`), and changes back to A. The process repeats indefinitely, producing a periodic output of 9 clock counts. A plot of the output would look like that of Fig. 04.1.

This complicated natural language specification of the system operation can be represented very simply in a state transition diagram, such as that of Fig. 04.2.

The arrows between states are commonly

Table 04.1: state transition table with ◯: no change.

when state is	and input is	then output		and make state
	`Clock`	`out`	`Clock`	
A	2	1	◯	B
B	5	2	◯	C
C	9	0	0	A

labeled as:

$$\left\langle \begin{array}{c}\text{event that caused}\\ \text{the transition}\end{array} \right\rangle \Big/ \left\langle \begin{array}{c}\text{output(s) as a}\\ \text{result of the}\\ \text{transition}\end{array} \right\rangle$$

Often the information in the state transition diagram is described in the form of a state transition table, such as that of Table 04.1.
As shown, the table lists all possible transitions between states, the conditions that cause the state transitions, and the corresponding outputs.
Now, how can this be efficiently coded? The listing on the following page illustrates one possibility.[2] You will need to study this code carefully. Be sure that you understand all the C constructs. Some of them are tricky!

2. See also Gomez (2000).

Each state is implemented as a separate C function. The heart of the program is the "Main state transition loop" (note: just three lines of code!) This infinite loop calls the function corresponding to the current state. The variable `curr_state` keeps track of which state is current. The loop also causes a wait for one clock period, increments `Clock`, and then repeats.
The primary task of each state function is to determine if the current state should be changed. If no change is needed, the function does nothing. If the state is to be changed, the function sets `curr_state` to the new state and alters the outputs appropriately.
A function, `initializeSM`, is included in the following to initialize the state machine.

```c
/* State Machine Example */

#include <stdio.h>

/* Prototypes */
void stateA(void);
void stateB(void);
void stateC(void);
void initializeSM(void);
void wait(void);

/* Define an enumerated type for states */
typedef enum {STATE_A=0, STATE_B, STATE_C} State_Type;

/* Define an array of pointers to each state function */
static void (*state_table[])(void) = {
  stateA, stateB, stateC
};

/* Global variable declaration */
static State_Type curr_state; // The "current state"
static int Clock;
static int out;

void main(void) {
  /* Initialize the state machine */
  initializeSM();

  /* Main state transition loop */
  while (1) {
    state_table[curr_state](); // call cur. state fnct.
    wait(); // wait fixed time interval
    Clock++;
  }
}

/* SM initialization function */
void initializeSM(void) {
  curr_state = STATE_A;
  out = 0;
  Clock = 1;
}

/* State functions */
void stateA(void) {
  if( Clock == 2 ) {       // change state?
    curr_state = STATE_B; // next state
    out = 1;              // new output
  }
}

void stateB(void) {
  if( Clock == 5 ) {       // change state?
    curr_state = STATE_C; // next state
    out = 2;              // new output
  }
}

void stateC(void) {
  if( Clock == 9 ) {       // change state?
    Clock = 0;            // reset clock
    curr_state = STATE_A; // next state
    out = 0;              // new output
  }
}
```

At first, this may appear to be unnecessarily complicated for this simple example. However, the same code can be expanded easily (by adding more state functions) to implement a state machine of any complexity, with an unlimited number of states, inputs, and outputs.

04.exe Exercises for Chapter 04

Exercise 04.13

Consider the series of actions controlling the operation of an airplane landing gear. For example, beginning in the "stowed" position, when the cockpit switch is set to "lower", first the landing gear door opens; and then the gear moves down to the "locked" position. Subsequently, when the switch is set to "raise", the gears moves up, and the door closes. See Fig. exe.1.

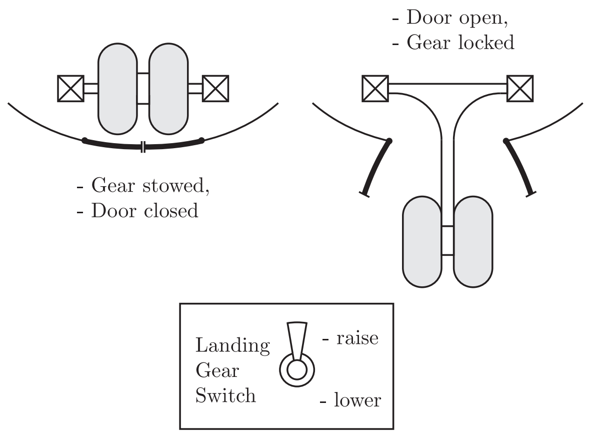


Figure exe.1: landing gear schematic.

Our task is to design a Finite State Machine (FSM) to produce outputs that actuate the motors that move the door and gear. Inputs to the FSM are from the cockpit switch, and from door and gear position limit sensors.

Operation rules The following are the required operation rules.

1. When the switch is set to "raise", the gear moves up to the stowed position, and then the door closes.
2. When the switch is set to "lower", the door opens, and then the gear moves down to the locked position.
3. If, while the gear is moving up, the switch is changed to "lower", the

gear should reverse direction and move down to the locked position.

4. If, while the gear is moving down, the switch is changed to “raise”, the gear should reverse direction and move up to the stowed position, with the door closed.
5. ...
6. If, while the door is closing, the switch is changed to “lower”, the sequence should reverse: door opens and gear moves down.
7. If, while the door is opening, the switch is changed to “raise”, the sequence should reverse: door closes.

FSM Inputs There are three inputs to the FSM. The possible values of each variable are shown in brackets.

1. switch (sw), [raise, lower] A two-position Landing Gear Switch used to command the raising or lowering of the gear.
2. gear limit sensor (gs), [top, bottom, other] The limit sensor variable gs indicates whether the gear is at the top, the bottom, or in between.
3. door limit sensor (ds), [opened, closed, other] The limit sensor variable ds indicates whether the door is completely open, completely closed, or in between.

FSM Outputs There are two outputs from the FSM. Again, the possible values of each variable are in brackets.

1. gear motor (gm), [raising, lowering, off] The motor variable gm controls whether the gear motor is raising or lowering the gear, or is off.
2. door motor (dm), [opening, closing, off] The motor variable dm controls

whether the door motor is opening or closing the door, or is off.

FSM States At any time, the landing gear control system can be in one of six states. The system remains in a state until conditions are met that cause a transition to another state.

Name the states as follows:

1.	gear stowed (up)	GS
2.	gear locked (down)	GL
3.	gear moving up	GMU
4.	gear moving down	GMD
5.	door moving open	DMO
6.	door moving closed	DMC

Perform the following steps to complete the exercise.

1. Draw the state transition diagram. Use the input, output, and state variable names and values defined above. For each transition show the event that caused the transition, and the output resulting from the transition.
2. From your diagram fill in the corresponding state transition table Table exe.1. Again, use the input, output, and state variable names and values defined previously. Use a "—" to indicate no change in a variable.

Table exe.1: the state transition table.

	Inputs			Outputs		
current state	sw	gs	ds	gm	dm	next state
GL						
GMU						DMC
GMU						
DMC						GS
GS						
DMO						GMD
GMD						
GMD						GL
DMO						
DMC						

04.L Lab Exercise: Finite state machine motor control

Objectives

The objectives of this exercise are to:

1. Become familiar with optical encoding.
2. Implement a finite state machine control algorithm.
3. Understand pulse-modulation control of a dc motor.
4. Use instruction timing to produce a calibrated delay.

Introduction

In this exercise, your program will drive and monitor the speed of a dc motor using a finite state machine model. The myRIO will drive the motor with pulse-width modulation (PWM) on a DIO channel configured as a digital output. This digital signal will be amplified by the analog amplifier described in Lec. 04.2 , as shown in Fig. L.1. The speed of the motor will be measured with a quadrature encoder on the motor and read by the myRIO FPGA encoder counter. Two buttons connected to myRIO DIO inputs will also control the operation of the system.

Pulse-width modulation

Channel-0 of Connector A, the digital signal on which we call $\overline{\text{run}}$ (the line over name denotes a logical "not," so we call this signal "not-run"), is connected to a motor driver circuit such that when $\overline{\text{run}}$ is `1` (high),[3] no voltage is applied to the motor; and when $\overline{\text{run}}$ is `0` (low), 20 V is applied. Your program will periodically alter this digital signal, applying an oscillating signal to the motor. The duty cycle (the percentage of time power is applied) is the percentage of time the channel is low.

3. We use the C notation that the integer `1` means boolean true and the integer `0` means boolean false.

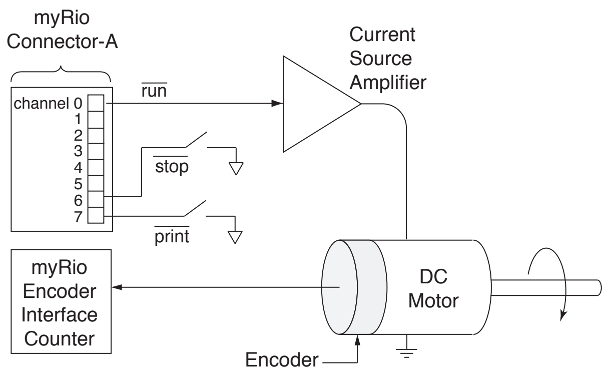


Figure L.1: a schematic of the pulse-modulation via $\overline{\text{run}}$ DIO output, the speed measurement via the FPGA encoder input, and the UI buttons $\overline{\text{print}}$ and $\overline{\text{stop}}$.

Encoder and counter

An optical encoder is mounted on the shaft of the dc motor. The encoder is the Avago HEDS-5640-A06. It is a quadrature encoder. It has 500 lines (i.e. counts per revolution, CPR), and two LED/Phototransistor pairs. The two signals (e.g. `A` and B) are 90 degrees of phase apart. If the encoder is rotating clockwise, `A` leads `B` by 90 degrees. If the encoder is rotating counter-clockwise `A` lags `B` by 90 degrees. This is how direction is encoded. In total, then, there are $4 \times 500 = 2000$ state changes per revolution. Therefore, each encoder state change corresponds to a motor rotation of 1/2000 revolution, called a basic displacement increment (BDI).
The two phases are connected to a quadrature counter. The counter detects two state changes (one down-to-up and one up-to-down) for each line passage. Two changes for phase A and two for phase B: a total of four state changes for each line. So, for one revolution the counter totals 2000 state changes, and counts up or down depending of which phase leads.
An encoder counter in the FPGA interface determines the total number of these state

changes. The speed is determined by computing the number of state changes from the encoder during a certain time interval, called the basic time interval (BTI). Therefore, the number of state changes occurring during each interval represents the angular speed of rotation in units of BDI/BTI.

Initializing the encoder counter

Counting of the encoder state changes is accomplished by the FPGA associated with the Xilinx Z-7010 system-on-a-chip, with dual Cortex-A9 ARM processors. The counter must be initialized before it can be used. Initialization includes identifying the encoder connection, setting the count value to zero, configuring the counter for a quadrature encoder, and clearing any error conditions. The function `EncoderC_initialize`, included in the `me477` library, alters the appropriate control registers to initialize the encoder interface on Connector C. The prototype for the initialization function is:

```
NiFpga_Status EncoderC_initialize(
  NiFpga_Session myrio_session,
  MyRio_Encoder *channel
);
```

The first argument, `myrio_session` (type: `NiFpga_Session`), identifies the FPGA session, and must be declared as a global variable for this application. That is, above `main`,

```
NiFpga_Session myrio_session;
```

The second argument `channel` (type: `MyRio_Encoder *`) points to a structure that maintains the current status and count value, and must also be declared as a global variable. We will use encoder #0. For example,

```
MyRio_Encoder encC0;
```

Reading the encoder counter

The position of the encoder (in BDI) may be found at any time by reading the counter value. The prototype of a library function provided for that purpose is:

```
uint32_t Encoder_Counter(MyRio_Encoder* channel);
```

where the argument is the counter channel declared during the initialization, and the returned value is the current count in the form of a 32-bit integer.

Pre-laboratory preparation

Main Program

Write a main program that produces a periodic waveform on $\overline{\text{run}}$ that applies an average voltage to the motor determined by the duty cycle. The period and 1 BTI will be controlled by calling N `wait` functions, each of which takes the same deterministic amount of time. During the first M `waits` each period, voltage will be applied to the motor. See the first graph in Fig. L.2.

In addition, while Channel 7 of Connector A is `0`, the program will print the measured speed on the display at the beginning of each BTI. You will control Channel 7 through a push button switch. The corresponding $\overline{\text{run}}$ waveform is shown in the second graph of Fig. L.2.

The algorithm should be implemented as a finite state machine (see Lec. 04.4). As shown in Fig. L.3, the machine will have five possible states: high, low, speed, stop, and (the terminal) exit. The inputs will be the `Clock` variable, and channels 6 and 7 (for the $\overline{\text{stop}}$ and $\overline{\text{print}}$ buttons). The outputs will be $\overline{\text{run}}$, `Clock` (which sometimes needs reset to `0`), and the motor speed printed to the LCD display. The corresponding state transition table, listing all possible transitions, is shown in Table L.1.

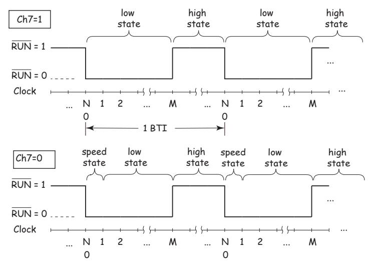


Figure L.2: $\overline{\mathrm{run}}$ waveforms for (top) when the $\overline{\mathrm{print}}$ button is not being pressed and (bottom)when the $\overline{\mathrm{print}}$ button is being pressed.

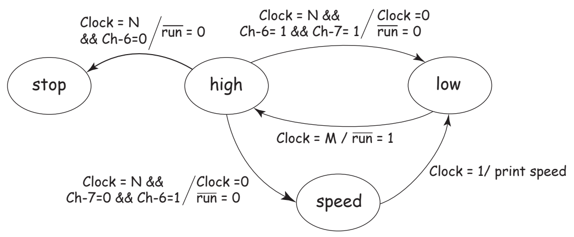

Clock = N && Ch-6=0 / run = 0
Clock = N && Ch-6= 1 && Ch-7= 1 / Clock =0 run = 0
stop
high
low
Clock = M / run = 1
Clock = 1/ print speed
Clock = N && Ch-7=0 && Ch-6=1 / Clock =0 run = 0
speed

Figure L.3: State transition diagram.

Overall, the `main` program will:

1. Use `MyRio_Open` to open the myRIO session, as usual.
2. Setup all interface conditions and initialize the finite state machine using `initializeSM`, described, below.
3. Request, from the user, the number (N) of wait intervals in each BTI.

Table L.1: ×: irrelevant input, ○: no change in output.

when state is	and input is			then output			and make state
	$\overline{\text{stop}}$ Ch6	$\overline{\text{print}}$ Ch7	`Clock`	$\overline{\text{run}}$ Ch0	`Clock`	speed	
high	1	1	N	0	0	○	low
high	1	0	N	0	0	○	speed
high	0	×	N	0	0	○	stop
low	×	×	M	1	○	○	high
speed	×	×	1	○	○	print	low
stop	×	×	×	○	○	○	exit

4. Request the number (M) of intervals the motor signal is "on" in each BTI.
5. Start the main state transition loop.
6. When the main state transition loop detects that the current state is `exit`, it should close the myRIO session, as usual.

Functions

In addition to `main`, several functions will be required, as described, below. These functions include one for each state: `high` for high, `low` for low, `speed` for speed, and `stop` for stop.

`double_in` To execute the user I/O you may use the routine `double_in` developed in Lab Exercise 01, or you may simply call it from the `me477` library:

```
double double_in(char *string);
```

`initializeSM` Perform the following:

1. Initialize channels 0, 6, and 7 on Connector A, in accordance with Fig. L.1, by specifying to which register each DIO corresponds. For example, for Channel 6,[4]

```
Ch6.dir = DIOA_70DIR; // "70" used for DIO 0-7
Ch6.out = DIOA_70OUT; // "70" used for DIO 0-7
Ch6.in = DIOA_70IN;   // "70" used for DIO 0-7
Ch6.bit = 6;
```

4. See `NiFpga_MyRio1900Fpga30.h` and `MyRio1900.h` for more description of the NI FPGA U8 Control `enum`s that specify register addresses. For instance, `DIOA_158DIR` is short for `NiFpga_MyRio1900Fpga30_ControlU8_DIOA_158DIR`, which is at address `0x181C6`, and stores the "direction" (input or output) of a DIO pin in the upper bank (pins 8–15) of Connector A.

