

Robot Programming in “C”

Tak Auyeung, Ph.D.

February 15, 2006

- 20040201 2242 TA: Add a little more to the debouncing code, mostly just explanations of how it works.
- 20040125 1636 TA: I am giving up on installing AVR Studio 4.08 at the lab. Hacking an .MSI file turned out to be more trouble than I anticipated. The notes now include instructions to use the debugger/simulator chain included with WinAVR.
- 20040118 2333 TA: add design log section
- 20031103 2107 TA: creation, set outline according to curriculum

Contents

| | | |
|-----------|---|-----------|
| 0.1 | Copyright Notice | 10 |
| I | Background | 11 |
| 1 | Programming Robots | 13 |
| 1.1 | Robots | 13 |
| 1.2 | Robot Behavior | 13 |
| 1.3 | Our Scope | 13 |
| 2 | Tools for the Class | 15 |
| 2.1 | Free Software | 15 |
| 2.2 | Windows Platforms | 16 |
| 2.2.1 | Getting the Software | 16 |
| 2.2.2 | Setting up and Testing | 16 |
| II | Basic Input/Output and Peripherals | 19 |
| 3 | Binary Input/Output | 21 |
| 3.1 | Hardware Aspects | 21 |
| 3.2 | Configuration | 21 |
| 3.2.1 | Including header files | 22 |
| 3.2.2 | Naming convention | 22 |
| 3.2.3 | “Registers” for the ports | 22 |
| 3.2.4 | What is “pull up”? | 23 |
| 3.2.5 | Configuration code | 23 |
| 3.3 | Common Input Techniques | 24 |
| 3.3.1 | Edge Sensing | 25 |
| 3.3.2 | Debouncing | 27 |
| 4 | The Timer Device | 31 |
| 4.1 | Functions | 31 |
| 4.1.1 | As a Stopwatch | 31 |
| 4.1.2 | As a Source of Periodic Tick | 32 |

| | | |
|------------|---|-----------|
| 4.1.3 | Pulse Width Modulation (PWM) | 32 |
| 4.2 | ATMega128 Timer Configuration | 33 |
| 4.2.1 | Timer Configuration | 33 |
| 4.2.2 | PWM Configuration | 36 |
| 5 | Interrupts and Interrupt Service Routines | 37 |
| 5.1 | Concept | 37 |
| 5.1.1 | The global interrupt enable flag | 37 |
| 5.1.2 | The interrupt vector | 38 |
| 5.1.3 | Interrupt service routines | 38 |
| 5.1.4 | Writing an ISR | 39 |
| 5.1.5 | But what about INTERRUPT? | 39 |
| 5.2 | Timer ISR | 40 |
| III | Basic Open Loop Motor Control | 43 |
| 6 | Stepper Motors | 45 |
| 6.1 | Concepts | 45 |
| 6.1.1 | Full Stepping | 45 |
| 6.1.2 | Half Stepping | 46 |
| 6.1.3 | Software Control | 46 |
| 6.2 | Frequency Division | 47 |
| 6.3 | Displacement, Velocity and Acceleration | 49 |
| 6.3.1 | Displacement | 50 |
| 6.3.2 | Velocity | 50 |
| 6.3.3 | Acceleration | 50 |
| 6.3.4 | Getting back to Displacement | 50 |
| 6.3.5 | Physical Parameters | 51 |
| 6.3.6 | Velocity Profile | 51 |
| 6.3.7 | C code framework | 52 |
| 6.3.8 | What about differential steering? | 53 |
| 6.3.9 | Implementational Alternatives | 55 |
| 6.4 | Coding Tips | 55 |
| 6.4.1 | Efficiency issues | 55 |
| 7 | Project 1 | 57 |
| 7.1 | Final Objectives | 57 |
| 7.2 | Step by Step | 58 |
| 7.2.1 | Set up the Tools | 59 |
| 7.2.2 | Input/output | 59 |
| 7.2.3 | Frequency Division and Velocity | 59 |
| 7.2.4 | Acceleration and Deceleration | 59 |
| 7.2.5 | Displacement Control | 60 |
| 7.2.6 | Handle Negative Displacement and Velocity | 60 |
| 7.2.7 | Add Left and Right Motors | 60 |

| | |
|--|-----------|
| <i>CONTENTS</i> | 5 |
| 7.2.8 Add Angular Control | 60 |
| 7.3 Due Date and Grading | 60 |
| 7.4 How to Turn it In? | 61 |
| 8 R/C Servo Motors | 63 |
| 8.1 Concepts and Rationale | 63 |
| 8.1.1 Components of an R/C Servo | 63 |
| 8.1.2 Control Mechanism | 63 |
| 8.1.3 Why use R/C servos? | 64 |
| 8.2 Modification for Full Rotation | 64 |
| 8.3 PWM Control | 65 |
| 8.4 Issues and Tips | 66 |
| 8.4.1 Poor Speed Control | 66 |
| 8.4.2 Uneven Full Rotation | 66 |
| 8.4.3 Fragile Plastic Gears | 66 |
| 8.4.4 Not Meant for Continuous Use | 66 |
| IV DC Motor and Closed Loop Motion Control | 69 |
| 9 DC Motor Control | 71 |
| 9.1 DC Motor Direction Control | 71 |
| 9.2 DC Motor “Strength” Control | 71 |
| 9.2.1 Analog Electrical Potential Control | 71 |
| 9.2.2 PWM | 72 |
| 9.3 Are we done? | 72 |
| 10 Motion Encoding | 75 |
| 10.1 Optical Encoding | 75 |
| 10.2 Magnetic Encoding | 76 |
| 10.3 Quadrature Encoding | 76 |
| 10.4 Implementation | 77 |
| 10.4.1 General Approach | 77 |
| 10.4.2 External Interrupts | 77 |
| 10.4.3 Algorithm for Quadrature Encoding | 77 |
| 10.5 Electronic Interface | 79 |
| 10.5.1 Problems with a plain phototransistor | 79 |
| 10.5.2 Solution | 80 |
| 10.5.3 Example | 81 |
| 11 PID Loop | 83 |
| 11.1 The Theory | 83 |
| 11.1.1 The Proportional Term | 83 |
| 11.1.2 The Integration Term | 84 |
| 11.1.3 The Derivative Term | 84 |
| 11.2 From Continuous to Discrete | 84 |

| | | |
|-----------|---|------------|
| 11.3 | Practical Concerns | 85 |
| 11.3.1 | Resolution of Output | 85 |
| 11.3.2 | The Resolution of $e(t)$ and K_p | 85 |
| 11.3.3 | The Resolution of the Integration Term | 86 |
| 11.3.4 | The Resolution of the Differential Term | 86 |
| 11.4 | Improving Feedback Resolution | 87 |
| 11.4.1 | The Problem | 87 |
| 11.4.2 | A Solution | 87 |
| 11.4.3 | Practical Considerations | 87 |
| 12 | Motor Driver Circuit Design | 89 |
| 12.1 | TPIC0108 Interfacing | 89 |
| 12.1.1 | TPIC0108 Logic for DC Motors | 89 |
| 12.1.2 | TPIC0108 Logic for Stepper Motors | 90 |
| 12.1.3 | TPIC0108 Chopper Drive Logic | 91 |
| 12.2 | Layout Considerations | 91 |
| V | Reading Sensors | 93 |
| 13 | Analog to Digital Conversion | 95 |
| 13.1 | Configuration | 95 |
| 13.2 | The ADC clock | 95 |
| 13.2.1 | Voltage Reference | 95 |
| 13.2.2 | Multiplexer Selection | 96 |
| 13.2.3 | Free Running versus Manual | 96 |
| 13.2.4 | Interrupts | 96 |
| 13.2.5 | Enabling and Starting | 96 |
| 13.2.6 | Bit Positions | 97 |
| 13.2.7 | Reading Results | 97 |
| 13.3 | Common Techniques | 97 |
| 13.3.1 | One-shot ADC Read | 97 |
| 13.3.2 | Free-running Background Update | 97 |
| 13.3.3 | Server-client Approach | 98 |
| 14 | IR Ranging Sensor | 101 |
| 14.1 | General Information | 101 |
| 14.2 | Interface | 101 |
| 14.3 | Important Notes | 102 |
| 14.3.1 | Stablizing the Supply Voltage | 102 |
| 14.3.2 | Debouncing | 102 |
| 14.3.3 | Sampling Frequency | 103 |
| 14.3.4 | Inter-measurement Filtering | 103 |
| 14.3.5 | Distance Noise Prediction | 104 |