2. Additionally, initialize Connector A DIO Channels 1 and 2 in the usual way. Furthermore, set them to 1 and 0, respectively, via `Dio_WriteBit`. (An example would be `Dio_WriteBit(&Ch1, NiFpga_True);` which sets Channel 1 to 1.) This sets the motor direction via its input pins INA and INB (1, 0 is "positive" rotation and 0, 1 is "negative"). See the motor driver manual for more information.
3. Initialize the encoder interface. See above.
4. Stop the motor (set $\overline{\text{run}}$ to `1`).
5. Set the initial state to low.
6. Set the `Clock` to `0`.

`high` If `Clock` is N, set it to 0 and $\overline{\text{run}}$ to `0`. If Ch7 is `0`, change the state to speed. If Ch6 is `0`, change the state to stop. Otherwise, change the state to low.

`low` If `Clock` is M, set $\overline{\text{run}}$ to `1`, and change the state to high.

`speed` Call `vel`. The function `vel` reads the encoder counter and computes the speed in units BDI/BTI. See `vel` below. Convert the speed to units of revolutions/min. Print the speed as follows:

```
printf_lcd("\fspeed %g rpm",rpm);
```

Finally, change the state to low.

`vel` Write a function to measure the velocity. Each time this subroutine is called, it should perform the following functions. Suppose that this is the start of the nth BTI.

1. Read the current encoder count: c_n (interpreted as an 32-bit signed binary number, `int`).
2. Compute the speed as the difference between the current and previous counts: $(c_n - c_{n-1})$.

3. Replace the previous count with the current count for use in the next BTI.
4. Return the speed `double` to the calling function.

Note: The first time `vel` is called, it should set the value of the previous count to the current count.

`stop` The final state of the program.

1. Stop the motor. That is, set $\overline{\text{run}}$ to `1`.
2. Clear the LCD and print the message: "stopping".
3. Set the current state to `exit`. The `while` loop in `main` should terminate if the current state is `exit`.
4. Save the response to a Matlab file. (See laboratory procedure of Lec. 04.L.)

`wait` Your program will determine the time by executing a calibrated delay-interval function. Consider this "wait" function.

```
/*----------------------------------
Function wait
Purpose:      waits for xxx  ms.
Parameters:   none
Returns:      none
*----------------------------------*/
void wait(void) {
  uint32_t i;

  i = 417000;
  while(i>0){
    i--;
  }
  return;
}
```

Notice that the above program does nothing but waste time! The compiler generates the following operation codes for this function. The first column contains the addresses, and the second contains the corresponding opcodes.

```
wait+0  push {r11}
wait+4  add r11, sp, #0
wait+8  sub sp, sp, #12

wait+12 mov r3, #417000
wait+16 str r3, [r11, #-8]
wait+20 b 0x8ed4 <wait+36>

wait+24 ldr r3, [r11, #-8]
wait+28 sub r3, r3, #1
wait+32 str r3, [r11, #-8]
wait+36 ldr r3, [r11, #-8]
wait+40 cmp r3, #0
wait+44 bne 0x8ec8 <wait+24>

wait+48 nop ; (mov r0, r0)
wait+52 add sp, r11, #0
wait+56 ldmfd sp!, {r11}
wait+60 bx lr
```

The clock frequency of our microprocessor is 667 MHz.[5] Note carefully how the branch instructions are used. Determine the exact number of clock cycles[6] for the code to execute, accounting for all instructions. From that, calculate the delay interval in ms.

5. See Instruments (2013).

6. See ARM (2012), Appendix B.

When free running, the speed of the motor is approximately 2000 RPM. Considering all the above, determine a reasonable value for N, the number of delay intervals in a BTI. What inaccuracies or programming difficulties are there in using a delay routine for control and time measurement?

Header files

The following header files will be required by your code.

```c
#include <stdio.h>
#include "Encoder.h"
```

```
#include "MyRio.h"
#include "DIO.h"
#include "me477.h"
#include <unistd.h>
#include "matlabfiles.h"
```

Modulo Arithmetic

We will estimate the rotational speed by computing the difference between the current encoder count c_n and the previous count c_{n-1}. The counter is capable of counting up and down, depending on the direction of rotation. Interpreting the count as 32-bit signed binary, the value is in the range $[-2^{31}, 2^{31} - 1]$. For example, starting from 0 and rotating in the clockwise direction, the count will increase until it reaches $2^{31} - 1$, then roll over to -2^{31}, and continue increasing.
How will this rollover affect our estimate of the velocity? Assume that the current and previous counts (c_n and c_{n-1}) are assigned to signed integer variables of width equal to that of the counter. For our C compiler the `int` data type is 32 bits (4-bytes). Further assume that the angular position of the encoder changes less than $2^{32}/2000$ revolutions (about 2 million revolutions!) during a single BTI. That is, $|c_n - c_{n-1}| < 2^{32}$.
When we compute the difference between two signed integer data types, the result is defined by the offset modulo function:

$$\operatorname{mod}(m, n, d) = m - n \left\lfloor \frac{m - d}{n} \right\rfloor \quad (1)$$

where m is the value, n is the modulus, d is the offset, and $\lfloor x \rfloor$ is the floor function (i.e. the greatest integer less than or equal to x.) The result is modulo-n, and always in the range $[d, d + n - 1]$.
Then, for our case of `int` data, we estimate the relative displacement using modulo 2^{32}, with

offset $d = -2^{31}$.

$$\begin{aligned}\Delta\theta &= \operatorname{mod}(c_n - c_{n-1}, 2^{32}, -2^{31}) \\ &= c_n - c_{n-1} - 2^{32}\left\lfloor \frac{c_n - c_{n-1} - (-2^{31})}{2^{32}} \right\rfloor \quad (2)\end{aligned}$$

Let's examine what happens when we cross the rollover point. Suppose that the previous counter value c_{n-1} was $2^{31} - 2$. And, that during the BTI the encoder has moved forward by $+4$, such that the current reading c_n is $-2^{31} + 2$. The numerical difference $c_n - c_{n-1}$ is $-4,294,967,292$. However, applying Equation 2, the 32-bit signed integer arithmetic gives the correct result: $\operatorname{mod}(-4294967292, 2^{32}, -2^{31}) = +4$. Note that C is automatically implementing Equation 2 and this description is to deepen our understanding.

Laboratory Procedure

1. Examine the circuit on the breadboard on Connector A of the myRIO. The push button switches of Fig. L.4 connect channels 6 and 7 to ground when pressed. Note: These channels have pull-up resistors.

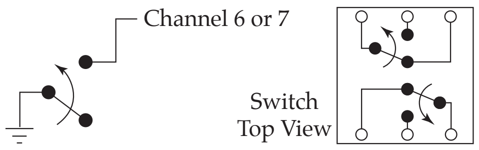


Figure L.4:

2. Use the oscilloscope to view the waveform produced by your program. For example, use $N = 5$, $M = 3$.
3. Use the oscilloscope to view the start/stop waveform produced by your program, and to measure the actual length of a BTI. Is it what you expect? If not, why not?
4. Repeat the previous step while printing the speed (press the switch). What does

the oscilloscope show has happened to the length of the BTI. What's going on!?

5. Describe how you made this measurement and discuss any limitations in accuracy. In a later lab, we will find ways of overcoming this limitation.
6. Recording a step response
   After you have your code running as described above, try this: Record the velocity step response of the DC motor, save it to a file and plot it in Matlab.
   Here's how:
   Add code to your speed function to save the measured speed at successive locations in a global buffer. You will need to keep track of a buffer pointer in a separate memory location. Increment the buffer pointer each time a value is put in the buffer. The program must stop putting values in the buffer when it is full. For example,

```
#define IMAX    2400          // max points
static  double  buffer[IMAX]; // speed buffer
static  double  *bp = buffer; // buffer pointer
```

   and, in the executable code,

```
if (bp < buffer+IMAX) *bp++ = rpm;
```

   To record an accurate velocity, temporarily comment-out the printf_lcd statement in speed, and hold down the Ch7 switch while you start the program.
7. Saving the response
   The program should save the response stored in the buffer to a Matlab (.mat) file on the myRIO under the real-time Linux operating system during the stop state. See Resource 9 for more details.
   The Matlab file must be called Lab4.mat. In the file, save the speed buffer, the values of N and M, and a character string

containing your name. The name string will allow you to verify that the file was filled by your program.
For your report, the array can be plotted using the Matlab `plot` command.

a) From your plot, estimate the time constant of the system. Plotting points, instead of a continuous line, will make interpretation easier.
b) What is the steady-state velocity in RPM?

8. Extra: fixing the $M = 1$ case
You may have noticed that when $M = 1$ the finite state machine does not function as desired. What is wrong? How would modifying the state transition diagram correct this problem? How would you modify the state transition table? Modify your program to correct the $M = 1$ case. Test the result.

A better way to PWM

In this exercise, among other things, we have come to understand PWM and the limitations of implementing it in the way we have.
Fortunately, there is a better way: using the PWM capabilities from the FPGA, accessible via `PWM.h`.
The PWM example (`myRIO Example - PWM`) from the NI archive `C_Support_for_myRIO_v3.0.zip` shows how to do this. This method mitigates several of the issues encountered in this exercise, especially those related to duty-cycle resolution and "jerky" operation due to low PWM rates. The FPGA-based PWM can operate as high as 10 MHz, but, as with our finite state machine implementation, loses duty cycle resolution as its rate increases. So, although we have a higher rate, and therefore more cushion, the same issue

of balancing PWM frequency and duty cycle resolution remain.

Of course, we still might need a finite state machine for controlling the state at a higher-level. For instance, we might include a knob for controlling the duty cycle of the FPGA PWM. This and the encoder, speed, and stop functionality of the finite state machine of the exercise could have a much lower frequency (governed by `wait`) than the PWM. Decoupling the timing for processes at much different rates, like this, is typically advantageous.

Resource R9 Saving myRIO C data to a Matlab file

The following C functions[7] write data of types `double` or `char` to a Matlab `.mat` file. They are included in the `me477` library. Be sure to `#include "matlabfiles.h"`.
Use the following functions to open a named file on the myRIO, and successively add any number of data arrays, variables, and strings to the file. Finally, close the file.

7. See http://www.malcolmmclean.site11.com/www/MatlabFiles/matfiles.html.

Open a `.mat` file The prototype for the open function is

```
MATFILE *openmatfile(char *fname, int *err);
```

where `fname` is the filename, and `err` receives any error code. The function returns a structure for containing the Matlab file pointer.
A typical call might be:

```
mf = openmatfile("Lab.mat", &err);
if(!mf) printf("Can't open mat file %d\n", err);
```

For this course, always use the file name: `Lab.mat`. Notice the use of pointers.

Add a matrix The prototype of the function for adding a matrix to the Matlab file is

```
int matfile_addmatrix(
  MATFILE *mf,
  char *name,
  double *data,
  int m,
  int n,
  int transpose
);
```

where `mf` is the Matlab file pointer from the open statement, `name` is a `char` string containing the name that the matrix will be given in Matlab, `data` is a C data array of type `double`, `m` and `n` are the array dimensions, `transpose` takes value of 0 or 1 to indicate where the matrix is to be transposed.

For example, to add a 1-D matrix the call might be

```
matfile_addmatrix(mf, "vel", buffer, IMAX, 1, 0);
```

Or, to add a single variable the call might be

```
double Npar;
Npar = (double) N;
matfile_addmatrix(mf, "N", &Npar, 1, 1, 0);
```

Again, note the use of pointers, and the cast to `double`.

Add a string The prototype of the function for adding a string to the Matlab file is

```
int
matfile_addstring(
  MATFILE *mf,
  char *name,
  char *str
);
```

where `mf` is the Matlab file pointer from the open statement, `name` is a `char` string containing the name that the matrix will be given in Matlab, and `str` is the string. For example, to add a string the call might be

```
matfile_addstring(mf,"myName","Bob Smith");
```

Close the file After all data have been added, the file must be closed. The prototype of the function for closing the Matlab file is

```
int matfile_close(MATFILE *mf);
```

where `mf` is the Matlab file pointer from the open statement.
For example, to close the Matlab file the call might be

```
matfile_close(mf);
```

Example code Putting these ideas together:

```
mf = openmatfile("Lab.mat", &err);
if(!mf) printf("Can't open mat file %d\n", err);
```

```
matfile_addstring(mf, "myName",  "Bob Smith");
matfile_addmatrix(mf, "N", &Npar, 1, 1, 0);
matfile_addmatrix(mf, "M", &Mpar, 1, 1, 0);
matfile_addmatrix(mf, "vel", buffer, IMAX, 1, 0);
matfile_close(mf);
```

Transfer file to Matlab After the `Lab.mat` file has been created, it can be transferred directly to Matlab.

1. In Eclipse's right pane of the Remote Systems Explorer perspective, select `172.22.11.2`, and select the icon Refresh information of selected resource.
2. Double click on the Matlab data file: `172.22.11.2>SftpFiles>MyHome>Lab.mat`.
3. The `Lab.mat` file will be opened in Matlab on your laptop. Use Matlab's `whos` command to list all of the named variables in the workspace.
4. In Matlab navigate to a convenient folder on your laptop. Then, issue the `save('Lab.mat')` command to save the Matlab workspace, locally. The file can later be opened from a Matlab script, using the command `load('Lab.mat')`, for plotting or analysis.

Note: You will also find the `Lab.mat` file in the `RemoteSystemsTempFiles` folder within your workspace folder.

Resource R10 Copley 412 analog amplifier setup

This should be adequate (and safe) for the Clifton Precision JDH-2000-V-1C or similar dc motor. It has a stall voltage of 24 V and stall current of 2.18 A.

Resistor settings

- RH15 Peak Current 6.2 kΩ (20 % of 20 A = 4 A)
- RH16 Continuous Current Limit 0 Ω (16 % of 20 A = 3.2 A)
- RH17 Peak Current Time Limit `open` (1 second)
- RH20 Armature Inductance 49.9 kΩ (0.6 to 1.9 mH)

Capacitor settings

- CH18 Armature Inductance 4.7 nF (0.6 to 1.9 mH)

Dip switch settings

- S1 Ground-active Enable `OFF` (up, away from the board)
- S2 Torque Mode `ON` (down, toward the board)

Gain adjustment

We will be operating the amplifier in `TORQUE MODE`. For transconductance, (output current / input voltage) = peak current / 10 V, which can be set up with the following steps:

1. Set S2 `ON`.
2. Set Ref Gain fully CW.
3. Set Loop Gain fully CCW.
4. Adjust the transconductance gain to 4 A / 10 V.
    a) To increase gain, turn Loop Gain CW.

b) To decrease gain, turn Ref Gain CCW.

05

Threads and interrupts

05.1 Processing threads

A processing thread is a sequences of instructions to be executed by the CPU. One or more threads typically comprise a process. Threads can share the same memory space and other resources, but processes are typically independent.
The computer operating system's scheduler controls the execution of processes and threads. For most modern processors, each core can handle two threads by sharing a core, which swaps back-and-forth between the threads—called simultaneous multithreading (SMT) or time-slicing.[1] A schematic of this process is shown in Fig. 05.1.

1. Intel uses the term "hyperthreading" for SMT.

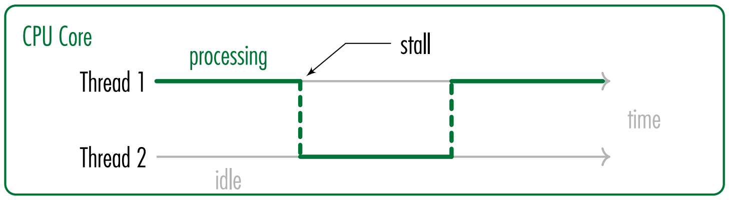


Figure 05.1: a schematic of simultaneous multithreading (SMT) in a single CPU core.

Although the core is not actually simultaneously processing the threads, there is frequently an overall speedup by exploiting stalls in the thread such as cache misses: when a thread requires data not available in the CPU caches and must wait for the data from some relatively slow source, such as the main memory. When there is a cache miss in one thread, the other can execute in what would have otherwise been stalled processing time. Frequently, a single program will make use of multiple threads. UNIX-based operating systems, such as the NI Linux Real-Time OS of the myRIO, have a standardized (IEEE POSIX 1003.1c) threading language in C called POSIX threads (Pthreads). This widely used interface is incorporated into

header files of the C library provided with the myRIO. This is how we will implement threading in our programs from Lab Exercise 05, on.
The ARM Cortex A-9 processor of the myRIO has two cores and is capable of handling multiple threads.

05.2 Interrupts

Embedded computing frequently requires a program to respond to events the timing of which is unkown, beforehand. These events include

- digital user input such as keypad and button presses;
- other digital input such as limit switch detection; and
- analog input such as sensor values.

One way to handle these types of events is to frequently poll the analog and digital inputs in a program's main loop. However, there are two drawbacks to this method: (1) events can be missed if the inputs are polled too infrequently, (2) it is difficult to control polling timing in a loop that contains other processes, and (3) the main loop's timing can be affected by the additional time it takes to handle the event.
The first and second concerns are mitigated by using another method: threaded interrupt handling. A new thread (in addition to the main thread) is created to handle an interrupt. This thread processes an interrupt service routine (ISR) which checks to see if there has been an interrupt request (IRQ) and, if so, responds to the event. The IRQ can be expressed in memory or externally on a programmable interrupt controller (PIC). In either case, the ISR frequently checks for the corresponding IRQ and, once serviced, clears it. We call an IRQ-ISR pair an interrupt.
Even when only a single core is available, interrupts can be given high-priority by the OS scheduler (via simultaneous multithreading) such that the program will be responsive to interrupts. Of course, unless the threads are run on distinct cores, an interrupt thread does add to the time of the iteration of the main loop (our

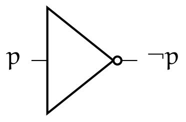
third concern from above). For some applications (such as that of Lab Exercise 05), this additional time is negligible. But for many real-time applications, this will be problematic. Mitigation can be achieved by using a timer interrupt, which will be explored in Lab Exercise 06.

05.3 Boolean algebra on digital signals

We will require an understanding of Boolean algebra on digital signals to implement a switch debouncing circuit in Lec. 05.4 . It is a digital circuit that operates with logic gates, which are here introduced.

A digital signal's Boolean variable values 1 and 0 are isomorphic to propositional calculus's truth values $\top$ (true) and $\bot$ (false). Similarly, Boolean algebra (i.e. Boolean logic) operations are isomorphic to propositional calculus operations, such as not ($\neg$), and ($\wedge$), and or ($\vee$). Table 05.1 is a truth table for a number of Boolean algebra operators.

Digital electronics instantiate these operators as logic gates, sometimes as subcircuits of CPUs and sometimes as discrete integrated circuits for incorporation on a prototyping board (as in Lab Exercise 05) and eventually on a PCB. The simplest gate is the not gate, which has the following circuit symbol.



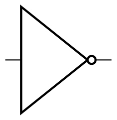
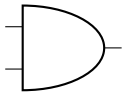
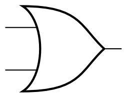
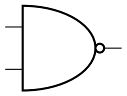
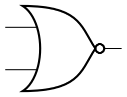
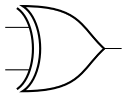
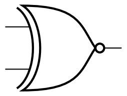
This gate accepts digital signal represented by Boolean variable p and returns $\neg p$. So, $p = 1 \Rightarrow \neg p = 0$ and $p = 0 \Rightarrow \neg p = 1$.

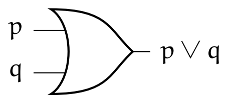
Most gates have two inputs. For instance, the or gate, what has circuit symbol

Table 05.1: a truth table for logic operations. The first two columns are operation inputs, the rest, outputs.

		not	and	or	nand	nor	xor	xnor
p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \uparrow q$	$p \downarrow q$	$p \veebar q$	$p \Leftrightarrow q$
0	0	1	0	0	1	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	1	0	0	0	1

Table 05.2: logic operations and equivalent C expressions and gate symbols.

name	logic	C	gate
not	$\neg p$	`!p`	
and	$p \wedge q$	`p&&q`	
or	$p \vee q$	`p||q`	
nand	$p \uparrow q$	`!(p&&q)`	
nor	$p \downarrow q$	`!(p||q)`	
xor	$p \veebar q$	`p!=q`	
xnor	$p \Leftrightarrow q$	`p==q`	



accepts digital signals with Boolean variables (say) p and q and returns $p \vee q$. Table 05.2 summarizes logic gates and their associated Boolean algebra operators.

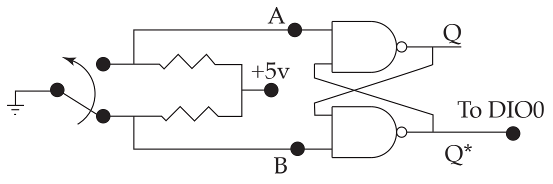
05.4 Debouncing circuits for switches

When a mechanical switch is thrown via a button, toggle, or some other interface, the new contact between the two conductors is not immediately seamless. In fact, over a few milliseconds, contact is made and broken dozens of times[2]. This phenomenon is called switch contact bounce.

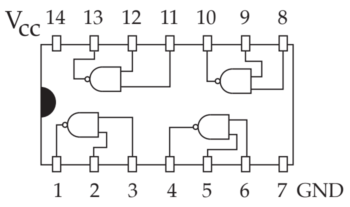
2. Horowitz and Hill, 2015.

Often, we mitigate switch bounce with a circuit—called a debouncing circuit—between the switch and the microcontroller. Debouncing circuits yield a single transition of the digital signal, low-to-high or high-to-low.

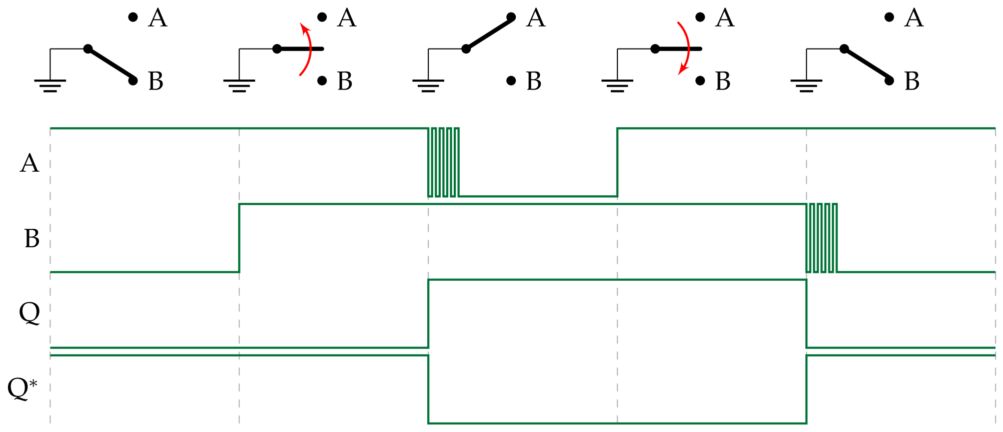
Consider in detail the debouncing circuit of Fig. 05.1. For the outputs to switch, both inputs must switch, effectively mitigating bounce.



(a) a debouncing circuit for a mechanical switch.



(b) 7401 quad nand, open-collector outs.



(c) logic levels corresponding different switch states through time.

Figure 05.1: an illustration of the operation of a debouncing circuit. With the switch initially drawing B low, Q* must be high and Q low. The loss of contact with B does not affect Q* or Q. Initial contact with A draws a low and therefore Q high and Q* low. The ensuing bounce doesn't affect Q because it doesn't affect Q* being low, so Q is high, regardless of A. This logic is then mirrored in the transition from contact with A to B, with its ensuing bounce. A TTL IC, shown in (b), can be used to instantiate this circuit.

05.exe Exercises for Chapter 05

05.L Lab Exercise: Introduction to interrupts

Objectives

The objectives of this exercise are to:

1. introduce the use of interrupts in I/O programming,
2. introduce the use of multiple threads,
3. become familiar with digital signal conditioning for interrupts, and
4. use TTL gates to "debounce" a switched input.

Introduction

This exercise illustrates the use of interrupts, originating from sources that are external to the microcomputer. The principal activity of your `main` program is to print the value of a counter on the LCD display. If uninterrupted, the counter display, which is updated once per second, would continue for 60 counts.
Generally, the "service" of an interrupt, may be arbitrarily complex in both form and function. However, in this exercise, each time an interrupt request (IRQ) occurs, the interrupt service routine (ISR) thread will simply print out the message, "`interrupt_`". A push-button switch on an external circuit will cause the IRQ to occur.
Therefore, the overall effect will be that the display will print the count repeatedly, with the word "`interrupt_`" interspersed for each push of the switch.
Although this program is not long, it is essential that you understand the events that take place at the time of the interrupt: (1) an unscheduled (asynchronous) external event causes the activity of the CPU to be suspended, and (2) a separate section of code (ISR) executes, before returning control to the original program at

precisely the point where the execution was interrupted. That the counter display continues to run accurately both before and after the interrupt illustrates that the main program is not altered, regardless of where the interrupt occurs in the execution.

The Threads

The *main* thread

The main program runs in the main thread. It will perform the following tasks:

1. Open the myRIO session.
2. Register the interrupt and the digital input (see below).
3. Create an interrupt thread to "catch" the interrupt (see below).
4. Begin a loop. Each time through the loop:
    - Wait one second by calling the (5 ms) `wait` function (from Lab Exercise 04) 200 times.
    - Clear the display and print the value of an `int count`.
    - Increment the value of `count`.
5. After a `count` of 60, signal the interrupt thread to stop, and wait until it terminates.
6. Unregister the interrupt.
7. Close the myRIO session.

The ISR thread

The ISR runs in an interrupt thread, separate from the main thread. It should begin a loop that terminates only when signaled by the main thread. Within the loop it will:

1. Wait for an external interrupt to occur on `DIO0`.
2. Service the interrupt by printing the message: "`interrupt_`" on the LCD display.

3. Acknowledge the interrupt.

Background

Several library interrupt functions are used in the following. For more documentation on them, see Resource 11.

Setting up *main* for interrupts, generally

Within `main` we will configure the DI interrupt and create a new thread to respond when the interrupt occurs. The two threads communicate through a globally defined thread resource structure:

```
typedef struct {
  NiFpga_IrqContext irqContext; // IRQ context reserved
  NiFpga_Bool irqThreadRdy;     // IRQ thread ready flag
  uint8_t irqNumber;            // IRQ number value
} ThreadResource;
```

National Instruments provides two C functions to set up the digital input (DI) interrupt request (IRQ).

Register the DI0 IRQ The first of these functions reserves the interrupt from the FPGA and configures the DI and IRQ. Its prototype is:

```
int32_t Irq_RegisterDiIrq(
  MyRio_IrqDi* irqChannel,
  NiFpga_IrqContext* irqContext,
  uint8_t irqNumber,
  uint32_t count,
  Irq_Dio_Type type
);
```

where the five input arguments are:

1. `irqChannel`: a pointer to a structure containing the registers and settings for the IRQ I/O to modify; defined in `DIIRQ.h` as:

```
typedef struct{
  uint32_t dioCount;          // count register
  uint32_t dioIrqNumber;      // number register
  uint32_t dioIrqEnable;      // enable register
  uint32_t dioIrqRisingEdge;  // rising edge-trig reg.
  uint32_t dioIrqFallingEdge; // falling edge-trig reg.
  Irq_Channel dioChannel;     // supported I/O
} MyRio_IrqDi;
```

2. `irqContext`: a pointer to a context variable identifying the interrupt to be reserved. It is the first component of the thread resources structure.
3. `irqNumber`: the IRQ number (1–8).
4. `count`: the number times the interrupt condition is met to trigger the interrupt.
5. `type`: the trigger type used to increment the count.

The returned value is `0` for success.

Create the interrupt thread The second function, `pthread_create` called from `main`, creates a new thread and configures it to "service" the DI interrupt. Its prototype is:

```
int pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void * (*start_routine) (void *),
  void *arg
);
```

where the four input arguments are:

1. `thread`: a pointer to a thread identifier.
2. `attr`: a pointer to thread attributes. In our case, use `NULL` to apply the default attributes.
3. `start_routine`: name of the starting function in the new thread. The prototype syntax means the function `start_routine`, which will be given argument `arg` in the new thread, should be given to `pthread_create` with no argument.

4. `arg`: the sole argument to be passed to `start_routine`. In our case, it will be a pointer to the thread resource structure defined above and used in the second argument of `Irq_RegisterDiIrq`.

This function returns `0` for success.

Setting up *main* for our interrupt, specifically

We can combine these ideas into a portion of the `main` code needed to initialize the DI IRQ.[3] For interrupts on falling-edge transitions on `DIO0` of Connector A, assigned to IRQ 2, we have:

3. Note: the IRQ channel settings symbols (and others) associated with the DI interrupt, are defined in header files: `DIIRQ.h` and `IRQConfigure.h`.

```
int32_t status;
MyRio_IrqDi irqDI0;
ThreadResource irqThread0;
pthread_t thread;
int i, j, count=0;

// Open the myRIO NiFpga Session.
status = MyRio_Open();
if (MyRio_IsNotSuccess(status)) return status;

// Configure the DI IRQ number, incremental times,
// and trigger type.
const uint8_t IrqNumber = 2;
const uint32_t Count = 1;
const Irq_Dio_Type TriggerType = Irq_Dio_FallingEdge;

// Specify the settings that correspond to
// the IRQ channel to be accessed.
irqDI0.dioChannel = Irq_Dio_A0;
irqDI0.dioIrqNumber = IRQDIO_A_0NO;
irqDI0.dioCount = IRQDIO_A_0CNT;
irqDI0.dioIrqRisingEdge = IRQDIO_A_70RISE;
irqDI0.dioIrqFallingEdge = IRQDIO_A_70FALL;
irqDI0.dioIrqEnable = IRQDIO_A_70ENA;

// Initiate the IRQ number resource of interrupt thread.
irqThread0.irqNumber = IrqNumber;

// Register  DIO IRQ. Terminate if not successful.
status=Irq_RegisterDiIrq(
  &irqDI0,
  &(irqThread0.irqContext),
  IrqNumber,
  Count,
  TriggerType
);
if (status != NiMyrio_Status_Success) {
  printf(
    "Status: %d\nConfiguration of DI IRQ failed\n",
    status
  );
  return status;
}

// Set the indicator to allow the interrupt thread.
irqThread0.irqThreadRdy = NiFpga_True;

// Create interrupt threads to catch
// the specified IRQ numbers.
status = pthread_create(
  &thread,
  NULL,
  DI_Irq_Thread,
  &irqThread0
);
```

Other `main` tasks go here.
After the other `main` tasks are completed, it should signal the new thread to terminate by setting the `irqThreadRdy` flag in the `ThreadResource` structure. Then, wait for the thread to terminate. For example,

```
irqThread0.irqThreadRdy = NiFpga_False;
status = pthread_join(thread,NULL);
```

Finally, the interrupt must be unregistered:

```
status = Irq_UnregisterDiIrq(
  MyRio_IrqDi* irqChannel,
  NiFpga_IrqContext irqContext,
  uint8_t irqNumber
);
```

using the same above arguments. To use the `pthread` functions, `#include <pthread.h>` in your code.

The ISR thread

This is the separate thread that was named and started by the `pthread_create` function. Its overall task is to perform any necessary function in response to the interrupt. This thread will execute until signaled to stop by `main`.

The beginning of the new thread is the starting routine specified in the `pthread_create` function called in `main`:
`void *DI_Irq_Thread(void* resource)`.
The first step in `DI_Irq_Thread` is to cast its input argument into appropriate form. In our case, we cast the `resource` argument back to the `ThreadResource` structure. For example, declare

```
ThreadResource* threadResource =
  (ThreadResource*) resource;
```

The second step is to enter a loop. Two tasks are performed each time through the loop, as described in Algorithm L.1.

Algorithm L.1 ISR thread loop pseudocode

```
while the main thread has not signaled this thread to stop do
    wait for the occurrence (or timeout) of the IRQ
    if the numbered IRQ has been asserted then
        perform operations to service the interrupt (print interrupt_)
        acknowledge the interrupt
    end if
end while
```

Let's explore how to do this. The `while` loop should continue until the `irqThreadRdy` flag (set in `main`) indicates that the thread should end. For example:[4]

```
while (threadResource->irqThreadRdy == NiFpga_True) {
  // stuff!
}
```

4. For pointer to a structure `struct * a` with member name `b`, the member value can be accessed with `a->b`, which is equivalent to `(*a).b`.

The two tasks within the loop are as follows.