| | | |
|------------|---|------------|
| VI | Communication | 105 |
| 15 | General Communication | 107 |
| 15.1 | Reasons | 107 |
| 15.1.1 | Program update | 107 |
| 15.1.2 | Remote control | 107 |
| 15.1.3 | Debugging and Logging | 107 |
| 15.1.4 | Environmental Interaction | 108 |
| 15.1.5 | Inter-robot Interaction | 108 |
| 15.2 | Devices | 109 |
| 15.2.1 | RS-232, asynchronous | 109 |
| 15.2.2 | RS-485 | 109 |
| 15.2.3 | Ethernet | 110 |
| 15.2.4 | Wifi | 110 |
| 15.2.5 | Bluetooth | 111 |
| 16 | Low-Level Asynchronous Communication | 113 |
| 16.1 | Timing Details | 113 |
| 16.1.1 | Start Bit(s) | 113 |
| 16.1.2 | Data Bits | 114 |
| 16.1.3 | Parity Bit | 114 |
| 16.1.4 | Stop Bit | 114 |
| 16.2 | The USART | 114 |
| 16.2.1 | Clock and Transmission Speed | 114 |
| 16.2.2 | Frame Format | 115 |
| 16.2.3 | The Ninth Bit and Networking | 115 |
| 16.2.4 | Errors | 116 |
| 16.2.5 | Interrupts | 116 |
| 16.2.6 | Enabling and Disabling | 117 |
| 16.3 | Circular Queues as Software Buffers | 117 |
| 16.3.1 | An Example: Circular Queue | 117 |
| 16.3.2 | Where is the Array? | 120 |
| 16.3.3 | The Record Structure | 120 |
| 16.3.4 | Logic for a Circular Queue | 121 |
| 16.4 | Circular Queue Implementation | 123 |
| 16.4.1 | <code>cq.h</code> | 123 |
| 16.4.2 | UART ISR Logic | 123 |
| 16.4.3 | UART API | 124 |
| VII | Robot Behavior | 125 |
| 17 | Threading | 127 |
| 17.1 | Life without Multithreading | 127 |

| | |
|--|------------|
| VIII Project | 129 |
| 18 Project | 131 |
| 18.1 Robot Objectives | 131 |
| 18.2 Robot Specifications | 131 |
| 18.3 Design | 132 |
| 18.3.1 Drive Platform | 133 |
| 18.3.2 Fire Extinguisher Subsystem | 133 |
| 18.3.3 Sensors | 133 |
| 18.3.4 Communication Subsystem | 133 |
| 18.3.5 Battery and Power Subsystem | 133 |
| 18.3.6 Control Units | 133 |
| IX Robot Design Log | 135 |
| 19 Robot Design Log | 137 |
| 19.1 Design Requirements | 137 |
| 19.1.1 Physical | 137 |
| 19.1.2 I/O | 137 |
| 19.1.3 Software (added 20040307) | 138 |
| 19.2 Components | 138 |
| 19.2.1 Pin count | 138 |
| 19.2.2 Component Selection | 139 |
| 19.2.3 Mini-Sumo Chassis Design | 140 |
| 19.3 Design (added 20040307) | 140 |
| 19.3.1 Bootloader (added 20040307) | 140 |
| 19.3.2 H-Bridges | 140 |
| 19.3.3 (20040322) RS232 | 141 |
| 19.4 PCB Design Log | 141 |
| 19.4.1 20040410 | 141 |
| 19.4.2 20040406 | 143 |
| 19.4.3 20040404 | 143 |
| 19.4.4 20040331 | 144 |
| 19.4.5 20040322 | 144 |
| 19.5 Drive System | 145 |
| 20 A.N.T. | 147 |
| 20.1 GPS | 147 |
| 20.1.1 Interface | 147 |
| 20.2 Stereo Vision | 147 |
| 20.2.1 Theory | 147 |
| 20.2.2 Equipment | 149 |

| | |
|-------------------------|------------|
| 21 Project Log | 151 |
| 21.1 20050129 | 151 |
| 21.2 20040326 | 152 |
| 21.3 20040319 | 153 |
| 21.4 20040318 | 154 |

X Work In Progress **155**

| | |
|--|------------|
| 22 DC Motor Encoding | 157 |
| 22.1 The EMF | 157 |
| 22.2 The Problem | 157 |
| 22.3 The Solution | 158 |
| 22.4 Detecting Circuit Open | 158 |
| 22.5 Detecting Circuit Closure | 158 |
| 22.6 | 158 |

Copyright Notice

All materials in this document are copyrighted. The author reserves all rights. Infringements will be prosecuted at the maximum extent allowed by law.

You are permitted to do the following:

1. add a link to the source of this document at www.drta.org
2. view the materials online at www.drta.org
3. make copies (electronic or paper) for *personal* use only, given that:
 - (a) copies are not distributed by *any* means, you can always refer someone else to the source
 - (b) copyright notice and author information be preserved, you cannot cut and paste portions of this document without also copying the copyright notice

Part I

Background

Chapter 1

Programming Robots

1.1 Robots

There are many different types of robots. For example, there are remotely controlled robots in Battlebots, mostly autonomous robots for Mars exploration, autonomous robots for Micromouse competitions and etc. There are also “vehicles” that are autonomous, hence sometimes considered special robots. For example, a cruise missile can be considered a robot.

1.2 Robot Behavior

Except in remotely controlled robots, the behavior of a robot is usually governed by one or more programs, running on one or more computers on board the robot. Robot behavior programming is often considered an area in artificial intelligence, and it is a form of high level programming. Programs that determine the behavior of a robot are often not written in “conventional” languages (such as C, Basic, Pascal and etc.).

1.3 Our Scope

This class approaches robot programming from the lowest level. In other words, we begin with very basic control of individual components on a robot, then gradually introduce concepts that combine components to more meaningful robot functions.

Chapter 2

Tools for the Class

2.1 Free Software

This class utilizes mostly, if not all, free software. There is a wealth of high quality free software tools out there, but most people do not know such tools because there is little advertisement about such tools.

The primary compiler we will use is `gcc` (GNU C Compiler). `gcc` is a mature C compiler that is multihomed and multitargetted. This means the compiler itself can run on many platforms (such as Linux and Windows). At the same time, the code compiled by `gcc` can also run on many platforms. In fact, the platform on which the compiler itself runs does not need to be the same platform that the compiled code is targetted for! A compiler that has a target platform different from its home platform is called a cross compiler.

Besides the compiler, there are other necessary tools we need to get:

- `binutils`: this is a collection of programs that supports a compiler. Such programs include a linker and an assembler.
- `libc`: the job of a compiler is to translate. This means a programmer still needs to specify how to do *everything*. `libc` is a library of useful subroutines that are “commonly used” in programs. In other words, `libc` is a collection of subroutines that most people find useful in programming. `libc` also include the necessary header files to use such pre-defined subroutines.
- `gdb`: this is the debugger. It is not too helpful unless we use ICE (in-circuit emulator) devices for debugging.
- `uisp`: this is a device programmer software tool that allows a PC write a program to a target computer.

2.2 Windows Platforms

The tool chain is *nix-oriented, but it is ported to the Windows platform. Ideally, the tools should be used on a Linux-based platform, but the Windows platform is sufficient for all practical purposes.

2.2.1 Getting the Software

The main website to get the Windows port of the tool chain is <http://www.avrfreaks.net>. Click on the “AVR GCC” tab, then click on “Download it HERE” after the content list. This brings you to another site. Scroll down until you find the “Latest File Releases”, then click on “Download” on the row of WinAVR. This brings to another page. Scroll down, and download the file ending with “bin-install.exe”. You may want to save the executable in case you need to reinstall WinAVR. Then, you can select a host that is close to you and click on the icon with lots of 0s and 1s.

Now, click “Back” on the browser and go all the way back to the “AVR GCC” page at www.avrfreaks.net. Be sure to download and save the document written by Colin O’Flynn about installing and configuring WinAVR.

Go to <http://www.avrfreaks.net/Tools/showtools.php?ToolID=258> and download AVR Studio 4.08. This version works well with the WinAVR tool set.

2.2.2 Setting up and Testing

First, install the simulator. This should be a fairly painless process (unless you don’t have admin. access rights). The simulator is not required, but it is convenient. Then, install WinAVR, which should also be painless.

Now a little bit of elbow grease. Start a command line interface (otherwise known as a DOS prompt) first. Go to the directory where WinAVR was installed. Let’s say you keep the defaults and install WinAVR at `c:\winavr`, then issue the following commands:

```
c:
cd \winavr\examples
cd demo
copy \winavr\sample\makefile .
edit makefile
```

In the editor, look for the line that reads

```
MCU = atmega128
```

Change `atmega128` to `at90s2313`.

Then, look for the line that reads

```
TARGET = main
```

Change `main` to `demo`.

Save the file and exit the editor. Generally, you only need the following command to make all the targets:

```
make all
```

AVR Studio 4.08

Back in the command line, issue the following command:

```
make extcoff
```

No error should occur, and a file name `demo.cof` should be created (along with many other files).