1. Use the `Irq_Wait` function to pause the loop while waiting for the interrupt. For our case the call might be:

```
uint32_t irqAssert = 0;
Irq_Wait(
  threadResource->irqContext,
  threadResource->irqNumber,
  &irqAssert,
  (NiFpga_Bool*) &(threadResource->irqThreadRdy)
);
```

Notice that it receives the `ThreadResource` context and IRQ number information, and returns the `irqThreadRdy` flag set in the `main` thread.

2. Because `Irq_Wait` times out after 100 ms, we must check the `irqAssert` bit flag[5] to see if our numbered IRQ has been asserted.
   In addition, after the interrupt is serviced, it must be acknowledged to the scheduler. For example, using bitwise operators,[6]

```
if (irqAssert & (1 << threadResource->irqNumber)) {
  // Your interrupt service code here
  Irq_Acknowledge(irqAssert);
}
```

5. A bit flag is bit of independently useful information stored in a (larger) integer variable. This is because a byte is the smallest addressable unit of memory. Of course, multiple bit flags can be assigned to a single integer variable.

6. The bitwise operator `<<` shifts `1` of `...0001` left `irqNumber` bits. Then the bitwise and `&` "bit masks" to see if any bits of both numbers match (there's only potentially one match, the `irqNumber` bit). Note that any nonzero integer is considered true (`1`) for a conditional statement.

The third step terminates the new thread and returns from the function:

```
pthread_exit(NULL);
return NULL;
```

Laboratory procedure

Build, debug, and execute your program. Provide interrupt signal by connecting the single-pole-double-throw (SPDT)[7] switch on the circuit bread board to `DIO0` of Connector A as shown in Figure L.1. Try your program. What happens? This undesirable phenomenon is caused by the bounce of the mechanical switch. Adjust the oscilloscope to examine the high-to-low transition of the IRQ signal. Typically, what length of time is required for the transition to settle at the low level? How many TTL triggers occur during the settling? Correct the problem by replacing the switch in Figure L.1 with the debouncing circuit shown in

7. The switch is actually double-pole-double-throw (DPDT), but one pole is disconnected.

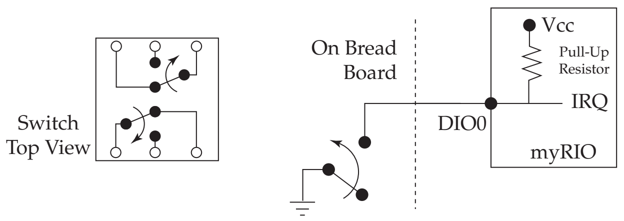


Figure L.1: Connecting the interrupt signal to myRIO.

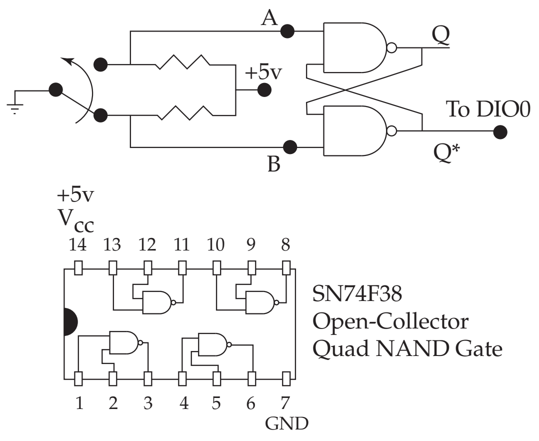


Figure L.2: Debouncing circuit.

Figure L.2. This circuit incorporates a (TTL) quad open-collector NAND gate (7401).

Box 05.1 caution

Be certain that V_{cc} and `GND` are connnected to the chip before wiring the rest of the circuit.

Try your program again. Explain, in detail, why this circuit should solve the switch bounce problem. That is, graph the time-history of signals at points A and B that would occur during the operation of a bouncing switch. Then, graph the corresponding signals at Q and Q*.

Finally, in your own words, explain how the `main` thread configures the interrupt thread, how it communicates with the interrupt thread during execution, and how the interrupt thread functions.

Resource R11 Interrupt functions documentation

This resource includes some documentation of functions from the National Instruments `C_Support_for_myRIO` library (included in the `me477` library) used in Lab Exercise 05. For more details, see the `me477` library header files `DIIRQ.h` and `IRQConfigure.h` and POSIX C library `pthread.h`.

Register DI IRQ

`Irq_RegisterDiIrq()`	Reserves the interrupt from FPGA and configures DI IRQ. Declared in the `DIIRQ.h` header file.

Prototype:

```
int32_t Irq_RegisterDiIrq(
  MyRio_IrqDi       *irqChannel,
  NiFpga_IrqContext *irqContext,
  uint8_t           irqNumber,
  uint32_t          count,
  Irq_Dio_Type      type
);
```

Arguments:

- `irqChannel` structure containing the registers and settings for a digital IRQ I/O
- `irqContext` IRQ context to be reserved
- `irqNumber` the IRQ number (`IRQNO_MIN-IRQNO_MAX`)
- `count` the incremental times that you use to trigger the interrupt
- `type` the trigger type that you use to increment the count
- `return` the configuration status

Unregister DI IRQ

`Irq_UnregisterDiIrq()`	Clears the DI IRQ configuration setting. Declared in the `DIIRQ.h` header file.

Prototype:

```
int32_t Irq_UnregisterDiIrq(
  MyRio_IrqDi       *irqChannel,
  NiFpga_IrqContext irqContext,
  uint8_t           irqNumber
);
```

Arguments:

- `*irqChannel` structure containing the registers and settings for a digital IRQ I/O
- `irqContext` IRQ context to be reserved
- `irqNumber` the IRQ number (`IRQNO_MIN-IRQNO_MAX`)

Wait for Interrupt

`Irq_Wait()`	Wait until the specified IRQ number occurred or ready signal arrives. Declared in the `IRQConfigure.h` header file.

Prototype:

```
void Irq_Wait(
  NiFpga_IrqContext irqContext,
  NiFpga_Irq        irqNumber,
  uint32_t          *irqAssert,
  NiFpga_Bool       *continueWaiting
);
```

Arguments:

- `irqContext` context of current IRQ
- `irqNumber` IRQ number
- `continueWaiting` signal which aborts the waiting thread
- `return irqAssert` asserted IRQ number

This is a blocking function that stops the calling thread until the FPGA asserts any IRQ in the number parameter, or until the function call times out. The `irqsAssert` parameter can be used to determine which IRQs were asserted for each function call.

Acknowledge IRQ

`Irq_Acknowledge()`	Acknowledges an IRQ to the FPGA. Declared in the `IRQConfigure.h` header file.

Prototype:

```
void Irq_Acknowledge(
  uint32_t irqAssert
);
```

Arguments:

- `irqAssert` asserted IRQ number

Create POSIX thread

`pthread_create()`	Creates a new thread within a process. Declared in the `pthread.h` header file.

Prototype:

```
int pthread_create(
  pthread_t            *thread,
  const pthread_attr_t *attr,
  void                 *(*start_routine) (void *),
  void                 *arg
);
```

Arguments:

- `*thread` new thread identifier
- `*attr` new thread attributes (`NULL` - default)
- `*start_routine` starting function of new thread

- `*arg` sole argument of `start_routine`
- `return status = 0` for success

Join POSIX thread

`pthread_join()`	Suspends execution of the calling thread until the target thread terminates. Declared in the `pthread.h` header file.

Prototype:

```
int pthread_join(
  pthread_t thread,
  void      **retval
);
```

Arguments:

- `thread` thread identifier
- `*retval` if not `NULL`, copies the exit status into the location pointed to by `retval`
- `return status = 0` for success

Exit POSIX thread

`pthread_exit()`	Terminates the calling thread. Declared in the `pthread.h` header file.

Prototype:

```
void pthread_exit(
  void *retval
);
```

Arguments:

- `*retval` if not `NULL`, copies the exit status into the location pointed to by `retval`
- `return status = 0` for success

Part IV

Feedback Control of Mechanical Systems

06

Discrete dynamic systems

Control systems engineers frequently need to make a discrete embedded computer system behave like a single-input-single-output (SISO) dynamic system. The input and output for the continuous system are continuous functions of time. The corresponding input and output for a discrete dynamic system are signals sampled (Lec. 06.1) to form discrete time sequences, as shown in Fig. 06.1.

A continuous system can be described by a differential equation or transfer function that operate on and returns continuous signals; A discrete system can be described by a difference equation (Lec. 06.2) or discrete transfer function

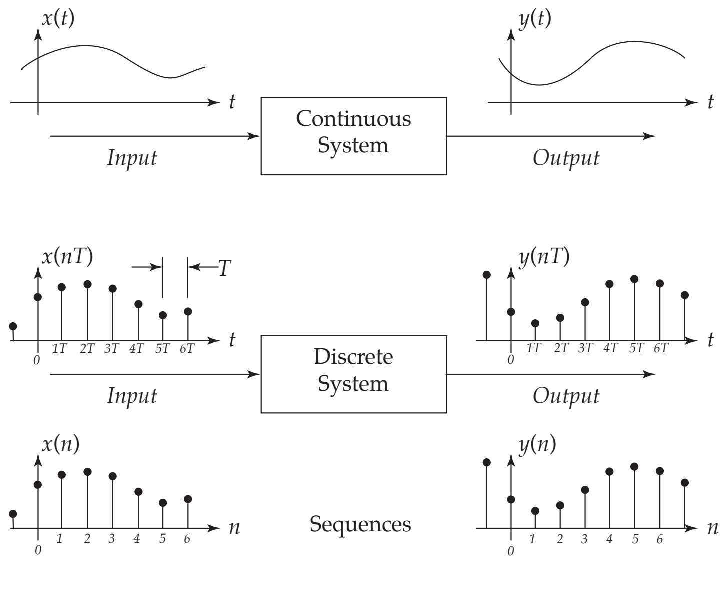


Figure 06.1: continuous systems, discrete systems, and sequences.

(Lec. 06.3) that operate on and returns sequences.
In addition to discrete system dynamics considerations, this chapter also introduces timer interrupts (Resource 12) to improve realtime performance. As an application of this material, in Lab Exercise 06, we will learn how to instantiate a dynamic system in our microcontroller.

06.1 Analog–to–digital and digital–to–analog conversion

Most sensors and actuators, which are necessary components for control systems, have as input and output analog signals. These are continuous in time and can take on a virtually infinite number of real values. In preceding chapters, we have learned that computers work with binary digital signals, which communicate information by changing voltage between conventional levels representing logical true and false. How can a binary digital signal represent an analog signal and vice versa?

Analog-to-digital conversion and analog outputs

Analog-to-digital conversion (ADC) is the process by which a binary digital signal is made to represent an analog signal. This proceeds in three operations, as illustrated in Fig. 06.1:

sampling the analog signal is sampled: measured at discerete moments in time, usually at a fixed sample period T (i.e. sample rate $1/T$ or angular sample rate $2\pi/T$);

quantization the sampled measurement is quantized: represented by one of a finite set of values limited by the number of bits available in binary conversion; and

binary conversion the quantized measurement is converted to binary: given a binary representation such that it can be represented by a binary digital signal.

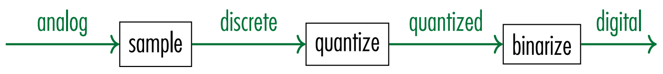


Figure 06.1: the operations required to convert an analog to a binary digital signal.

It is convenient to give a name to the measurement after it is sampled, but before it is quantized: a discrete signal,[1] which is represented mathematically as a sequence of real numbers paired with corresponding time intervals (usually fixed). These sequences are either denoted as functions of an integer n or its product nT with a fixed sample period T.[2]

1. Sometimes this is called a digital signal, but we reserve that term for (usually binary-) quantized signals.

2. These are distinct mathematical functions. Sometimes we call a function of integer n a sequence and a function of nT a discrete function. Even this obscures some details, but a precise mathematical treatment is unnecessary for our purposes.

A microcontroller, such as the myRIO, frequently has at least a few analog inputs (AIs): voltage measurement channels with ADC. The ADC takes time, so a lag is introduced in the process. Microcontrollers with an FPGA (like the myRIO) can frequently use it perform much faster ADC than those without.

For details about programming the myRIO analog inputs, see Resource 14.

Digital-to-analog conversion and analog outputs

Microcontrollers also commonly have at least a few analog outputs (AOs): channels that can programmatically produce an analog signal voltage in some range (usually something like $[0, 5]$ V or $[-10, +10]$ V). But, as we know, that continuous output once had a digital representation, so it can only construct a continuous representation of some finite number of voltage levels.

The process of converting a digital signal to an analog signal is called digital-to-analog conversion (DAC). It is essentially the opposite process as ADC; the key move is to convert a discrete signal to a continuous signal. Considered broadly, it is the approximation of a discrete function by a continuous one, which is called "curve fitting": an attempt to fill-in continuous values in the intermediate time between discrete samples.

There is a tradeoff here between accuracy and speed because curve fitting requires computation time. In embedded computing applications, it is frequently more-important to immediately represent the realtime output than to be accurate with intermediate approximations, especially if (as is usually the case) output resolution is sufficient. This suggests the ubiquitous method of the zero-order hold, which simply maintains the previous sample value throughout the intermediate sample period, yielding a step-like analog signal. While this introduces some high-frequency noise, it is usually the best option.

For details about programming the myRIO analog outputs, see Resource 14.

06.2 Difference equations

Many continuous dynamic systems can be described by a linear, constant-coefficient differential equation:

$$\alpha_n \frac{d^n y}{dt^n} + \alpha_{n-1} \frac{d^{n-1} y}{dt^{n-1}} + \ldots + \alpha_1 \frac{dy}{dt} + \alpha_0 y = \\ = \beta_m \frac{d^m x}{dt^m} + \beta_{m-1} \frac{d^{m-1} x}{dt^{m-1}} + \ldots + \beta_1 \frac{dx}{dt} + \beta_0 x \tag{1}$$

where α_k and β_k are constants.
The corresponding discrete system is described by a difference equation that operates on the sequence of input values $x(n)$ to produce the output sequence $y(n)$. The difference equation has the form

$$a_0 y(n) + a_1 y(n-1) + \ldots + a_N y(n-N) = \\ = b_0 x(n) + b_1 x(n-1) + \ldots + b_M x(n-M) \tag{2}$$

for $n = 0, 1, 2, \ldots$, where $x(n)$ is a sequence of periodically digitized values of the analog input signal, $y(n)$ is a sequence of values that determine the output signal, and a_k for $k = 0, 1, \ldots, N$ and b_k for $k = 0, 1, \ldots, M$ are constants.
This equation can also be written in summation form:

$$\sum_{k=0}^{N} a_k y(n-k) = \sum_{k=0}^{M} b_k x(n-k) \tag{3}$$

or, solving this for the current output sample $y(n)$,

$$y(n) = \frac{1}{a_0} \left[\sum_{k=0}^{M} b_k x(n-k) - \sum_{k=1}^{N} a_k y(n-k) \right] \tag{4}$$

Notice that the current output value $y(n)$ depends on previous values of y and on the previous and current values of the input x.
The problem of finding a discrete approximation of a continuous dynamic system

represented by the differential equation Eq. 1, then, is now just the problem of finding appropriate constants a_k and b_k in the difference equation such that its behavior approximates that of Eq. 1 with its constants α_k and β_k.
It turns out the best methods of approximation are derived not directly from the differential-difference equation relationship, but instead from the (implied) continuous-discrete transfer function relationship thereof. It is to the discrete transfer function that we therefore turn.

06.3 Discrete transfer functions

We begin with a review of Laplace transforms and continuous transfer functions.