Now, start up AVR Studio 4.08 (in “Atmel AVR Tools” from the Start menu). Click “Open” from the opening screen, then look for the file named `demo_cof.aps` (the extension `.aps`) may not show. You can now run the program step-by-step using the simulator.

Simulavr + Insight

Unfortunately, AVR Studio 4.08 is not installed at the lab. The older version of AVR Studio has a bug that makes it incompatible with the WinAVR tool set. This is not a mistake on WinAVR’s part (or any of its contributors), but on Atmel’s part. WinAVR has the necessary tools to create files conforming to Atmel’s description of the COFF format, but AVR Studio 3.5x itself does not conform to Atmel’s own standard.

Another reason why software source should be open.

Anyway, we can still debug programs without Atmel’s simulator. A GPL’d simulator, called `simulavr`, has been written to provide the backend to simulate the execution of AVR code on practically any platform. This simulator runs with `gdb` (GNU Debugger) to provide debugging ability.

Let’s get familiarized with the necessary components to get this to work:

- `WinAVR`: this is the inclusive package that you should install on a PC running Windows. If you use Linux, you need to download various packages.
- `avr-gcc`: this is the GNU C Compiler cross targetted to the AVR.
- `avr-libc`: this is a collection of libraries for use with `avr-gcc`.
- `avr-gdb`: this is a debugger cross targetted to the AVR.
- `avr-binutils`: this is a collection of useful utility programs to create executables.
- `simulavr`: this is the simulator. It can run as a “server” so that a debugger connects to it as it is a remote target machine. The connection, however, is via a socket rather than a serial port.

- insight: this is a GUI-based frontend of gdb. You *don't* need this to debug a program. Most people find the text interface of gdb a little too primitive and commands difficult to learn. This is included in WinAVR.

Now, let's go through the step-by-step procedures to debug a program using this tool chain:

1. Make sure the program is compiled and linked, see subsection 2.2.2.
2. Click the Start button, then “run...”, then type in the following:

```
simulavr --gdbserver --device at90s2313
```

Substitute `at90s2313` with whatever target MCU your program is targeted. Selecting the correct target is very important.

Note that once `simulavr` runs, it does not exit. It gives you a message like “Waiting on port 1212 for gdb client to connect...” if everything is okay. Leave the command line interface alone (minimize it if you don't want to clutter up your screen).

3. If you install WinAVR, you should see an icon titled “AVR Insight (WinAVR)” on the desktop. Run the program.

If you do not see the desktop icon, go to the folder containing the executable and launch it manually. The full path is usually `c:\WinAVR\bin\avr-insight.exe`.

4. In Insight, use “File — Target Settings”, and select “GDBServer/TCP” as the Target, the “Host” should be “localhost”, and the “Port” should be set to 1212. For convenience, make sure “Set breakpoint at ‘main’ ” is checked.

Click on “More Options”, and make sure *both* “Attach to Target” and “Download Program” are both checked.

You only need to do this once. Insight saves the options automatically.

5. In Insight, use “File — Open” and locate `demo.elf`. Click “Open”.
6. In Insight, use “Run — Download” to upload the executable. This step should not have been necessary, but it is!
7. Click the run icon (a running person), the program should start, and the debugger stops on the first statement.
8. If you see a line highlighted in green, you have successfully launched the debugger!

Part II

Basic Input/Output and Peripherals

Chapter 3

Binary Input/Output

This chapter deals with the most basic I/O ability of any controller board intended for embedded system control. Robots are essentially very specialized machines with embedded control systems. As a result, almost every robot controller has these binary I/O ports.

3.1 Hardware Aspects

On a physical controller, many pins (or electrical contacts) can be configured to serve as a binary input or a binary output. Even a US\$12 microcontroller IC (Atmel's ATmega128) has more than 40 pins of binary I/O.

A binary input pin is a pin where the electronics can “sample” the voltage. A sampled voltage of high (5V or 3.3V) is registered as a 1 in software, whereas a sampled voltage of low (0V) is registered as a 0 in software.

A binary output pin is a pin where the electronics can “drive” the voltage. When software writes a 1 to a binary output, most controllers will attempt to drive the output to a high voltage (5V or 3.3V). When software writes a 0 to a binary output, most controllers will attempt to drive the output to a low voltage (0V).

Note that the binary I/O of a controller is, more often than not, unsuitable for sensing common voltages or controlling anything directly. Interface circuits are often needed to protect the binary I/O pins on controllers. Since this class is not an electronic class, such interface circuits are not discussed.

3.2 Configuration

Most MCUs allow software configure whether a physical pin be used as input or output. On the ATmega128 (and other members of the AVR family), each binary I/O pin has two bits for configuration:

- direction: whether the pin is input or output

- state: for an output pin, whether it should drive high or low

In addition, there is also a “sense” or “input” bit for each binary I/O pin that reports either a 1 (for a high voltage) or a 0 (for a low voltage) at the pin.

3.2.1 Including header files

When a C program includes the proper header, and the `Makefile` is properly set up, a C program can use handy symbolic name to control the I/O features of an MCU.

Generally speaking, you want to place the following lines at the beginning of a C program:

```
#include <avr/io.h>
#include <avr/ina90.h>
```

Also, specify the MCU in a `Makefile`:

```
MCU = atmega128
```

This combination allows `gcc` automatically include the correct header file for the chosen MCU (so that you don’t have to make your program specify to a particular MCU).

3.2.2 Naming convention

Although software can configure each individual pin, the AVR MCUs provides interfaces to configure 8 pins at the same time. A “port” is a group of 8 pins, and software can configure all 8 pins in a port in one operation.

It is conventional with AVR MCUs to refer to the ports as port A, port B and etc. The ATMega128 has port A to port G available, while some of the smaller MCUs only have one or two ports.

3.2.3 “Registers” for the ports

For port A, there are three “variables” defined. Bit n of each of these variables corresponds to a pin n of port A.

- `DDRA` (Data Direction Register A): this “register” (not to be confused with registers in a processor) controls the directions of the 8 pins of port A. A bit value of 0 means the corresponding pin is configured as input, whereas a bit value of 1 means the corresponding pin is configured as output.

`DDRA` is a read/write variable.

- `PINA` (Port INput A): this is a read-only variable (writing to it doesn’t do anything) that reports the sensed state of the pin. A bit value of 0 means the pin is sensing a low voltage, whereas a bit value of 1 means the pin is sensing a high voltage.

- PORTA (data register of PORT A): this register is a read-write register, and its bits has different meanings depending on whether the corresponding pin is configured for input or output.

For an output pin, a bit value of 0 means the pin drives low (tries to lower the voltage), whereas a bit value of 1 means the pin drives high (tries to raise the voltage).

For an input pin, a bit value of 0 means the pin has no “pull-up”. A bit value of 1 means the pin is “pulled up” by a large resistor. We’ll discuss the concept of “pull up” later.

3.2.4 What is “pull up”?

Consider “pull up” as “a weak tendency to default to a high voltage for an input pin”. This means that if a pin, configured as input and with “pull up”, is not connected to anything, it senses a high voltage.

The “weak tendency” is important, because we want the sensed voltage of this input pin change as soon as some component (such as a sensor) drives the voltage low. Most sensors that are transistor based or labeled “open collector” only have the ability to drive low but not drive high, which means it is important for the input pin to have the ability to “default” to a high voltage.

To make this more clear, there are only two situations with an input pin that is configured to “pull up”:

- The external component is not driving low. The input pin defaults to a high voltage and reports a 1.
- The external component drives low, overrides the default voltage, the input pin senses a low voltage and reports 0.

If there is a “pull up” configuration, is there a “pull down” configuration? The answer is yes. However, “pull down” is less commonly used, so MCUs typically do not have built-in abilities to configure for “pull down”. One can still configure “pull down” with circuits external to the MCU.

3.2.5 Configuration code

The easiest way to configure pins of a port is simply to use assignment statements. For example, the following assignment statement configures pins 0, 2, 3 and 7 for input, and all other pins (1, 4, 5 and 6) for output:

```
DDRA = 0x72;
```

The value 0x72 is the value $2^1 + 2^4 + 2^5 + 2^6$, represented by a hexadecimal number.

While simple assignment statements are useful for the initialization of port pins, they are not suitable when only one bit needs to be changed. For example, if we only need to change the state of pin 4 of port A so it drives high (assuming it is already configured for output), we should not use the following statement:

```
PORTA = 0x10;
```

This is because this statement also affects pins 0, 1, 2, 3, 5, 6 and 7 of port A! Fortunately, `PORTA` is a read/write variable. We can read it back first, then use bitwise operators to change only the bits that we want to change. In this example, we should use the following statement to make pin 4 drive high:

```
PORTA = PORTA | 0x10;
```

This works because when a bit is ored with 0, the original value is preserved. To change pin 2 to drive low (assuming it is already configured for output), we can use the following code:

```
PORTA = PORTA & ~0x04;
```

The `~` operator means bitwise not. It is also called one's complement. Essentially, `~0x04` is the same as `0xfb`, but the former is easier to read because it is more clear which bit is a 1. Note that any bit anded with 1 preserves its original value, and that's why this operation does not the configuration of pins other than pin 2.

For the C savvy programmers, you can also replace the previous simple assignment operators with the operator-assignment operators:

```
PORTA |= 0x10;
PORTA &= ~0x04;
```

If you'd rather let the compiler figure out the bit pattern (from the pin number), you can use the left-shift operator `<<` as follows:

```
PORTA |= 1 << 4;
PORTA &= ~(1 << 2);
```

3.3 Common Input Techniques

This section discusses some common techniques to “filter” raw input from the pins of an MCU. Both techniques can use push buttons as examples. In this context, we assume a push button is “normally open”. This means when the push button is not being pushed, the two electrical contacts are not connected. When the push button is pushed, the two electrical contacts are connected.

A typical use of such NO (normally-open) push buttons is to connect one end to ground, and another end to a binary input pin that has a pull-up. This way, there are two states:

- When the button is released, there is no connectivity to ground. As a result, the binary input pin goes to its default state. Due to the pull-up configuration, the default state is high. In software, this pin reads 1.

- When the button is pushed, it connects the binary input pin to ground (0V). This connection results in a very strong “force” to change the voltage at the binary input pin to 0V. As a result, the default pull-up is overcome, and the binary input pin reads 0.

The bottom line is: 0 means pressed, 1 means released.

3.3.1 Edge Sensing

Push buttons are often used as crude user interface for embedded systems. Most push buttons are fairly inexpensive, small, and easy to solder onto a PCB (printed circuit board).

As a result, it is a common task that an embedded program needs to interpret buttons. Let us assume pin 5 of port A is configured as an input pin with internal pull up. You should already know by now that the configuration code for this pin should be as follows (assuming all other pins are already configured):

```
DDRA &= ~(1 << 5);
PORTA |= 1 << 5;
```

To call `button_down` when the button is pressed, and call `button_up` when the button is released, we can use the following code:

```
if (PINA & (1 << 5)) button_up();
else button_down();
```

This works if you only want to know the *current* state of a button. However, often we need to know a *change* of state, rather than the current state. In other words, a program may not be interested at all in whether a button is pressed or released, but when the button is *being* pushed or released.

There are two techniques to detect the “edge” of a binary signal change. The first method is called busy polling, whereas the second does not have an actual technical name.

Busy Polling

This technique keeps monitoring the state of a button, and a change of state causes the code exit a loop. For example, the following code waits for a button to be pressed, and then released, before it calls `button_event`:

```
while (PINA & (1 << 5));
// button is now pressed
while (!(PINA & (1 << 5)));
// button is now released
button_event();
```

The two loops are busy loops that do not do anything inside. The first loop exits when the button is pressed, whereas the second loop exits when the button is released. You can insert another function call between the loops if you need to respond to the “button is being pushed” event.

Busy polling is relatively easy to code, but it suffers from one major problem: it is wasteful of processing resources. In addition, it is difficult to extend the code, due to its structure, to detect and respond to edge events of multiple buttons.

Stateful Event Detection

Okay, I am inventing this name here.

In this second approach, we write our code in a much more modular fashion. For each binary input, we write a function similar to the following:

```
void event_a5(unsigned char oldstate, unsigned char newstate)
{
    if ((oldstate ^ newstate) & (1 << 5))
    {
        if (oldstate & (1 << 5))
        {
            // respond to falling edge
        }
        else
        {
            // respond to rising edge
        }
    }
}
```

This function, by itself, does not even read any inputs! So, we need some extra code to wrap around it:

```
newstate = PINA;
event_a5(oldstate, newstate);
oldstate = newstate;
```

The code shown above updates the variables `newstate` and `oldstate`. `newstate` can be an auto (stack allocated) local variable in a function, while `oldstate` should be either a global variable, or a static local variable. This is because we can potentially invoke the function containing this code repeatedly by yet another caller.

But wait a minute here, what have we gained with this complicated scheme?

The main advantage of this approach, compared to the previous polling technique, is that this code is easily extended. If you want to handle events for another button connected to pin 6 of port A, write a subroutine `event_a6`, and insert a call so you have

```
newstate = PINA;
event_a5(oldstate, newstate);
event_a6(oldstate, newstate);
oldstate = newstate;
```

The kind of control structure allows the handling of events for each pin to be separated from each other, resulting in cleaner code that is easier to maintain.

To be complete, the entire subroutine (to call `event_a5`) should look like the following:

```
void one_tick(void)
{
    static unsigned char oldstate = 0xff;
    unsigned char newstate;

    // ... whatever code you want to put here
    newstate = PINA;
    event_a5(oldstate, newstate);
    event_a6(oldstate, newstate);
    oldstate = newstate;

    // whatever code you want to put here
}
```

Shouldn't this function `one_tick` be called repeatedly? Yes, it must be called periodically for the logic to work. You can write the following code:

```
while (1)
{
    one_click();
}
```

Besides the fact that you can detect and handle events for multiple input pins, this approach may not seem to be that different from busy polling. However, once we discuss timers and timer interrupts, you will appreciate the strengths of this approach.

3.3.2 Debouncing

In the perfect world, when a switch changes state, it is a single event. Unfortunately, switches, such as push button switches, are mechanical devices. The contact bounces a few times before it settles. In other words, a program, polling at a relatively high frequency, sees a whole bunch of transitions for a single button push. The same is true when a button is released.

This is not good. When a button is pushed once, a program may register it as several push-and-release events. If you rely on counting events to change settings, you will have a hard time getting to the correct settings!

Instead of fixing the problem in the switches, the problem can be solved in software. Debouncing is a software technique used to “filter out” bounces to yield clean state change events.

The Concept

The concept of debouncing is quite simple. Unless we can observe the same state n times, we keep the previous state. That is, we need n consistent reads to confirm the state.

The Implementation

The implementation of debouncing varies from very clumsy hard coding, to flexible schemes that is easy to extend.

At the core of a flexible scheme, we need to maintain a circular queue to track the previous n readings. Let us assume the `const int db_n` represents this n . We can write a quick-and-dirty subroutine to maintain the circular queue:

```
void db_tick(void)
{
    const int db_n = 5; // or whatever
    static unsigned char db_buffer[db_n] = {
        0xff,0xff,0xff,0xff,0xff };
    static int db_cursor = 0;

    db_buf[db_cursor] = PINA;
    if (++db_cursor >= db_n) db_cursor -= db_n;
}
```

I called this quick-and-dirty because a circular queue should have been implemented as a `struct` with associated functions to initialize and maintain its state.

Every time we call `db_tick`, it reads from the input pins, and update the circular buffer. However, it does not track the debounced states of each pin. We can add to the function to do this:

```
void db_tick(void)
{
    const int db_n = 5; // or whatever
    static unsigned char db_buffer[db_n] = {
        0xff,0xff,0xff,0xff,0xff };
    static int db_cursor = 0;
    static unsigned char db_state = 0xff; // default state
    unsigned char db_and, db_or;
    int i;

    db_buf[db_cursor] = PINA;
```

```

if (++db_cursor >= db_n) db_cursor -= db_n;
db_or = 0x00;
db_and = 0xff;
for (i = 0; i < db_n; ++i)
{
    db_or |= db_buffer[i];
    db_and &= db_buffer[i];
}

db_state |= (db_and);
db_state &= (db_or);
}

```

Explanations

The key to the previous code is the loop. Let take a look at the initialization, the loop itself, and its results.

```

db_or = 0x00;
db_and = 0xff;

```

This initializes the two variables we track in the loop. `db_or` is a cumulative bitwise-or “sum” (disjunction is somewhat analogous to addition in boolean algebra). The reason why each bit is initialized to 0 is simple: if any bit is initialized to 1, it will always be a 1, making the loop useless!

Likewise, `db_and` is a cumulative bitwise-and “product”. All bits in this variable are initialized to 1 because if any were initialized to 0, it will always remain 0.

```

for (i = 0; i < db_n; ++i)
{
    db_or |= db_buffer[i];
    db_and &= db_buffer[i];
}

```

In the loop, both `db_or` and `db_and` are getting updated by the elements of `db_buffer`. Recall that `db_buffer` is a history of previous states of a port.

Let us think for a second here, not that I think you are starting to zone out. After the loop exits, how do we interpret each of the following?

- a bit value of 0 in `db_or`
- a bit value of 1 in `db_or`
- a bit value of 0 in `db_and`
- a bit value of 1 in `db_and`

Here is the answer.