Laplace transforms

In the analysis of this continuous systems, we use the Laplace transform, defined by

$$\mathcal{L}\left(f(t)\right) = \int_0^\infty f(t)e^{-st}dt \tag{1}$$

which leads directly to the familiar Laplace transform properties (1) of linearity and (2) of differentiation: the Laplace transform of the derivative of a function $f(t)$ (with zero initial conditions) is s times the transform of the function $F(s) \equiv \mathcal{L}(f(t))$:

$$\mathcal{L}\left(\frac{df(t)}{dt}\right) = sF(s). \tag{2}$$

Continuous transfer functions

These properties allow us to find the transfer function of a linear continuous system, given its differential equation. We define the continuous transfer function $T(s)$ to be the Laplace transform of the output $Y(s)$ divided by the Laplace transform of the input $X(s)$; i.e.

$$T(s) = \frac{Y(s)}{X(s)}. \tag{3}$$

Reconsider the continuous differential equation for a dynamic system Eq. 1. The equivalent transfer function, using the linearity and differentiation properties of the Laplace transform, is

$$T(s) = \frac{\beta_m s^m + \beta_{m-1} s^{m-1} + \cdots + \beta_1 s^1 + \beta_0}{\alpha_n s^n + \alpha_{n-1} s^{n-1} + \cdots + \alpha_1 s^1_+ \alpha_0} \tag{4}$$

where α_k and β_k are the same constants that appeared in Eq. 1.

z-Transforms

For discrete systems and their difference equations, a very similar procedure is available. The z-transform $F(z) \equiv \mathcal{Z}(f(n))$ of a sequence $f(n)$, with complex variable z (analogous to s), is defined by[3]

$$\mathcal{Z}(f(n)) = \sum_{n=0}^{\infty} f(n) z^{-n}. \tag{5}$$

3. There are many more uses for z-transforms. For more details, see Franklin, Powell and Workman (1998).

This leads directly to the z-transform properties (1) of linearity and (2) of delay, analogous to (2) for discrete systems: the z-transform of a function delayed by one sample period is z^{-1} times the transform of the function $F(z)$:

$$\mathcal{Z}(f(n-1)) = z^{-1} F(z), \tag{6}$$

Discrete transfer functions

We define the discrete transfer function $T(z)$ to be the z-transform of the output $Y(z)$ divided by the z-transform of the input $X(z)$; i.e.

$$T(z) = \frac{Y(z)}{X(z)}. \tag{7}$$

Given the z-transform properties, we can easily find the transfer function of a discrete system given its difference equation.

Example 06.3 −1 **re: discrete transfer function**

What is the discrete tranfer function corresponding to the second-order difference equation

$$\begin{aligned} &a_0 y(n) + a_1 y(n-1) + a_2 y(n-2) = \\ &= b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) \end{aligned} \tag{8}$$

with constants a_n and b_n?

The z-transform of the difference equation is determined by linearity and successively applying (6) to arrive at

$$\left(1 + a_1 z^{-1} + a_2 z^{-2}\right) Y(z) = \left(b_0 + b_1 z^{-1} + b_2 z^{-2}\right) X(z). \tag{9}$$

Rearranging, the discrete transfer function is

$$\frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{10}$$

Notice that the transfer function (10) and the difference equation (8), can be derived from each other by inspection. Notice also that the transfer function of a discrete system is the ratio of two polynomials in z, just as the transfer function of a continuous system is the ratio of two polynomials in s.

Discrete approximations of continuous transfer functions

There are several ways to derive an approximate discrete transfer function from a corresponding continuous transfer function. We will use a popular technique called Tustin's method that approximates a continuous function of time by straight lines connecting the sampled points (i.e. trapezoidal integration). The discrete transfer function is found using Tustin's method by making the following substitution:

$$s \mapsto \frac{2}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right) \tag{11}$$

and rewriting the transfer function in the form of equation (10). Here, T is the sample period.

Example 06.3 −2 **re: Tustin's method**

Consider a continuous first order system described by the transfer function:

$$\frac{Y(s)}{X(s)} = \frac{1}{\tau s + 1}, \text{ where } \tau \text{ is the time constant.} \tag{12}$$

Using Tustin's method, derive a discrete transfer function and the corresponding difference equation.

Substituting Equation 11 into the transfer function, we have:

$$\frac{Y(z)}{X(z)} = \frac{\alpha + \alpha z^{-1}}{1 - (1 - 2\alpha) z^{-1}},$$

where α is a constant:

$$\alpha = \frac{T}{2\tau + T}$$

from which the difference equation can be inferred (see Eqs. 8 to 10 above):

$$y(n) = (1 - 2\alpha) y(n-1) + \alpha x(n) + \alpha x(n-1)$$

Notice again that the current value of the output $y(n)$ depends on the previous output, $y(n-1)$, and on the current and previous inputs, $x(n)$ and $x(n-1)$.
Notice also that the coefficients depend on the time constant τ in the original continuous system and on the sample period T.
During each sample period, the value of the current value of the input $x(n)$ is measured and the current value of the output $y(n)$ is computed. Suppose that the time constant $\tau = 2$, the sample period $T = 1$, and that the input is a unit step ($x(n) = 1$ for all n), and the initial condition $y(0) = 0$.
Then, from our solution for $y(n)$,

$$y(n) = 0.6 y(n-1) + 0.4 \tag{13}$$

and we can compute the output sequence:

Figure 06.1 shows plots of the input and output sequences.

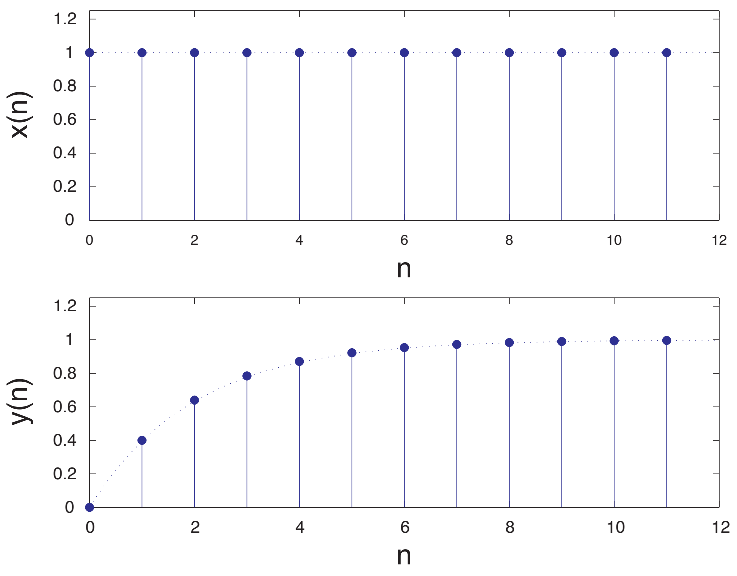


Figure 06.1: input and output sequences.

The dotted line is the exact solution $y(t/T)$ of the original continuous differential equation. As you can see, in this example, Tustin's method is very close to the exact solution at the sample points.

See Resource 13 for a table of common controller transfer functions converted to discrete transfer functions via Tustin's method.

Matlab's `c2d`

The Matlab Control Systems Toolbox includes a function `c2d` that computes the Tustin equivalent discrete system `sysd` from the continuous system `sys`, as follows.

```
sysd = c2d(sys, T, 'tustin')
```

This function can also use other common techniques to yield a discrete approximation of a continuous transfer function.

06.4 The biquad cascade

Although we could implement Eq. 4 as shown, the sensitivity of the output to the coefficients leads to numerical inaccuracies as the order of the system N becomes large. We will solve this problem by breaking the Nth order system it into a series of n_s second-order systems.
The technique is called a biquad cascade and is illustrated in Figure 06.1.
Notice that the output of each second-order section (biquad)[4] is the input to the subsequent section. Each biquad implements the same second-order difference equation, but with different coefficients, inputs, and outputs.
For example, the current output $y_i(n)$ from the ith section would be:

$$y_i(n) = \frac{1}{a_{0_i}}\big(b_{0_i}x_i(n) + b_{1_i}x_i(n-1) + b_{2_i}x_i(n-2) + \\ - a_{1_i}y_i(n-1) - a_{2_i}y_i(n-2)\big). \tag{1}$$

4. "Biquad" is short for "biquadratic." The biquad transfer function has second-order polynomials in both numerator and denominator.

Of course, a first or second order transfer function would require only one biquad.
Depending on the value of N, some of the coefficients of at least one biquad may be zero.
We will implement a function to handle any value of N.
There are a variety of algorithms for breaking a transfer function into biquadric sections.
Matlab's Signal Processing Toolbox contains a

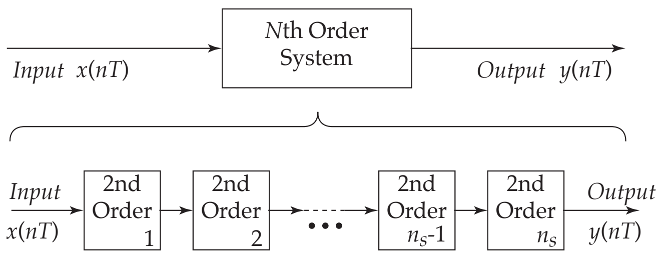


Figure 06.1: a biquad cascade.

function `tf2sos` (transfer function to second order sections) for this purpose.

Resource R12 Timer interrupts

This resource describes how to program the myRIO in C to perform timer interrupts.

Main thread: background

Initializing the timer interrupt is similar to initializing the digital input interrupt.
We will use a separate thread to produce interrupts at periodic intervals. Within `main`, we will configure the timer interrupt and create a new thread to respond when the interrupt occurs. The two threads communicate through a globally defined thread resource structure:

```
typedef struct {
  NiFpga_IrqContext irqContext; // IRQ context reserved
  NiFpga_Bool irqThreadRdy;     // IRQ thread ready flag
} ThreadResource;
```

National Instruments provides C functions to set up the timer interrupt request (IRQ).

Register the Timer IRQ

The first of these functions reserves the interrupt from the FPGA and configures the timer and IRQ. Its prototype is:

```
int32_t Irq_RegisterTimerIrq(
  MyRio_IrqTimer* irqChannel,
  NiFpga_IrqContext* irqContext,
  uint32_t timeout
);
```

where the three input arguments are:

1. `irqChannel`: A pointer to a structure containing the registers and settings for the IRQ I/O to modify; defined in `TimerIRQ.h` as:

```
typedef struct {
  uint32_t timerWrite;  // Timer IRQ interval register
  uint32_t timerSet;    // Timer IRQ setting register
  Irq_Channel timerChannel; // Timer IRQ supported I/O
} MyRio_IrqTimer;
```

2. `irqContext`: a pointer to a context variable identifying the interrupt to be reserved. It is the first component of the thread resources structure.
3. `timeout`: the timeout interval in μs.

The returned value is `0` for success.

Create the interrupt thread

A new thread must be configured to service the timer interrupt. In `main` we will use `pthread_create` to set up that thread. Its prototype is:

```
int pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg
);
```

where the four input arguments are:

1. `thread`: a pointer to a thread identifier.
2. `attr`: a pointer to thread attributes. In our case, use `NULL` to apply the default attributes.
3. `start_routine`: the name of the starting function in the new thread.
4. `arg`: the sole argument to be passed to the new thread. In our case, it will be a pointer to the thread resource structure defined above and used in the second argument of `Irq_RegisterDiIrq`.

This function also returns `0` for success.

Main thread: our case

We can combine these ideas into a portion of the `main` code needed to initialize the Timer IRQ.[5]

5. The IRQ settings symbols associated with the timer interrupt, are defined in the header file: `TimerIRQ.h`.

For interrupts triggered by the timer in the FPGA, we have:

```
int32_t status;
MyRio_IrqTimer irqTimer0;
ThreadResource irqThread0;
pthread_t thread;

// Registers corresponding to the IRQ channel
irqTimer0.timerWrite = IRQTIMERWRITE;
irqTimer0.timerSet = IRQTIMERSETTIME;
timeoutValue = 5;
status = Irq_RegisterTimerIrq(
  &irqTimer0,
  &irqThread0.irqContext,
  timeoutValue
);

// Set the indicator to allow the new thread.
irqThread0.irqThreadRdy = NiFpga_True;

// Create new thread to catch the IRQ.
status = pthread_create(
  &thread,
  NULL,
  Timer_Irq_Thread,
  &irqThread0
);
```

Other `main` tasks go here.
After the tasks of `main` are completed, it should signal the new thread to terminate by setting the `irqThreadRdy` flag in the `ThreadResource` structure. Then it should wait for the thread to terminate. For example,

```
irqThread0.irqThreadRdy = NiFpga_False;
status = pthread_join(thread, NULL);
```

Finally, the timer interrupt must be unregistered:

```
status = Irq_UnregisterTimerIrq(
  &irqTimer0,
  irqThread0.irqContext
);
```

using the same arguments from above.

The interrupt thread

This is the separate thread that was named and started by the `pthread_create` function. Its overall task is to perform any necessary function in response to the interrupt. This thread will run until signaled to stop by `main`.
The new thread is the starting routine specified in the `pthread_create` function called in `main`. In our case:
`void *Timer_Irq_Thread(void* resource)`.
The first step in `Timer_Irq_Thread` is to cast its input argument (passed as `void *`) into appropriate form. In our case, we cast the `resource` argument back to a `ThreadResource` structure. For example, declare

```
ThreadResource* threadResource =
  (ThreadResource*) resource;
```

The second step is to enter a `while` loop. Two functions are performed each time through the loop, as described in Algorithm 06.1.

Algorithm 06.1 ISR thread loop pseudocode

```
while the main thread has not signaled this thread to stop do
    wait for the occurrence (or timeout) of the IRQ
    schedule the next interrupt
    if the Timer IRQ has been asserted then
        perform operations to service the interrupt
        acknowledge the interrupt
    end if
end while
```

The `while` loop should continue until the `irqThreadRdy` flag (set in `main`) indicates that the thread should end. For example,

1. Use the `Irq_Wait` function to pause the loop while waiting for the interrupt. For our case the call might be, with `TIMERIRQNO` a constant defining the Timer

IRQ's IRQ number, defined in `TimerIRQ.h`:

```
uint32_t irqAssert = 0;
Irq_Wait(
  threadResource->irqContext,
  TIMERIRQNO,
  &irqAssert,
  (NiFpga_Bool*) &(threadResource->irqThreadRdy)
);
```

Notice that it receives the `ThreadResource` context and Timer IRQ number information, and returns the `irqThreadRdy` flag set in the `main` thread. Schedule the next interrupt by writing the time interval into the `IRQTIMERWRITE` register, and setting the `IRQTIMERSETTIME` flag. That is,

```
NiFpga_WriteU32(
  myrio_session,
  IRQTIMERWRITE,
  timeoutValue
);
NiFpga_WriteBool(
  myrio_session,
  IRQTIMERSETTIME,
  NiFpga_True
);
```

The `timeoutValue` is the number of μs (`uint32_t`) until the next interrupt. The `myrio_session` used in these functions should be declared within this timer thread. That is,

```
extern NiFpga_Session myrio_session;
```

This variable was defined when you called `MyRio_Open` in the `main` thread.

2. Because the `Irq_Wait` times out after 100 ms, we must check the `irqAssert` flag to see if the Timer IRQ has been asserted. In addition, after the interrupt is serviced, it must be acknowledged to the scheduler. For example,

```
if(irqAssert & (1 << TIMERIRQNO)) { // Bit mask
  // Your interrupt service code here
  Irq_Acknowledge(irqAssert);
}
```

In the third step (after the end of the loop) we terminate the new thread, and return from the function:

```
pthread_exit(NULL);
return NULL;
```

06.exe Exercises for Chapter 06

06.L Lab Exercise: Transfer function generator

Objectives

The objectives of this exercise are to:

1. Use real-time clock interrupts to provide timing.
2. Implement an arbitrary transfer function generator.
3. Introduce A/D and D/A conversion.

Introduction

In this lab exercise, you will write a general purpose program capable of approximating the performance of any SISO, LTI system! The system input and output will both be analog electrical signals. Your program will implement this with a difference equation.
At the beginning of each BTI, your ISR will read an analog input to obtain the current input value, compute the current value of the output $y(n)$, and apply the current output value to an analog output.
This process continues until ← is entered on the keypad. The input voltage will be provided by a function generator. Both the input and output voltages will be displayed on the oscilloscope.
You will use three new myRIO features in this experiment: an interrupt timer, the ADC, and the DAC. The first is described in detail in Resource 12 and the others in Resource 14.
Although we could implement the difference equation Eq. 4 as shown, the sensitivity of the output to the coefficients leads to numerical inaccuracies as the order of the system N becomes large, so we use the biquad cascade representation of Lec. 06.4 .

Pre-laboratory preparation

The program consists of a `main` function and an interrupt service routine (ISR) running in a separate thread. The ISR is set to execute with a period of 0.5 ms (determined by the Timer IRQ), and computes the DAC output from the ADC input by means of a difference equation.

Main program

The only tasks of `main` will be the following.

1. Set up and enable the Timer IRQ interrupt,
2. Enter a loop, ending only when a ← is received from the keypad. Use `getkey`.
3. Signal the timer thread to terminate using the `irqThreadRdy` flag, and wait for it to terminate.