- a bit value of 0 in `db_or`: the corresponding pin of the port has a complete history of 0.
- a bit value of 1 in `db_or`: the corresponding pin of the port has at least one sample of 1 in the history.
- a bit value of 0 in `db_and`: the corresponding pin of the port has at least one sample of 0 in the history.
- a bit value of 1 in `db_and`: the corresponding pin of the port has a complete history of 1.

Isn't this fantastic? Afterall, we *were* looking for consistent samples for the entire history buffer!

```
db_state |= (db_and);  
db_state &= (db_or);
```

Marks the end of this logic. `db_state` is the “debounced state”. Each bit in this variable represents a filtered state of the corresponding pin of the port. If a pin has consistently read 1 for the whole history, it is safe to update `db_state` and make the corresponding bit a 1. This is done by a bitwise-or with `db_and`.

Similarly, if a pin has consistently read 0 for the whole history, it is safe to update `db_state` and make the corresponding bit a 0. This is done by a bitwise-and with `db_or`.

Q.E.D., quod erat demonstrandum (which was to be demonstrated, or in plain English, end of proof).

Chapter 4

The Timer Device

In the previous chapter, we discussed basic binary input and output pins. We mentioned how busy polling is wasteful of processor resources. For most applications, continuous sampling of a pin is not required. Sampling a pin “once in a while” is often appropriate. “Once in a while” here translates to tens of milliseconds to even hundreds of milliseconds.

In this chapter, we discuss how this can be done. Sampling at a certain frequency is one of the many possible applications of a timer device. Modern MCUs often include at least one timer devices on the same chip with the processor and other components. Each timer operates in parallel to the processor itself, and most of the time timers are independent of each other in an MCU with multiple timers.

4.1 Functions

At the core of a timer is a counter. Depending on the design, this counter can either count up or down. The frequency of counting (increment or decrement) is often a factor of the main processor frequency. However, a “timer” often has a alternate operation mode as a counter, in which an *external* clock is used to trigger the increment or decrement.

4.1.1 As a Stopwatch

The counter in a timer is software readable. In other words, a program can check on the counter from time to time. As a result, a timer can be used as a stop watch by software:

- processor resets the counter of a timer
- processor wait for signal to time
- signal received, start the mentioned timer to count (using internal clock)

- processor wait for signal to stop the stopwatch
- signal received, stop timer and read the counter

In this case, the counter value is a measurement of the duration between the start signal and the stop signal.

4.1.2 As a Source of Periodic Tick

In an application that needs to keep track of time, or has operations to perform periodically, a timer can be used to provide periodic ticks.

In this mode, a timer is set up to count continuously. The counting frequency is often a factor of the main clock frequency. For count up timers, software usually sets up an overflow value. For count down timers, software can usually set up a restart value. Sometimes, the overflow or restart value cannot be specified in software, and hardcoded to some powers of 2.

For a count up timer, when the counter value becomes greater than or equal to the overflow value, the timer resets the counter to zero *and* causes a hardware interrupt. We'll talk about interrupts later. For now, let's just say that "the timer calls a particular subroutine in software".

For a count down timer, when the counter value gets to zero, the timer reloads the counter with the reload value, and it causes a hardware interrupt.

As you can see, whether a timer counts up or down is not important. The important point is that it "calls a particular subroutine in software" periodically. We'll discuss interrupt service routines (ISRs) later.

4.1.3 Pulse Width Modulation (PWM)

On some AVR MCUs, timers can also be used to generate PWM signals. A PWM signal has a fixed period. However, the on-duration (pulse width) of one period can range from 0 to the entire period. PWM is a very useful concept because it can be applied to normally on/off switches to achieve "analog" or "graduated" results.

In the context of robot programming, PWM is often used for the following purposes:

- directly control the duty cycle of DC motors
- generate the control signal to control RC servos
- implement software-based chopper drive for stepper motors

In order for a timer to generate PWM, there is one additional value called "output compare". This value is less than or equal to the overflow value. When the counter of a timer is less than this output compare value, an output pin drives high, when the counter of a timer is greater than or equal to this output compare value, the same output pin drives low. As a result, the duty cycle is

directly proportional to the ratio of the output compare value and the overflow value.

Note that the overflow value controls the period of a PWM signal.

The PWM feature of a timer is very useful. It is *possible* to generate PWM in software. However, software generated PWM is processor power intensive, and it does not have the same precision of timer generated PWM signals.

4.2 ATmega128 Timer Configuration

This section is specific to timers in the ATmega128. However, many of the concepts are also applicable to other members in the AVR family. Of all the timers in the ATmega128, this section concentrates on timer1, although timer3 is almost identical in terms of features of configuration.

Because timer1 is such a flexible device, it is impossible to enumerate all possible configurations. This section focuses on setting up timer1 so that it does the following:

- overflows every 20ms
- outputs PWM signals
- interrupts whenever it overflows

4.2.1 Timer Configuration

Given a clock frequency of 16MHz and a desired period of 20ms, we can compute that we want the timer to overflow every $16000000\text{Hz} \times 20\text{ms} = 320000$ main clock cycles. Since the overflow value is a 16-bit number, we know that we need a divider to reduce the counting frequency.

$320000 \div 65536$ is more than 4 but less than 5. As a result, we know a prescaler of more than 4 will work. According to page 134 of the ATmega128 data sheet, the closest prescaler is a divide-by-8 prescaler. This means, from the table, that the clock select bit patterns to be 010_2 . These three bits are bit 2 to bit 0 in Timer/Counter1 Control Register B (TCCR1B).

We have so far determined that TCCR1B has a bit pattern of $????010_2$. Because we are not using the device as a counter, bits related to “input capture” are useless, so we can further determine that the bit pattern is $00???010_2$. Bit 5 is reserved and must be written as a 0, so our pattern is now $000???010_2$.

Based on the divide-by-8 prescaler, the counter counts at a frequency of 2MHz. At this frequency, we need an overflow value of $2000000 \times 20\text{ms} = 40000$. We can use the single slope “fast PWM” mode because we do not change the period. We can choose to use either OCR1A (Output Compare Register 1 for Channel A) or ICR1 (Input Capture Register 1) for this overflow value. Since we want to reserve as many PWM output channels as possible, we use ICR1.

Each I/O location is only 8-bit, but ICR1 is a 16-bit quantity. As a result, ICR1 is split into two 8-bit I/O locations: ICR1L and ICR1H. AVRs use an

internal buffer to synchronize the update of a 16-bit number. It requires that the high byte be written first, then the write of the low byte updates all 16 bits at once. The following code does just this:

```
ICR1H = 40000U >> 8;
ICR1L = 40000U & 0xff;
```

As per table 61 on page 132 of the data sheet, to use ICR1 as the overflow value and use fast PWM, we need to use mode 14. The WGM (Wave Generation Mode) bit pattern should be 1110_2 . According to page 129, bit 1 and bit 0 of WGM are bits 1 and 0 of Timer/counter1 Control Register A (TCCR1A). We know that TCCR1A should have a bit pattern of $?????10$. Bit 3 and bit 2 of WGM are bit 4 and bit 3 of TCCR1B. Since we already know all the other bits of TCCR1B, we now know the following:

```
TCCR1B = 0x1a; // 00011010
```

Because hardware PWM is such a useful feature, we want to use all channel A, B and C for that purpose. Each channel has a 2 bit COM pattern to configure its characteristics. According to table 59 on page 130 of the datasheet, bit 1 and bit 0 of COM has the following meanings when the timer is set up for fast PWM:

- 00: normal port operation, OC1A/B/C disconnected. This means we are not using the PWM feature for this channel. You can enable and disable the channels independently.
- 01: same as 00 because we choose $WGM = 14$ (binary 1110_2)
- 10: clear on compare match, set when the timer overflows
- 11: set on compare match, clear when the timer overflows

It is clear that we want to use either 10 or 11. Mode 10 is more intuitive because a larger OCR (Output Compare Register) value means more time without the corresponding pin at a high voltage. *However*, some electronic devices are “active-low”, which means the device is on when the control signal is low, and off when the control signal is high. For these “active-low” devices, mode 11 makes more sense.

Assuming we want mode 10 for all three channels, we can now determine the value of TCCR1A (page 129):

```
TCCR1A = 0xfe; // 11 11 11 10
```

We are almost done. We still need to set up the timer to interrupt when the counter overflows. This is done by setting bits in TIMSK (Timer/counter Interrupt Mask Register). According to page 137, we want bit 2 set. Because TIMSK is also used to control timer0, we want to make sure bits unrelated to timer1 remains unchanged. The following code can be used:

```
TIMSK = (TIMSK & ~(0x3c)) | 0x04;
```

It is also a good idea to reset the counter TCNT1. Again, the counter is a 16-bit number, which means we need to write to the high byte first, then the low byte:

```
TCNT1H = 0;
TCNT1L = 0;
```

That's it! We are done with the initialization of timer1.
Let's look at the code so far:

```
TCCR1B = 0x1a; // 00011010
ICR1H = 40000U >> 8;
ICR1L = 40000U & 0xff;
TCCR1A = 0xfe; // 11 11 11 10
TIMSK = (TIMSK & ~(0x3c)) | 0x04;
TCNT1H = 0;
TCNT1L = 0;
```

Although this code makes sense from the perspective of working out the values, it is not the best sequence. With this sequence, the timer is “ticking” after the first statement. The following code is safer, because

- it first disables global interrupt (we'll talk about this later)
- it disables the ticking
- initializes everything,
- start ticking again,
- reenables global interrupt

```
_CLI(); // disable all interrupts
TCCR1B = 0; // disable ticking
TIMSK = (TIMSK & ~(0x3c)) | 0x04; // enable overflow interrupt
TCNT1H = 0; // reset counter
TCNT1L = 0;
ICR1H = 40000U >> 8; // set overflow value
ICR1L = 40000U & 0xff;
TCCR1A = 0xfe; // 11 11 11 10, set channel config
TCCR1B = 0x1a; // 00011010 start ticking
_SEI(); // reenables interrupts
```

4.2.2 PWM Configuration

The previous section initializes the timer so that channels A, B and C are set up for PWM. However, as the program runs, it can change the duty cycle of the PWM channels. This is done by setting OCR1A, OCR1B and OCR1C.

Each of these OCR (Output Compare Register) is a 16-bit number. So to initialize them, we need to write to the high byte first, then the low byte.

It is best to write a small subroutine to initialize these OCRs. You can write one for each channel, I am using channel A as an example:

```
void setOCR1A(unsigned value)
{
    OCR1AH = value >> 8;
    OCR1AL = value & 0xff;
}
```

Note that this code is not interrupt safe. All 16-bit timer registers use the same internal 8-bit buffer to store the high byte value. If an interrupt occurs between the two statements, and use the 8-bit buffer in the ISR, then OCR1A will be corrupted because when the low byte is written, the buffered high byte is already corrupted by the ISR.

Chapter 5

Interrupts and Interrupt Service Routines

As mentioned in the previous chapter, interrupts are like invocations of subroutines “by hardware”. This chapter follows up on this thread and expand on the topic.

5.1 Concept

Interrupts are essential concepts in modern computers (and controllers) because it relieves a processor from the responsibility to poll and check which hardware device requires attention. When a hardware device (such as a timer) requires attention (when the counter overflows), it requests attention from the processor. This request is an *interrupt*. The processor then responds to the request and perform whatever operation is necessary.

5.1.1 The global interrupt enable flag

It is sometimes important to disable interrupt. We’ll explain this later. In order *not* to listen for any request from any device, most processors have a flag that enables interrupt for all devices.

On the ATMega128 (as well as all AVR variants), this is a boolean flag called the I flag. It is a bit in the status register (SREG).

To clear the I flag, you can use the macro `_CLI()`. This disables all interrupts. To set the I flag (to enable global interrupt), you use the macro `_SEI()`. These two macros typically translate to assembly code.

It is not always the case that you want to disable or enable interrupts. For example, sometimes you need to disable interrupts, then *restore* the system to what it used to be. In other words, if interrupt was disabled originally, you don’t want to enable it.

To accomplish this, you can use a local variable to remember whether interrupt was enabled to begin with. This is done by the following code template:

```
{
    unsigned char oldSREG = SREG;

    _CLI(); // disable interrupt
    ... // do whatever
    if (oldSREG & (1 << SREG_I)) _SEI();
}
```

5.1.2 The interrupt vector

Although it is *possible* to have one single source interrupt, and use software to find out which hardware device requires attention, it is much better to allocate a “vector” for each device. This means that the processor automatically knows which code to execute, depending on which device requests an interrupt.

Most processors, including the AVR_s, use a vector table. A vector table, in C terms, is an array of pointers to functions. In plain English, a vector table is an ordered list of addresses of subroutines. Each hardware device that can interrupt has a unique entry in this table. When a device requires attention and that the processor is willing to respond, the processor automatically indexes to the correct entry in the vector table, and starts to execute whatever code is at the address.

Some processors can relocate this vector table, most low-end AVR_s must have this table located from location zero.

In C programming, you don’t have to deal with this vector table directly. The `SIGNAL` macros take care of the low level details.

5.1.3 Interrupt service routines

When the processor jumps to execute code to respond to an interrupt, it actually makes a call. In other words, the processor first saves the address of the next instruction onto the stack, then it jumps to the code to respond to an interrupt.

This is necessary because after the processor responds to an interrupt, it needs to know how to resume execution of whatever it was doing when it got interrupted.

Right before the processor saves the address of the instruction to resume to, it also disables interrupt. This is done automatically. This practice is common to most processors and MCUs because it is tricky to handle another interrupt when a processor is in the process of responding to the first one.

An ISR is typically short and does not take much time to complete. At the completion of an ISR, a special return instruction is used. This instruction returns to the saved address on the stack. This allows the processor resume execution of code that was interrupted. However, this special instruction also reenables interrupt *after the address to resume to is retrieved*.

This way, even if there are multiple devices request interrupts, we only need enough space to save one single instruction address.

5.1.4 Writing an ISR

It is relatively easy to write an ISR in C using `avr-gcc` and the `libc` package.

If you plan to write your own ISR, be sure to include the following lines at the top of your program files:

```
#include <avr/io.h>
#include <avr/signal.h>
```

The first line allows the compiler and header files automatically determine which MCU is being used, then automatically include the proper macro definitions. For a list of applicable interrupts for a particular MCU, do the following:

- locate where header files are placed. This is usually in `include/avr` from the main folder. For linux installations, the main folder is often `/usr/avr`, for WinAVR installations, the main folder is often `C:\WinAVR`.
- find the header file corresponding to the MCU. For example, the header file for the ATMega128 is `iom128.h`.
- use an editor or text viewer to inspect the file. Look for the pattern `SIG`.
- you should see a table of definitions of `SIG_`. Note that entry 0 is missing because that is the main reset vector.

Once you have found the name of the interrupt you want to handle, define a shell ISR as follows:

```
SIGNAL(SIG_OVERFLOW1)
{
}
```

This example is a shell to handle overflow interrupts from timer 1. Note that the macro `SIGNAL` automatically specifies the necessary compiler options so that the ISR uses the special return instruction at the end.

5.1.5 But what about INTERRUPT?

If you read `avr/signal.h`, you will notice that there is another macro called `INTERRUPT`. Indeed, you can define ISRs using `INTERRUPT` instead of `SIGNAL`.

However, ISRs defined with `INTERRUPT` have the global interrupt reenabled in the body of the subroutine. This is potentially very risky and it lead to bugs that are very difficult to reproduce.

I recommend using `SIGNAL` instead of `INTERRUPT` unless there is a strong reason.

5.2 Timer ISR

As mentioned in a previous section, you can define a shell ISR for a timer as follows:

```
SIGNAL(SIG_OVERFLOW1)
{
}
```

A very common thing to do is to keep track of a tick counter. This way, the rest of the program can get a sense of time. The first cut is simply to use a global variable as follows:

```
unsigned long tick_counter = 0;
SIGNAL(SIG_OVERFLOW1)
{
    ++tick_counter;
}
```

Then the rest of the program can track time as follows:

```
{
    unsigned long first_tick;

    while (PINA & (1 << PB1)); // wait for PB1 be pressed
    first_tick = tick_counter; // mark the time
    while (!(PINA & (1 << PB1))); // wait for PB1 be released
    duration = tick_counter - first_tick;
}
```

This code attempts to time how long a push button has been kept pushed. Even though `tick_counter` can overflow, the unsigned subtraction actually is insensitive to overflows. As long as the duration to measure does not exceed $2^{32} - 1$, `duration` still reflects the number of ticks that has passed.

This code does have a serious problem. As we read `tick_counter` (which is a 32-bit integer), the timer interrupt can occur. This is very bad. We can be halfway loading the 32-bit integer into registers when this happens. The ISR will complete without any problems and update `tick_counter`. However, as it returns, we resume to load the other part of the now-updated `tick_counter` into registers. This means half of the bits of the registers comes from the “original” `tick_counter`, while the other portion comes from the “updated” `tick_counter`. What if `tick_counter` wraps around to 0 in the ISR?

Besides, `gcc` is a smart compiler. It is likely to optimize the program and just assume that “nothing is going to happen to `tick_counter`, so it may just decide that `duration` *always* get 0. In other words, by default, `gcc` assumes a global variable is not modified by ISRs.

To fix this code, we need to use the following definition for `tick_counter`:

```
volatile unsigned long tick_counter;
```

The reserved word `volatile` tells the compiler that “anything can happen to this variable at any time”. However, we still have not fixed the first problem (getting interrupted when a multi-byte variable is getting loaded or stored).