Interrupt service routine

The interrupt service routine thread implements a dynamic system. The heart of the ISR is a while loop that checks the `irqThreadRdy` flag (set in `main`) to see if the thread should continue. Before the loop begins, initialize the analog input/output, and set the analog output to 0 V. Each time through the loop:

1. Get ready for the next interrupt by waiting for the IRQ to assert. Then write the time interval to wait between interrupts (BTI) to the `IRQTIMERWRITE` register and write `TRUE` to the `IRQTIMERSETTIME` register.
2. Read the analog input to obtain the current input value $x(n)$.
3. Call a function `cascade` (see below) to calculate the current value of the output $y(n)$ by computing all of the sections in the biquad cascade. Each biquad section is computed according to Eq. 1.
4. Send the output value to the analog output.

5. Acknowledge the interrupt.

After the loop terminates, save the response to `Lab6.mat`.
The ISR must allocate storage for variables and arrays associated with the discrete dynamic system, including:

1. the length of the BTI in microseconds,
2. the number of biquad sections n_s, and
3. the system constants (a_i and b_i) for the biquad sections.

The dynamic system corresponding to the collection of biquad sections can be conveniently referred to and manipulated by first defining a structure to contain the coefficients and previous values of input and output for a single biquad section. We might define a "biquad" structure as follows.

```
struct biquad {
  double  b0; double  b1; double  b2; // numerator
  double  a0; double  a1; double  a2; // denominator
  double  x0; double  x1; double  x2; // input
  double  y1; double  y2;             // output
};
```

This definition should be placed just before the prototypes section of your program.
Then, a specific dynamic system can be defined as an array of these biquad structures, each array element corresponds to an individual biquad section:

```
int myFilter_ns = 2;      // No. of sections
uint32_t timeoutValue = 500;  // T - us; f_s = 2000 Hz
static struct biquad myFilter[] = {
  {1.0000e+00,  9.9999e-01, 0.0000e+00,
   1.0000e+00, -8.8177e-01, 0.0000e+00, 0, 0, 0, 0, 0},
  {2.1878e-04,  4.3755e-04, 2.1878e-04,
   1.0000e+00, -1.8674e+00, 8.8220e-01, 0, 0, 0, 0, 0}
};
```

This system description can be placed within the ISR, near its beginning. The first two lines

establish the number of biquad sections, and the length of the BTI in microseconds. Finally, `myFilter` is the name of an array of biquad structures being initialized.
For testing purposes, the initialized constants in the example above correspond to a system of two biquad sections ($n_s = 2$), encoding a unity-gain low-pass filter, with sampling frequency of 2000 Hz. Derived using Tustin's method, they correspond to a third-order continuous system having a pair of complex poles with natural frequency of 40 Hz, and with damping ratio 0.5. The remaining real pole is at 40 Hz.

Crazy about pointers!

The most challenging part of this task is the calculation of the current output value $y(n)$. The use of pointers makes the calculation both straightforward and efficient.

Box 06.1 hint

Don't be tempted to code this algorithm using array indices (instead of pointers); that would be much too slow for our purposes.

The *cascade* function

The `cascade` function implements the complete dynamic system by passing the measured input through the string of biquad sections. The ISR must pass to `cascade` the value of the current input $x(n)$ measured by the ADC, the number of biquad sections n_s, the array of biquad structures containing the coefficients and history variables (x_i and y_i) for all sections. It might have a prototype that looks like:

```
double cascade(
  double xin,          // input
  struct biquad *fa,  // biquad array
  int    ns,          // no. segments
  double ymin,        // min output
  double ymax         // max output
);
```

Here, `xin` is the current system input, `fa` is the name of an array of biquad structures, `ns` is the corresponding number of biquad sections, and `ymin` and `ymax` are the saturation limits.
In the above example, `myFilter` would be passed through `fa`. The value returned by `cascade` is the current value of the system output $y(n)$.

Coding *cascade*

An efficient way to code `cascade` is to allocate a pointer `f` in `cascade` that will be used to point to elements of the array of biquad structures.
Begin the function by equating the pointer to the first element in the array (i.e. the first biquad): `f = fa;`. Variables inside the biquad structure are accessed by using the pointer name, e. g. `f->a0`, `f->b0`, `f->x0`, `f->y1`, etc. (The `->` operator is equivalent to dereferencing and then accessing a member (say, `(*f).a0`) and is typed as a minus sign immediately followed by >.)
Then, loop n_s times, to cycle through each of the biquad sections in the array. At the beginning of each loop, the output value `y0` of previous biquad must be passed to the input value `f->x0` of the current biquad.
Within the loop, coding the output value `y0` might look like:

```
y0 = (
  f->b0*f->x0 + f->b1*f->x1 + // ... etc.
)/f->a0;
```

See Equation 1.
Each time through the loop, after the output value has been computed, the previous values x

and y must be updated, so that they will be correct at the next time step. For example,

```
f->x2 = f->x1; f->x1 = f->x0; // ... etc.
```

At the end of the loop, the pointer `f` is incremented to advance to the next biquad in the array.
One more point: if the DAC is given a value beyond its range $[-10, +10]$ V, it will saturate its output value appropriately. However, our difference equation Eq. 1 depends on previous values of the output, but doesn't saturate. To correct this disparity, `cascade` should saturate the output `y0` of the final biquad before it is saved for the next iteration.
For example, define the macro

```
#define SATURATE(x,lo,hi) { \
  ((x) < (lo) ? (lo) : (x) > (hi) ? (hi) : (x)) \
}
```

Pass appropriate values of the `xmin` and `xmax` parameters to `cascade`. Then, for the last biquad, immediately after `y0` is computed, saturate its value:
`y0 = SATURATE(y0, ymin, ymax);`

Laboratory Procedure

A good strategy to follow in writing this program is to first implement and debug everything except the calculation of the biquad cascade. That is, set up the `main` program and the ISR, including all arrays and timing. In the ISR, simply pass the input value from the ADC directly to the DAC. For example,

```
VADin = Aio_Read(&CI0);
Aio_Write(&CO0, VADin);
```

This will allow you to observe the input and output on the oscilloscope, and determine if the interrupt timing is functioning properly.

When you have debugged those portions of the program augment the code above with the call to `cascade`.

Does it work?

The low-pass digital filter described above was derived using Tustin's method from the transfer function of the three-pole continuous system:

$$\frac{V_{out}(s)}{V_{in}(s)} = \frac{\omega_n^3}{(s+\omega_n)(s^2+2\zeta\omega_n s+\omega_n^2)} \qquad (1)$$

where $\omega_n = 2\pi \times 40$ rad/s, and $\zeta = 0.5$. This system belongs to a class of filters called Butterworth filters. They are signal processing filters designed to have the flattest possible frequency response in the passband.

Step response

Using the oscilloscope (DC coupled), observe the step response of the system by applying a low frequency square wave (e.g. at 8 Hz) with an amplitude of 5 V as the input with the function generator.
Save the input and output of `cascade` in 500-point buffers. After the timer loop ends, save the buffers to `Lab6.mat`, and transfer the data to Matlab. Plot and compare the measured step response to the theoretical response of the corresponding continuous system.[6]. Explain.

6. To simulate the theoretical response, Matlab's `lsim` is a good choice.

Frequency response

Again, using the oscilloscope (AC coupled), observe the frequency response by altering the frequency of a 5 V input sine wave.
Record (write down) the amplitude and phase of the output relative to the input sine wave at the following frequencies:
$[5, 10, 20, 40, 60, 100, 140, 200]$ Hz. Given the input amplitude, compute the transfer function magnitude (dB) at each frequency.

In Matlab, plot the theoretical magnitude (dB) and phase (deg) versus the frequency (Hz on a logarithmic scale) for the continuous system transfer function.[7] Plot the corresponding measured data as discrete symbols on top of the theoretical frequency response. Explain.

7. To generate the data for this plot, Matlab's `bode` is a good choice. Note that the specifications for the plot format require you to generate the plot separately from your call to `bode`.

Resource R13 Discrete−time controllers

For reference, Table 06.1 contains Tustin equivalents for some common continuous-time controllers.

Table 06.1: Tustin equivalents for common continuous-time controllers. Usage of z is contextual, meaning a zero in continuous transfer functions and meaning the z-transform z in discrete transfer functions.

	phase lag/lead	PI	PID
continuous	$k\frac{s+z}{s+p}$	$K_p + \frac{K_i}{s}$	$K_p + \frac{K_i}{s} + K_d s$
discrete	$k\frac{b_0 + b_1 z^{-1}}{a_0 + a_1 z^{-1}}$	$\frac{b_0 + b_1 z^{-1}}{a_0 + a_1 z^{-1}}$	$\frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}$
differential equation	$\frac{dy}{dt} + py = k\left(\frac{dx}{dt} + zx\right)$	$y = K_p x + K_i \int_0^t x\,dt$	$y = K_p x + K_i \int_0^t x\,dt + K_d \frac{dx}{dt}$
difference equation	$y(n) = -\frac{a_1}{a_0} y(n-1) + \frac{b_0}{a_0} x(n) + \frac{b_1}{a_0} x(n-1)$	$y(n) = -\frac{a_1}{a_0} y(n-1) + \frac{b_0}{a_0} x(n) + \frac{b_1}{a_0} x(n-1)$	$y(n) = -\frac{a_1}{a_0} y(n-1) - \frac{a_2}{a_0} y(n-2) + \frac{b_0}{a_0} x(n) + \frac{b_1}{a_0} x(n-1) + \frac{b_2}{a_0} x(n-2)$
a_0	1	1	1
a_1	$(pT-2)/(pT+2)$	-1	0
a_2			-1
b_0	$k(zT+2)/(pT+2)$	$K_p + K_i T/2$	$K_p + K_i T/2 + 2K_d/T$
b_1	$k(zT-2)/(pT+2)$	$-K_p + K_i T/2$	$K_i T - 4K_d/T$
b_2			$-K_p + K_i T/2 + 2K_d/T$

Resource R14 Analog input and output

Analog initialization

For our project, we will use the analog input channel `CI0` and the analog output channel `CO0` on Connector C. They communicate with the processor through the FPGA.
Before they can be used, they must be initialized using

```
AIO_initialize(&CI0, &CO0);
```

Call it once, where `CI0` and `CO0` are structures that must be of type `MyRio_Aio`. This initialization function is included in the `me477` library.

Analog-to-digital converter

The single-channel 12-bit analog-to-digital converter (ADC) measures the current value of the applied voltage in the range $[-10.000, +9.995]$ V. Voltages outside that range saturate the conversion as shown in Figure 06.1. The ADC has a resolution of 4.883 mV, with absolute accuracy of ± 200 mV. Each channel has input impedance of > 500 kΩ and overload protection of ± 16 V.
Our library contains a function that reads a specified channel of the ADC and returns the converted value. Its prototype is:

```
double Aio_Read(MyRio_Aio* channel);
```

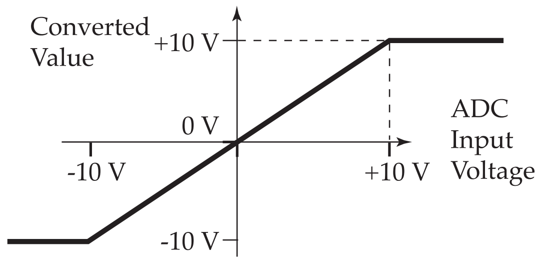


Figure 06.1: ADC saturation.

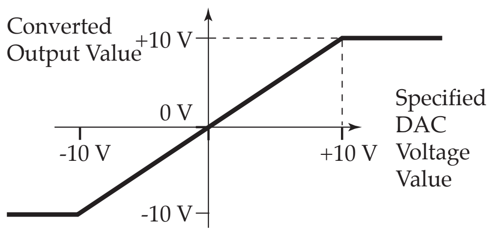


Figure 06.2: DAC saturation.

where `channel` is the pointer to the channel structure defined above: `&CI0`.

Digital-to-analog converter

The single-channel 12-bit digital-to-analog converter (DAC) produces a voltage at the output terminal in the range $[-10.000, +9.995]$ V. Again, specified voltages outside that range saturate the conversion as shown in Figure 06.2. The DAC has a resolution of 4.883 mV, with absolute accuracy ± 200 mV. Each channel has a maximum drive current of 3 mA, a maximum slew rate of 2 V/µs, and an overload protection of ± 16 V.
Our library contains a function that accepts a specified channel for the DAC, and returns the converted value. Its prototype is:

```
void Aio_Write(MyRio_Aio* channel, double value);
```

where `channel` is the pointer to the channel structure defined above: `&CO0` and `value` is the specified value of the analog output voltage.

07

Closed–loop control

07.1 DC motor velocity control

07.2 Designing a PI controller

07.exe Exercises for Chapter 07

07.L Lab Exercise: DC motor PI velocity control

Objectives

The objectives of this exercise are to:

1. Incorporate many of the hardware and software elements developed previously in this course into an integrated closed-loop control system.
2. Implement a program structure allowing continuous modification of the control parameters without halting the control algorithm.
3. Implement a proportional-integral (PI) velocity control algorithm for the DC motor.

Introduction

In this exercise, a closed-loop control system for the DC motor will be developed. This system is similar to actuators used in many types of positioning systems. The primary drive of one axis of an automated machine tool or of one axis of motion of an industrial robot is often a computer-controlled DC motor.

Our system will control the motor speed. The control algorithm will repeatedly compare the actual velocity of the motor V_{act} with the desired reference velocity V_{ref}, and automatically alter the applied control voltage to correct any differences. Although this is not a trivial computer control task, you have developed nearly all the required elements in the preceding six exercises.

The optical encoder (through the FPGA), the D/A converter (connected to the motor amplifier), and the periodic timer interrupt, will be combined to control the DC motor. As in Lab Exercise 06, a separate timer thread will produce an interrupt at the end of each basic

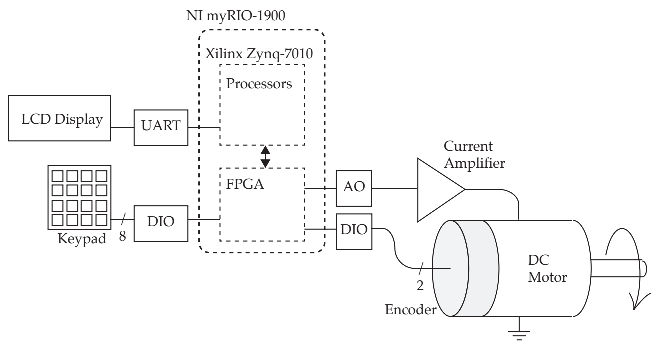


Figure L.1:

time interval (BTI). The ISR will load the `IRQTIMERWRITE` and `IRQTIMERSETTIME` registers to schedule the next BTI, and then call functions to:

1. read the encoder counter and compute the velocity,
2. execute the motor control algorithm, and
3. save the results, as necessary.

The control system will be "table-driven." That is, the parameters used by the control algorithm (reference speed, system gains, and BTI length) will be kept in a special table of values. Through the keypad/LCD, the values of the parameters in the table will be altered (interactively) by a "table editor" function called from the main program thread. The only tasks of the table editor will be to change the table values in response to commands from the keypad, and to display performance information.

This table-driven structure will allow the program user to change any of the control parameters, at any time, without stopping the execution of the control algorithm. It will appear as though two programs, the table editor and the control algorithm, are executing simultaneously.

You will not write the table editor. The required function, `ctable2` is described in Resource 15. It

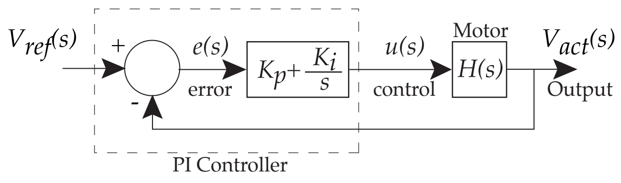


Figure L.2: continuous representation of the control loop.

has been included in our library, and is automatically linked with your program. Although you will not write this function, it uses the basic keypad/display algorithms that you developed in Lab Exercises 01, 02 and 03. The prototype for `ctable2` is in `ctable2.h`. The motor will be controlled using a proportional-plus-integral (PI) control law as shown in Figure L.2. The PI control law relates the error $e(t)$ to the output control signal $u(t)$ using the gain constants K_p and K_i. Applying Tustin's method to the continuous controller transfer function $K_p + \frac{K_i}{s}$, the corresponding discrete transfer function is

$$\frac{U(z)}{E(z)} = \frac{b_0 + b_1 z^{-1}}{a_0 + a_1 z^{-1}}, \tag{1}$$

where

$$a_0 = 1, \quad a_1 = -1, \quad b_0 = K_p + \frac{1}{2}K_i T, \text{ and } \quad b_1 = -K_p + \frac{1}{2}K_i T. \tag{2}$$

where T is the sample time, and the error is

$$e(n) = V_{ref}(n) - V_{act}(n)$$

For more on Tustin's method, see Lec. 06.3 . You will implement the corresponding difference equation using the general-purpose algorithm you developed in Lab Exercise 06.