The solution to the first problem is a little longer. We’ll define a subroutine first:

```
volatile unsigned long get_tick_counter(void)
{
    unsigned char oldSREG = SREG;
    unsigned long tmp;

    _CLI(); // disable interrupt
    tmp = tick_counter;
    if (SREG & (1 << SREG_I)) _SEI();
    return tmp;
}
```

Again, `volatile` tells the compiler that this subroutine has the potential to return something different everytime it is called. This subroutine has the wrapper to disable interrupt first, then restore the interrupt flag at the end. The key of this subroutine is that we capture the `tick_counter` value after interrupt is disabled. This guarantees the operation is “atomic”. The captured value in local variable `tmp` is then returned at the end.

Having defined `get_tick_counter`, we just need to change all references to `tick_counter` to `get_tick_counter()` in the program.

Part III

Basic Open Loop Motor Control

Chapter 6

Stepper Motors

6.1 Concepts

6.1.1 Full Stepping

Since this is not a hardware class, we'll only use bipolar stepper motors in our discussion. There are more unipolar stepper motors, but bipolar stepper motors are more efficient. As a result, most energy-conserving robots use bipolar stepper motors.

Figure 6.1 shows a bipolar stepper motor in its “schematic form”. In this diagram, there are two “phases”. Each phase is, essentially, a set of coils. The coils with terminals “A” and “C” represent one phase, while the coils with terminals “B” and “D” represent another phase. The arrow in the diagram represents the “rotor”, which is a freely rotating permanent magnet. In a real stepper motor, the permanent magnet is connected to the drive axle of a stepper motor.

As current passes through coils, a magnetic field develops. This magnetic field, in return, tries to align the permanent magnet (rotor) in a certain direction. Let's assume that when current passes from “A” to “C”, the magnet arrow points to the north. As we turn off coil A-C and turn on coil B-D (with current flowing from “B” to “D”), the magnet rotates clockwise to point to the east.

Here comes the fun part. If we turn off coil B-D, then turn on coil A-C in reverse (current flowing from “C” to “A”), the magnet rotates clockwise and points south. This is why it is called a “bipolar” stepper motor: each terminal can assume two polarities. Not surprisingly, we can also turn off coil A-C, then make current flow from “D” to “B” so that the magnet rotates and points west.

This completes one cycle. With a two-phase bipolar stepper motor, there are these four steps: A-C, B-D, C-A and D-B.

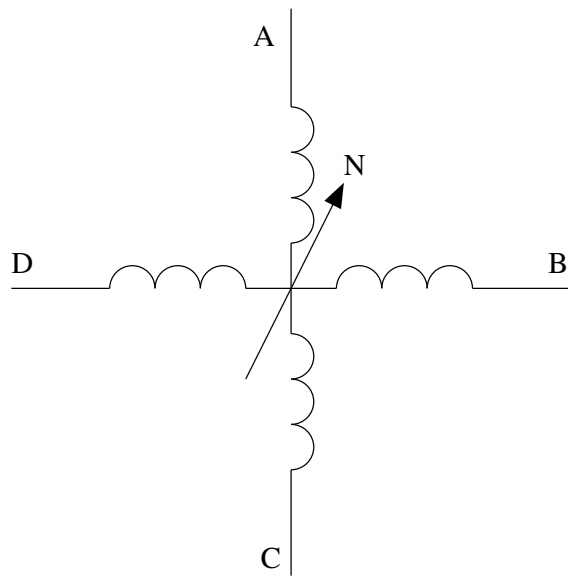


Figure 6.1: Schematics of a bipolar stepper motor.

6.1.2 Half Stepping

What happens when we enable both coil sets A-C and B-D? The magnet points north-east. This is called a half step because it is half way between the full steps A-C and B-D. You can also insert half steps between all the other full steps.

Half stepping is a useful concept because it effectively doubles the resolution of a stepper motor with no mechanical change. It also improves the torque of a stepper motor at the expense of some more current consumption.

6.1.3 Software Control

Since this is not an electronics class, I am going to skip the discussion of H-bridges.

From the software perspective, there are only two bits to control per phase. One bit indicates whether a phase is active or not, while the other bit controls the polarity of current. As a result, we can write the following table for a full cycle, using half-stepping:

| Phase 1 Enable | Phase 1 Polarity | Phase 2 Enable | Phase 2 Polarity |
|----------------|------------------|----------------|------------------|
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

In this table, a value of 1 for “Enable” means the phase is switched on. A value of 1 for “Polarity” means one way, a value of 0 means the other way.

As a program generates these steps patterns in this sequence, the motor rotates in one direction. To rotate in the opposite direction, one only needs to reverse the order of these steps. To make the motor spin faster, generate these steps more frequently, to make the motor spin slower, generate these steps less frequently.

At first glance, a stepper motor is difficult to control. However, due to the precise stepping nature, stepper motors are actually the easiest to control for open-loop control. Open-loop means there is no feedback utilized in the control code.

6.2 Frequency Division

Because the rotational speed of a stepper motor is proportional to the frequency of cycling through the stepping bit patterns, it becomes important to control this stepping frequency precisely.

The most intuitive method is simply to change the overflow value of a timer. This approach works to an extent, but it is not well suited for precise frequency control. The problem is because $f = \frac{1}{p}$, in which f is frequency, and p is period. In our case, p is always quantized by the divider of the timer clock.

Most stepper motors can step at 500Hz. This translates to a period of exactly 2ms. But how about 499Hz? It translates to a period of about 2.004ms. It may seem to make sense to set up the timer so we can control its period at 0.004ms intervals. But this is not true. At 333Hz, the period is 3.333ms, which is not a multiple of 0.004ms!

Using the period of a timer to directly control stepper also has other disadvantages. For example, a differentially driven robot will need two timers to control its two motors. As we will discuss later, acceleration control also needs its own frequency control. Does this mean that we need to use two more timers to control acceleration and deceleration? Soon, we are left with no timer to provide the basic monotonic tick for time keeping and other periodic logic.

This is why we need a different scheme for generating a variety of frequencies. We will use one single timer to do this.

The basic idea is that we begin with a timer overflow period that is much shorter than the shortest stepping period. In our example, let’s assume that a


```

sum += step_freq;
if (sum > base_freq)
{
    sum -= base_freq;
    perform_step();
}

```

In this code, `step_freq` is the stepping frequency (499Hz), whereas `base_freq` is the base frequency (5000Hz).

Intuitively, this code should work. As `step_freq` increases, it takes fewer iterations to overflow `sum`. To generate a 499Hz frequency (for stepping), this code should be executed at 5kHz (5000 times per second).

To make this logic cleaner, one can always make use of structures. The following code makes this frequency division logic flexible:

```

struct FreqDiv
{
    unsigned int freq;
    unsigned int sum;
    void (*cb_func)(void *);
    void *cb_param;
};

void FreqDiv_tick(struct FreqDiv *pfd)
{
    pfd->sum += pfd->freq;
    if (pfd->sum >= base_freq)
    {
        pfd->sum -= base_freq;
        if (pfd->cb_func) pfd->cb_func(pfd->cb_param);
    }
}

```

The only tricky part in this code are the use of the `cb_func` and `cb_param`. `cb_func` is a “call back” function whenever there is an overflow. `cb_param` is just a pointer to *something* that is passed to the callback function. This way, the call back function can have its own very complex structure for storing parameters and non-volatile state information.

Also, note that `base_freq` is not specific to the structure. This is because typically, all mechanisms that rely on frequency division can rely on the same timer, hence having the same base frequency.

6.3 Displacement, Velocity and Acceleration

When you control the motion of a robot, there are three main parameters. Displacement is the amount of distance traveled, velocity is the speed, and acceleration is the rate of change of speed.

This section explains some of the physics background of these terms, and relate the physics concepts to software coding.

6.3.1 Displacement

Displacement is the amount of distance traveled. In stepper motor control, this linear distance is translated into number of steps. The translation is linear, which means $D \propto S$, in which D is the linear displacement (in millimeters, inches or other length units), and S is the number of steps.

In stepper motor control, it is helpful to keep track of the number of steps yet to perform.

6.3.2 Velocity

Velocity is the rate of change of displacement. In mathematical terms, $v = \frac{dD}{dt}$. If we know that a robot is running at a constant velocity over a displacement D , then $v = \frac{D}{t}$, in which t is the amount of time to cover the displacement D (at constant velocity).

In stepper motor control, velocity is proportional to the frequency of stepping. In other words, $v \propto f_{\text{step}}$.

6.3.3 Acceleration

Acceleration is the rate of change of velocity. In mathematical terms, $a = \frac{dv}{dt}$. If we know that a robot changes its velocity at a constant acceleration, then $a = \frac{v}{t}$, in which v is the velocity after accelerating for t from stationary.