Pre-laboratory preparation

Drawing on your previous work, write two threads to: (1) communicate with the user and (2) control the motor.

Two threads

Main program thread The main program performs these tasks:

1. Initialize the table editor variables.
2. Set up and enable the timer IRQ interrupt (as in Lab Exercise 06).
3. As in Lab Exercise 06, register the Timer Thread and create the thread to catch the Timer Interrupt. In this lab, the Timer Thread will gain access to the table data through a pointer. Modify the Timer Thread resource to include a pointer to the table. For example,

```
typedef struct {
  NiFpga_IrqContext irqContext; // context
  table *a_table;               // table
  NiFpga_Bool irqThreadRdy;     // ready flag
} ThreadResource;
```

4. Call the table editor. The table should contain six values, labeled as shown:

`V_ref: rpm`	`{edit}`
`V_act: rpm`	`{show}`
`VDAout: mV`	`{show}`
`Kp: V-s/r`	`{edit}`
`Ki: V/r`	`{edit}`
`BTI: ms`	`{edit}`

   All of the table edit values should be initialized to zero, except for the BTI length, which should be 5 ms. Note the units.
   After the main program calls the table editor, the user may edit and view the table values whenever desired.
5. When the table editor exits, signal the Timer Thread to terminate. Wait for it to terminate.

Timer thread – ISR At the beginning of the starting function, declare convenient names for the table entries from the table pointer:

```
double *vref = &((threadResource->a_table+0)->value);
double *vact = &((threadResource->a_table+1)->value);
// ...etc.
```

As in Lab Exercise 06, the Timer Thread includes a main loop timed by the IRQ, and is terminated only by its ready flag.
Before the loop begins:

- initialize the analog I/0, and set the motor voltage to zero, using `Aio_Write` (as is Lab Exercise 06).
- Set up the encoder counter interface (as in Lab Exercise 04).

Each time through the loop of the it should:

1. Get ready for the next interrupt by: waiting for the IRQ to assert, writing the Timer Write Register, and writing `TRUE` to the Timer Set Time Register.
2. Call `vel`, from Lab Exercise 04, to measure the velocity of the motor.
3. Compute the current coefficients (a's and b's) for the PI control law from the current values of K_p and K_i. See Equation 1. Update the values of the `biquad` structure.
4. Compute the current error $V_{ref} - V_{act}$.
5. Call `cascade` to compute the control value from the current error using the difference equation for PI control law. Important: limit the computed control value to the range $[-7.5, 7.5]$ V.
6. Send the control value to the D/A converter `C00` using `Aio_Write`.
7. Change the show values in the table to reflect the current conditions of the controller.
8. Save the results of this BTI for later analysis. (See below.)
9. Acknowledge the interrupt.

Functions

`cascade` – The `cascade` function, called once, from the ISR, during each BTI, implements the general-purpose linear difference equation algorithm from Lab 06. For this lab use the same C code that you used in Lab Exercise 04. In this case, the number of biquad sections will be one. Note that, as in Lab Exercise 06, all calculations should be made in (`double`) floating-point arithmetic.
`vel` – Use the `vel` function, developed in Lab Exercise 04, to read the encoder counter and estimate the angular velocity in units of BDI/BTI.

Saving the Responses

A convenient method of saving the data is to define data arrays in the ISR for both the velocity and the torque. Then an auto-incremented index variable is used to store the data in the arrays during each BTI. Increment the index as needed, stopping when index reaches the length of the arrays. A convenient length would be 250 points each. Since our program runs continuously, you may wish to save the response whenever the reference velocity is changed. This is easily accomplished by checking to see if the reference velocity has changed since the last BTI, and resetting the index to zero if it has. Since the index is then less than the length of the arrays, the arrays will be refilled. This is equivalent to recording the response to a step input in the reference velocity.
In addition, when the ISR resets the index to zero, save the previous value of the reference velocity. That value, along with other system parameters, will be used in MATLAB to predict the theoretical model response.

Table L.1: the base set of controller parameters.

V_{ref}	± 200	rpm
BTI length	5	ms
K_p	0.1	V-s/rad
K_i	2.0	V/rad

After the main loop terminates, but while still in the Timer Thread, write the results to the `Lab7.mat` file. The results should include:

1. your name (string),
2. the actual velocity array (rad/s),
3. the torque array (N-m),
4. the current and the last, previous reference velocities (rad/s),
5. K_p (V-s/rad),
6. K_i (V/rad), and
7. BTI length (s).

Use the same methods as Lab Exercise 04 and Lab Exercise 06 to bring the `Lab7.mat` file to Matlab.

Laboratory Procedure

1. Test and debug your program.
   For debugging purposes, use the base set of controller parameters in Table L.1.
2. While the motor is at steady-state speed, gently apply a steady load torque to the motor shaft. What are the responses of the actual speed and control voltage? Explain.
3. Beginning with the base set of parameters, explore the effect of varying the proportional gain K_p on the transient response. Try small (0.05) and large (0.2) values of K_p. What are the effects on the oscillation frequency and on the damping? Explain in terms of the transfer function parameters.
4. Beginning with the base set of parameters, explore the effect of varying the integral

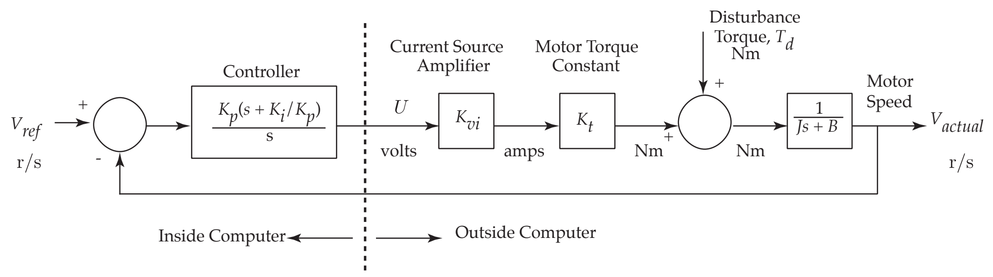


Figure L.3: continuous version of the full control loop.

gain K_i on the transient responses. Try small (1) and large (10) values of K_i. What are the effects on the oscillation frequency and on the damping? Explain in terms of the transfer function parameters.

5. Finally, using the base set of parameters, record the control torque and actual velocity responses for a step change in the reference velocity that starts from −200 rpm and goes to +200 rpm.
   In Matlab, compare these experimental responses with the analytical responses for the continuous system approximation. The theoretical responses can be calculated using the (appropriately scaled) Matlab `step` command. The analysis should be plotted over the experimental responses. Use the `subplot` command to place both the control value and the measured velocity plots on the same page. What do you conclude?

DC Motor Controller Model

The model in Figure L.3 is the continuous approximation of the actually discrete control of the DC motor.
For accuracy of the approximation, we need the following:

1. the sampling frequency is much larger than the natural frequency of the system

Table L.2:

K_{vi}	Current Source Amplifier Gain	0.41	A/volt
K_t	Motor Torque Constant	0.11	N-m/A
J	Inertia in conventional units	3.8×10^{-4}	N-m-s^2/r
K_p	Proportional Gain	—	V-s/r
K_i	Integral Gain	—	V/r

and

2. time delays caused by computation are insignificant and
3. the control value does not saturate and
4. the mechanical damping B is small in comparison with the effects of the proportional term K_p in the controller.

Parameter values are shown in Table L.2.
The following transfer functions can be obtained from the block diagram:

$$\frac{V_{act}(s)}{V_{ref}(s)} = \frac{\tau s + 1}{\frac{s^2}{{\omega_n}^2} + \frac{2\zeta}{\omega_n}s + 1}, \tag{3}$$

$$\frac{V_{act}(s)}{T_d(s)} = \frac{sK_d}{\frac{s^2}{{\omega_n}^2} + \frac{2\zeta}{\omega_n}s + 1}, \text{ and} \tag{4}$$

$$\frac{U(s)}{V_{ref}(s)} = \frac{sK_u(\tau s + 1)}{\frac{s^2}{{\omega_n}^2} + \frac{2\zeta}{\omega_n}s + 1}. \tag{5}$$

Let $K = K_{vi}K_t$ N-m/V; then the following values can be used to model the system:

numerator values:	$\tau = \frac{K_p}{K_i}$	s,
	$K_d = \frac{1}{K_i K}$	rad/N-m,
	$K_u = \frac{J}{K}$	V-s^2/rad
natural frequency:	$\omega_n = \sqrt{\frac{K_i K}{J}}$	rad/s
damping ratio:	$\zeta = \frac{K_p}{2}\sqrt{\frac{K}{JK_i}}$.	

Resource R15 A table editor for the myRIO

The following describes `ctable2()`, a utility program that displays values that are stored in memory, and allows the user to change selected values. The values, with appropriate labels, appear on the LCD display. The user enters values on the keypad.

When `ctable2()` is called, it then runs continually, returning to the calling program only when ← is entered. However, other threads may use and cause to be displayed the information stored by `ctable2()`.

A table "title" is displayed on the first line of the LCD display. The table can have as many as nine numbered entries. Three of these entries are always displayed below the title. The user can scroll the entries up and down using the `UP` and `DWN` keys. Alternately, the user can cause any entry to become the top entry by entering its number.

For example, a three-entry table, shown with the third entry scrolled to the top, might look like:

```
Flow Control Table
3 BTI: ms            3.0
1 Qref: (cc/s)      450.
2 Qact: (cc/s)      453.
```

The user may alter an entry by scrolling it to the top of the list, and pressing ENTR. The display prompts for a new value of the parameter. For the example above, pressing ENTR would cause the prompt: `Enter: BTI: ms` to be displayed. The user could then enter a new value (followed by `ENTR`), causing the new value to be placed in memory and displayed.

"Edit" values and "Show" values

There are two kinds of values, called "edit" values and "show" values. Edit values are those that the user may change at will. Each edit value is presumed not to have changed since the last time it was changed (edited) by the user.

Show values are those that the user may observe more or less continually. A separate thread, created within `ctable2()`, periodically updates the table to reflect the current show values. Show values may not be edited; each show value is presumed perhaps to have changed since the last time the table was updated. (The changes would generally be made by another thread, which would determine a new show value and place it in memory; the new value would then be displayed when the table is updated.)
Typically, edit values are system parameters set by the user, while show values are computed and change with time.

Calling ctable2()
The prototype of `ctable2()` is:

```
int ctable2(char *title, struct table *entries, int nval);
```

The `ctable2()` function is automatically linked with your code from the ME477Library. The statement: `#include "ctable2.h"` must appear in `main.c`.
When calling `ctable2()`, your program must supply appropriate values for the following arguments:

title is a string array for the table title. Less than 20 characters.

entries is an array of structures of type `table` defined as:

```
typedef struct {
    char *e_label;  // entry label label
    int e_type;     // entry type (0-show; 1-edit)
    double value;   // value
} table;
```

Each element of the array corresponds to an entry in the table, and specifies the entry label, type (edit or show), and value of the entry. A good practice is to make the length of the labels 12 characters or less.

nval specifies the number of table entries. Again, the total number of edit and show entries must be no greater than 9.

Entering ← while the table is displayed causes `ctable2()` to terminate, returning `0` for a normal exit.
For example,
In this table entitled: `Flow Control Table`, there are two edit values that can be changed by the user (`qref` and `bti`), and one show value (`qact`).
In the main thread, the variables for the table title, and the table structure array are declared and initialized.

```
char *Table_Title ="Flow Control Table";
table my_table[] = {
    {"Qref: (cc/s)",  1, 0.   },
    {"Qact: (cc/s)",  0, 0.0  },
    {"BTI: ms     ",  1, 5.0  }
};
```

Notice that the each element of the array `my_table` is a `struct` of type `table` containing the entry label, type, and initial value.

Finally, the table editor is called:

```
ctable2(Table_Title, my_table, 3);
```

Within the thread that uses or changes the table values, pointers corresponding to convenient names of the table values can be declared. In the example:

```
  double *qref = &((threadResource->a_table+0)->value);
  double *qact = &((threadResource->a_table+1)->value);
  double *bti  = &((threadResource->a_table+2)->value);
```

Then, variables may be referred to by their named pointers. For example,

```
T = *bti/1000.;
```

Note the dereferencing of the `bti` pointer.

08 Path planning

08.1 Path planning

08.2 Designing a PID controller

08.exe Exercises for Chapter 08

08.L Lab Exercise: DC motor PID position control

Objectives

The objectives of this exercise are to:

1. implement a position control system for an inertia dominated load,
2. explore appropriate path planning, and
3. integrate the use of a standard Matlab design tool into the application development system.

Introduction

In this exercise, a closed-loop position control system for the DC motor will be developed. The physical system is identical to that of Lab Exercise 07: as shown in Figure L.1, the optical encoder (through the FPGA), the D/A converter (connected to the motor amplifier), and the periodic timer interrupt will be combined to control the DC motor.
A Matlab tool will be used to design an appropriate proportional-integral-derivative (PID) controller, shown in Figure L.2. Later, you will evaluate the controller performance for a time-varying position reference path $x_{\text{ref}}(t)$.
This project builds on your past work. The program is structurally similar to that of Lab

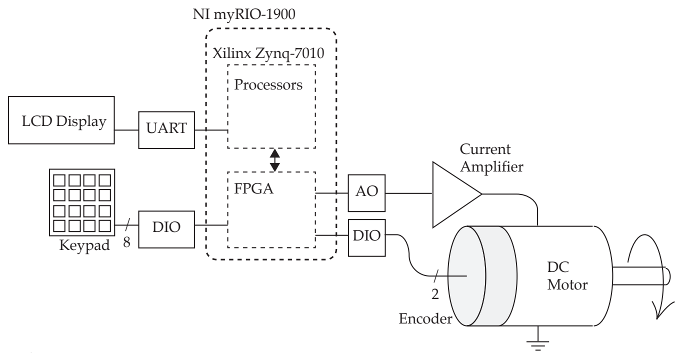


Figure L.1: schematic of the test apparatus.

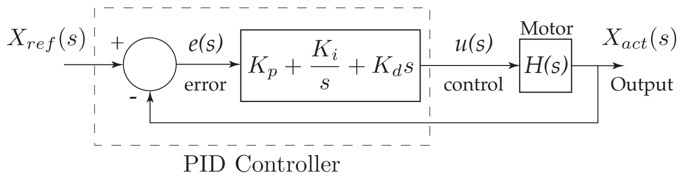


Figure L.2: block diagram of the system with a PID controller in the loop.

Exercise 07, and many of its components are reused.

Path planning

A common task for a positioning system is to start from a stationary position, move to a new location, and then stop. Of course, one way to do this is to apply an appropriate size step to the reference input of the position control system. However, depending on the system bandwidth, a sufficiently large step may require torques (current) and/or velocities (voltages) that exceed the motor/driver capabilities. In addition, the dynamic characteristics (e.g. rise time and overshoot) may be inconsistent with the application requirements. One remedy is to use a form of truncated ramp instead of the step reference input.
Suppose that we wish to reposition a mass-dominated load by X_{max} as rapidly as possible, subject to limitations on the maximum acceleration A_{max} and velocity V_{max}, while avoiding discontinuities in the position slope. One such command is constructed as shown in Figure L.3.
The motion has been divided into three sections: acceleration, constant velocity, and deceleration. Within this scheme, many variations are possible: High A_{max} would result in a long constant velocity section, with short accelerations. Alternately, for high V_{max}, the displacement would approach an s-shaped curve with no constant velocity section. Finally,

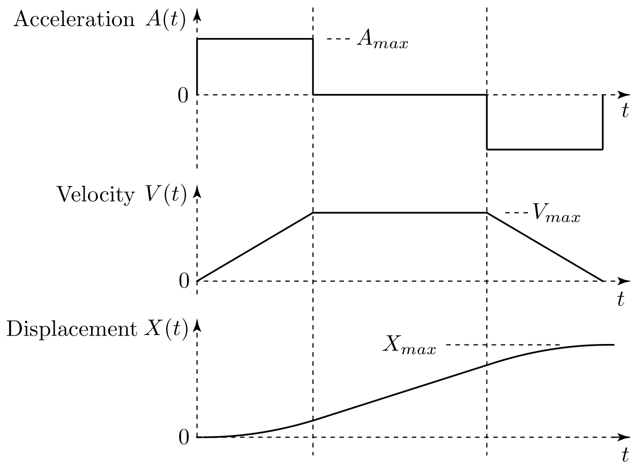


Figure L.3: a method of path planning for position control is to integrate piecewise-constant acceleration (top), to obtain piecewise-linear velocity (middle), also to be integrated to obtain piecewise-quadratic (and continuously differentiable) position (bottom).

by allowing both high A_{max} and V_{max}, the curve would approach a step.

In this lab exercise, you will use a C function `Sramps` that implements this time-varying displacement as the control system reference input. The function can link any number of ramp segments in succession, including specified dwell times at the end of each segment. It can also repeat the sequence of ramps indefinitely. See Lecture 08.L and Resource 16 for details.

PID control design and evaluation

For the lab exercise, you will write two Matlab scripts: one to design your PID controller and another to compare its performance to an analytical model. Specifically, the first script will design a PIDF controller using the MATLAB Control System toolbox function `pidtune`. This compensator should be designed to track the reference input, and to have control bandwidth of 8 Hz. A PID$\underline{\text{F}}$ controller improves noise immunity of a PID controller by limiting the high-frequency response of the derivative

term. Check your controller design by plotting the closed-loop step response using the plant parameters from Lab Exercise 07.
The script should convert the continuous-time transfer function to discrete-time (`c2d`, `tf`, and `tfdata`, with sample time $T = 0.0005$ s), and then use `tf2sos` (transfer functions to second order sections) to break the transfer function into biquads. Finally, use the `sos2header` function (see Resource 17) to write the biquad filter to a C header file (`PIDF.h`) in your Lab Exercise 08 project folder. That header can be `#include`d in your myRIO C program (after the definition of the `biquad struct`.) In this way, when you run your C program in Eclipse, it will automatically incorporate the latest version of your compensator design.
As in Lab Exercise 07, your second script will load the actual response of position control system (`Lab8.mat`), and compare it to both the ideal reference displacement and the dynamic model prediction. See below for details.

Program description

The program is similar in structure to that of Lab Exercise 07, consisting of (1) a Main thread that initializes the task and calls `ctable2` to communicate with the user, and (2) a Timer thread that maintains timing using an interrupt, implements the position control, and saves the results. Your specific controller definition is derived from the header file written from your Matlab script.

Two threads

Main thread The main thread performs the following tasks.

1. Initialize the table editor variables.
2. Initialize the path profile variables as follows.

```
typedef struct {
  double xfa; double v; double a; double d;
} seg;
```

3. Set up the timer IRQ interrupt (as in Lab Exercise 06 and Lab Exercise 07).
4. As in Lab Exercise 07, register and create the Timer thread to catch the timer interrupt. The Timer thread will gain access to both the table data and the path profile through pointers in the Thread resource. For example,

```
typedef struct {
  NiFpga_IrqContext irqContext;   // context
  table             *a_table;     // table
  seg               *profile;     // profile
  int               nseg;         // no. of segs
  NiFpga_Bool       irqThreadRdy; // ready flag
} ThreadResource;
```

5. Call the table editor. The table should contain three "show" values, labeled as follows.

   ```
   P_ref: revs
   P_act: revs
   VDAout: mV
   ```

6. When the table editor exits, signal the Timer thread to terminate. Wait for it to terminate.

Timer thread The Timer thread calls the interrupt service routine (ISR). At the beginning of the starting function, declare convenient names for the table entries from the table pointer, and for the ramp segment variables. For example,

```
double *pref  = &((threadResource->a_table+0)->value);
double *pact  = &((threadResource->a_table+1)->value);
double *VDAmV = &((threadResource->a_table+2)->value);
seg *mySegs = threadResource->profile;
int nseg    = threadResource->nseg;
```

The Timer thread includes a loop timed by the IRQ, and terminated only by its ready flag. Before the control loop begins:

- initialize the analog I/0, and set the motor voltage to zero, using `Aio_Write` (as is Lab Exercise 07) and
- set up the encoder counter interface (as in Lab Exercise 04).

Each time through the loop, it should:

1. Get ready for the next interrupt by: waiting for IRQ to assert, then writing the Timer Write Register, and writing `TRUE` to the Timer Set Time Register.
2. Call `Sramps` to compute the value of the current reference position P_{ref}. See below.
3. Call `pos`, to obtain the position of the motor P_{act}. See below.
4. Compute the current error $e = P_{\text{ref}} - P_{\text{act}}$.
5. Call `cascade` to compute the control value from the current error using PIDF control filter. Important: limit the computed control value to the range $[+7.5, -7.5]$ V.
6. Send the control value to the D/A converter `C00` using `Aio_Write`.
7. Change the table to reflect the current conditions of the controller.
8. Save the results of this BTI for later analysis. See below.

Functions

`cascade` – The `cascade` function, called once, from the ISR, during each BTI, implements the general-purpose linear difference equation algorithm from Lab Exercise 06. For this lab use the same C code that you used in Lab Exercise 06. In this case, the number of biquad sections will be 1.

Note that, as in Lab Exercise 06, all calculations should be made in (`double`) floating-point arithmetic.

`pos` – Write a `pos` function to read the encoder counter and return the displacement as a (`double`) in units of BDI (encoder counts), relative to the first position read.

`Sramps` – The C function `Sramps`, given in Resource 16, returns the current input reference position P_{ref}. The function accepts an input array of structures, each describing a separate displacement ramp segment. Called once each cycle of the control loop, `Sramps` steps through the segments, then repeats the complete path indefinitely.

We will initialize the path array in `main`, then pass the array and the number of segments to the Timer thread through the Thread Resource (described above in the Main thread section).

First, define the new segment data type `seg`:

```c
typedef struct {
  double xfa;  // position (revs)
  double v;    // velocity limit
  double a;    // acceleration limit
  double d;    // dwell time (s)
} seg;
```

Then, to test the position control system, initialize an array `mySegs` of type `seg` as follows:

```c
vmax = 50.;        // rev/s
amax = 20.;        // rev/s^2
dwell = 1.0;       // s
seg mySegs[8] = { // rev
  {10.125, vmax, amax, dwell},
  {20.250, vmax, amax, dwell},
  {30.375, vmax, amax, dwell},
  {40.500, vmax, amax, dwell},
  {30.625, vmax, amax, dwell},
  {20.750, vmax, amax, dwell},
  {10.875, vmax, amax, dwell},
  { 0.000, vmax, amax, dwell}
};
nseg = 8;
```

Notice that `mySegs` consists of four increasing ramps of 10.125 revolutions each, followed by four similar decreasing ramps that will return the motor to the starting position. All of the segments are subject to the same velocity and acceleration limits, and all dwell for one second before proceeding to the next segment.
You should declare the prototype of `Sramps` as:

```
int Sramps(
  seg *segs,    // segments array
  int nseg,     // number of segments
  int *iseg,    // current segment index
  int *itime,   // current time index
  double T,     // sample period
  double *xa    // next reference positon
);
```

At the end of the last segment, `Sramps` returns the total number of time steps in all of the segments. It returns `0` otherwise.
A typical call of `Sramps` might be:

```
nsamp = Sramps(mySegs, &iseg, nseg, &itime, T, &Pref);
```

When `Sramps` is called for the first time, set `*itime = -1`, and `*iseg = -1`, to initialize its operation.

Saving the responses

The data can be conveniently saved by defining data arrays in the ISR for each of the reference position, the actual position, and the torque. Then an auto-incremented index variable is used to store the data in the arrays during each BTI. Increment the index as needed, stopping when it reaches the length of the arrays. A convenient length would be 4000 points each. After the main loop terminates, but while still in the Timer thread, write the results, to the `Lab8.mat` file. The results should include:

1. your name (`string`),

2. the reference position array (rad), cast to `double *`,
3. the current position array (rad),
4. the torque array (N-m),
5. the PIDF array, cast to `double *`, and
6. the BTI length (s).

Use the same methods as Lab Exercises 04, 06 and 07 to bring the `Lab8.mat` file to Matlab.

Laboratory procedure

Test and debug your program.

Matlab analysis

In the second of your Matlab scripts:

1. Load the experimental results from the `Lab8.mat` file.
2. Define a discrete version of the motor/load plant transfer function from Lab Exercise 07. Consider using `c2d`.
3. Form the discrete controller from the values in the PIDF array in `Lab8.mat`.
4. Form the closed loop system models relating the reference position P_{ref} input to the position P_{act} and torque T outputs:

$$G_1(z) = \frac{P_{\text{act}}(z)}{P_{\text{ref}}(z)} \quad \text{and} \quad G_2(z) = \frac{T(z)}{P_{\text{ref}}(z)}. \tag{1}$$

5. Using `lsim`, simulate the system to find the theoretical responses for both the position $P_{\text{act}}(t)$ and the torque $T(t)$ to the reference position $P_{\text{ref}}(t)$ array that you stored in `Lab8.mat`.
6. In a single Matlab figure plot the results in three `subplot`s versus time, as follows:
   a) reference position, theoretical position, and experimental position;
   b) experimental error (reference − experimental position) and

theoretical error (reference − theoretical position); and

c) theoretical and experimental torque.

What do you conclude?

Resource R16 C function `Sramps` for position path planning

The following C function `Sramps` can be used to construct position commands that smoothly transition from one to another position over a period of time. It is used in Lab Exercise 08 to define position commands. A file containing this function, called `Sramps.c`, can be found at `ricopic.one/embedded_computing/source/Sramps.c`.

```c
/*
* Sramps.c
*
*  Created on: Mar 18, 2016
*      Author: garbini
*/
#include "math.h"

typedef struct {
  double xfa; double v; double a; double d;
} seg;

int Sramps(
  seg *segs,
  int nseg,
  int *iseg,
  int *itime,
  double T,
  double *xa
){
// Computes the next position, *xa,
// of a uniform sampled position profile.
// The profile is composed of an array
// of segments (type: seg)
// Each segment consists of:
//      xfa:    final position
//      v:      maximum velocity
//      a:      maximum acceleration
//      d:      dwell time at the final position
// Called from a loop, the profile proceeds from
// the current position,
// through each segment in turn, and then repeats.
// Inputs:
//  seg *segs:  - segments array
//  int nseg:   - number of segments in the profile
//  int *iseg:  - variable hold segment index
//  int *itime  - time index within a segment
//  (= -1 at segment beginning)
//  double T:   - time increment
// Outputs:
//  double *xa: - next position in profile
// Returns:
//  n - number of samples in the profile,
//  0 otherwise
//
// Call with *itime = -1, *iseg = -1, outside the loop
// to initialize.

double t, t1=0, t2=1, tf=1, tramp;
double x1=1, xramp, xfr=1, xr, d;
static double x0, dir;
static int ntot;
double vmax=1, amax=1;
int n;

if (*itime==-1) {
  (*iseg)++;
  if(*iseg==nseg) {
    *iseg=0;
    ntot = 0;
  }
  *itime=0;
  x0=*xa;
}
vmax=segs[*iseg].v;
amax=segs[*iseg].a;
d=segs[*iseg].d;
xfr=segs[*iseg].xfa-x0;
dir=1.0;
if(xfr<0){
  dir=-1.;
  xfr=-xfr;
}
t1 = vmax/amax;
x1 = 1./2.*amax*t1*t1;
if (x1<xfr/2) {
  xramp = xfr-2.*x1;
  tramp = xramp/vmax;
  t2 = t1+tramp;
  tf = t2+t1;
} else {
  x1 = xfr/2;
  t1 = sqrt(2*x1/amax);
  t2 = t1;
  tf = 2.*t1;
}
n = trunc((tf+d)/T)+1;

t = *itime*T;
if(t<t1) {
  xr = 1./2.*amax*t*t;
} else if (t>=t1 && t<t2) {
  xr = x1+vmax*(t-t1);
} else if (t>=t2 && t<tf) {
  xr = xfr-1./2.*amax*(tf-t)*(tf-t);
} else {
  xr = xfr;
}
```

```
*xa=x0+dir*xr;
(*itime)++;
if(*itime==n+1) {
  ntot = ntot + *itime - 1;
  *itime=-1;
  if(*iseg==nseg-1) {
    return ntot;
  }
}
return 0;
}
```

Resource R17 Matlab function `sos2header` for converting controllers to C

The following Matlab function `sos2header` can be used to convert a Matlab controller in second-order sections to a C floating point header file. It is used in Lab Exercise 08 to convert a controller designed in Matlab to a file a myRIO can read and implement in C. A file containing this function, called `sos2header.m`, can be found at `ricopic.one/embedded_computing/source/sos2header.m`.

```
function sos2header(fid, sos, name, T, comment)
  % Print to the filter definition for
  % FLOATING POINT header file.
  %
  % sos2header(fid, sos, name, T, comment)
  %
  %- fid      - File indentity
  %- sos      - Scaled second order sections, from "tf2sos"
  %- name     - Name to be given to the array
  %             of biquad structures, and
  %             associated with the number of sections.
  %- T        - Sample period in seconds
  %- comment  - comment added at top of header

%---structure form of cascade
fprintf(fid,'//---%s\n', comment);
fprintf(fid,'//---%s\n', datestr(now,0));
[ns,m]=size(sos);
fprintf(...
  fid,...
  'int %s_ns = %d; // number of sections\n',...
  name,...
  ns...
);
fprintf(...
  fid,...
  ['uint32_t timeoutValue = %d; ',...
  '// time interval - us; f_s = %g Hz\n'],...
  T*1e6,...
  1/T...
);
fprintf(
  fid,
  ['static\tstruct\tbiquad,'...
  '%s[]={ // define the array of floating point ',...
  'biquads\n'],...
  name...
);
for i=1:ns-1
  fprintf(fid,'        {');
  for j=[1,2,3,4,5,6]
    fprintf(fid,'%e, ',sos(i,j));
  end
  fprintf(fid,'0, 0, 0, 0, 0},\n');
end
fprintf(fid,'        {');
for j=[1,2,3,4,5,6]
  fprintf(fid,'%e, ',sos(ns,j));
end
fprintf(fid,'0, 0, 0, 0, 0}\n        };\n');
```

The Embedded Computing Laboratory

B Resources

Bibliography

Agarwal, A. and J. Lang (2005). Foundations of Analog and Digital Electronic Circuits. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science. isbn: 9780080506814.

Altera (2018). Is Tomorrow's Embedded-Systems Programming Language Still C? web.

ARM (june 2012). Cortex-A9 Revision: r4p1 Technical Reference Manual. ARM DDI 0388I (ID073015). ARM.

— (may 2014). ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. ARM DDI 0406C.c (ID051414). ARM.

Baldursson, Stefán (2005). ?BLDC Motor Modelling and Control – A Matlab®/Simulink®Implementation? mathesis. Chalmers University.

Barney, Blaise (july 2019). POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/.

Baz1521 (2018). Microprocessor—Wikipedia, The Free Encyclopedia. [Online; accessed 21-January-2018].

Booton, Richard C. and Simon Ramo (july 1984). ?The development of systems engineering? inIEEE Transactions on Aerospace and Electronic Systems: AES–20, pages 306–9.

Collins, Danielle (november 2018). When do you need a linear amplifier versus a PWM drive? web.

Franklin, G.F., J.D. Powell and M.L. Workman (1998). Digital Control of Dynamic Systems. Addison-Wesley world student series. Addison-Wesley. isbn: 9780201331530.

Gomez, Martin (december 2000). ?Embedded State Machine Implementation? inEmbedded Systems Programming: pages 40–50.

Horowitz, P and W Hill (2015). The Art of Electronics. Cambridge University Press. isbn: 9780521809269.

Instruments, National (august 2013). User Guide and Specifications NI myRIO-1900. 376047A-01. National Instruments.

Kernighan, B.W. and D. Ritchie (1988). C Programming Language. 2nd. Pearson Education. isbn: 9780133086218.

Lamberson, Jim (2018). Arithmetic logic unit—Wikipedia, The Free Encyclopedia. [Online; accessed 24-January-2018].

Nise, N.S. (2015). Control Systems Engineering, 7th Edition. Wiley. isbn: 9781118800829.

Patterson, David A. and John L. Hennessy (2013). Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. isbn: 9780124077263.

— (2016). Computer Organization and Design: The Hardware Software Interface ARM Edition. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. isbn: 9780128017333.

Rowell, Derek and David N. Wormley (1997). System Dynamics: An Introduction. Prentice Hall.

Sameli, Ioan (2018). Microcontroller—Wikipedia, The Free Encyclopedia. [Online; accessed 21-January-2018].

Xilinx (june 2017). Zynq-7000 All Programmable SoC Data Sheet: Overview. DS190 (v1.11). Xilinx.