In stepper motor control, acceleration is proportional to the frequency of *changing (increment or decrement)* f_{step} . We call this $f_{f_{\text{step}}\pm 1}$.

You need one object of `struct FreqDiv` just to control acceleration. The call-back function `cb_func` should be responsible to increment or decrement the `freq` property of the `struct FreqDiv` for velocity control. You may also want to maintain a parameter structure (and initialize property `cb_param`) for acceleration control (to at least indicate whether it is acceleration or deceleration).

6.3.4 Getting back to Displacement

Mathematically, we can derive displacement as $D = \int v(t)dt$, in which $v(t)$ is the velocity at time t . If the velocity does not change (constant velocity), then

$$D = vt \tag{6.1}$$

in which v is the constant velocity, and t is the amount of time that the robot travels at v . This equation is useful when the robot is traveling at a constant velocity.

Another way to write velocity is $v = \int a(t)dt$, in which $a(t)$ is the acceleration at time t . If we assume acceleration is constant, then $v(t) = at$, in which a is the constant acceleration, and t is the amount time for acceleration.

Substituting $v(t) = at$ into $D = \int v(t)dt$, we can say that

$$D(t) = \int_0^t v(x)dx \quad (6.2)$$

$$= \int_0^t axdx \quad (6.3)$$

$$= a \int_0^t xdx \quad (6.4)$$

$$= \frac{at^2}{2} \quad (6.5)$$

$$= \frac{v(t)^2}{2a} \quad (6.6)$$

This equation is useful during acceleration.

6.3.5 Physical Parameters

What is a good top velocity? What is a good acceleration? It all depends on the physical characteristics of a robot. Without going into too much physics, we can always experiment and find out the maximum reliable velocity and acceleration/deceleration constants.

Let us use v_{\max} to represent the maximum velocity, and a_{\max} to represent the maximum acceleration. For simplicity, let us use $-a_{\max}$ as the maximum deceleration.

6.3.6 Velocity Profile

Given a distance D to travel, a robot needs to plan for acceleration, constant velocity travel, and deceleration.

Given what we know from 6.6 that the distance for acceleration is $D_{\text{accel}} = \frac{v_{\max}^2}{2a_{\max}}$. It takes the same distance to stop. In other words, the total distance to accelerate and decelerate is $D_{\text{ramp}} = \frac{v_{\max}^2}{a_{\max}}$. What is $D < D_{\text{ramp}}$?

If this is the case, it means that we need to stop accelerating and immediately begin deceleration before we reach v_{\max} .

Instead of predetermining the speed profile, we can do the following (in pseudocode). Assume v is the current velocity.

```

if  $\frac{v^2}{2a_{\max}} \geq D$  then
  start deceleration at  $-a_{\max}$ 
else
  if  $v < v_{\max}$  then
    start acceleration at  $a_{\max}$ 

```

```

else
    stop acceleration
end if
end if
end if

```

Sometimes it is necessary to change the target velocity. As a result, we usually want to use another term, v_{set} , to represent the desired velocity. But this means that the current velocity can exceed v_{set} . Our logic needs to be modified as illustrated in algorithm 1. Note that in this algorithm, D , v_{set} can both be changed dynamically.

Algorithm 1 Code to control velocity and acceleration based on remaining displacement

```

if  $D = 0$  then
    clear acceleration/deceleration
    set  $v$  to 0
    optionally notify completion of motion
else if  $(D > 0) \wedge (\frac{v^2}{2a_{\text{max}}} \geq D)$  then
    set  $v_{\text{set}} \leftarrow 0$ 
    start deceleration at  $-a_{\text{max}}$ 
else if  $(D < 0) \wedge (\frac{v^2}{2a_{\text{max}}} \geq -D)$  then
    set  $v_{\text{set}} \leftarrow 0$ 
    start acceleration at  $a_{\text{max}}$ 
else
    if  $v < v_{\text{set}}$  then
        start acceleration at  $a_{\text{max}}$ 
    else if  $v > v_{\text{set}}$  then
        start deceleration at  $-a_{\text{max}}$ 
    else
        switch to constant velocity
    end if
end if

```

Note that this code also handles positive and negative D and v_{set} .

In a program, v should be replaced by f_{step} , and v_{set} should be replaced by an appropriate frequency term. Furthermore, the acceleration term should be replaced by $f_{f_{\text{step}} \pm 1}$, and a_{max} replaced by an appropriate frequency term determined by experiment.

Note that algorithm 1 should be executed independent of the stepping logic, but within the same tick. This logic should also be invoked whenever D or v_{set} is changed.

6.3.7 C code framework

You can code the velocity profile logic anyway you want. However, from the organization point of view, I suggest you use a structure to include all the

properties:

```

struct MotionProfile
{
    struct FreqDiv vel; // velocity control
    struct FreqDiv acc; // acceleration control
    long D; // displacement
    int vset; // desired velocity
    int amax; // maximum acceleration
    int v_sign; // sign of velocity
    int a_sign; // sign of acceleration
};

```

Given that `pmp` is of type `struct MotionProfile *`, you should initialize it as follows:

```

pmp->vel.freq = 0; // initially stationary
pmp->vel.sum = 0; // reset to zero
pmp->vel.cb_func = step_func; // step_func performs a step
pmp->vel.cb_param = pmp; // it is given the entire struct
pmp->acc.freq = 0; // initially no acceleration
pmp->acc.sum = 0; // reset to zero
pmp->acc.cb_func = accel_func; // accel_func increments/decrements fstep
pmp->acc.cb_param = pmp; // it is also given the entire struct
pmp->amax = ...; // whatever constant works
pmp->D = 0; // initially nowhere to go
pmp->vset = 0; // initially stationary

```

In this example, the function `step_func` must handle positive and negative velocities. While `pmp->vel.freq` represents the *magnitude* of velocity, `pmp->v_sign` represents the direction. Similarly, the function `accel_func` must take into account the sign of acceleration, stored in `pmp->a_sign`.

`step_func` has to handle the sign crossover of `pmp->D`. This is easy because `pmp->D` is a signed number. `accel_func` has to handle the sign crossover of velocity magnitude `pmp->vel.freq` and velocity direction `pmp->v_sign`.

6.3.8 What about differential steering?

You can treat each motor independently in a differentially steered, and you can use one `struct MotionProfile` for each motor. This also means you probably need different versions `step_func` and `accel_func`, one for each motor.

This approach works fine when the two motors are symmetric and have exactly the same profile. However, for error correction and turning, it is often necessary to speed up one and slow down the other one. As a result, treating two motors independently is often a bad idea, leading to many problems down the line.

From the top level, it is best to look at motion from two physical perspectives. First, there is linear motion. Second, there is angular motion.

Consider the following scenario. Your robot is heading north (zero degree). It makes a 90-degree turn clockwise using differential steering at an average velocity of 300mm/s. Also, let us assume the turning radius is 600mm, and each wheel is 100mm from the center of the robot.

This means the left wheel needs to move at $300\text{mm/s} \times \frac{600+100}{600} = 350\text{mm/s}$. The right wheel, on the other hand, needs to move at $300\text{mm/s} \times \frac{600-100}{600} = 250\text{mm/s}$.

At the same time, you can also express this motion as a linear speed of 300mm/s, with an angular speed of 100mm/s. The linear speed is easy to interpret. However, the angular speed needs some explanation. The angular speed is the difference of speed between the two wheels.

Once you separate into these two profiles (linear and angular), each profile has its own parameters. As a result, each profile also ends up with its own displacement, target velocity and velocity. It is also important to know that the limits (v_{\max} and a_{\max}) for these two profiles will be different.

In our example, we keep the target velocity for the linear profile as 300mm/s, and its displacement is kept the same. However, for the angular profile, the displacement was originally zero. It is then set to $200\text{mm} \times \pi \div 2$, which is approximately 314mm. The target velocity for the angular profile is set to whatever maximum that the robot is capable of. The logic of 1 automatically takes care of the angular aspects of turning.

Here comes our next question. So far, we don't have anything that is useful for controlling the motors.

The linear and angular motion profiles need to be combined so we can control the stepper motors. This can be done easily. After all, the angular profile describes *how the two motors differ*. As a result, $v_L = v_{\text{lin}} + \frac{v_{\text{ang}}}{2}$, and $v_R = v_{\text{lin}} - \frac{v_{\text{ang}}}{2}$.

What does this mean for our C code? As it turns out, we don't need to track the displacement or acceleration for the individual motors anymore. Instead, we only need the velocity (step frequency) of each motor.

We can now create our macro structure for a differentially steered robot as follows:

```
struct DiffSteer
{
    struct MotionProfile linear;
    struct MotionProfile angular;
    struct FreqDiv left;
    struct FreqDiv right;
};
```

Assuming `ds` is a `struct DiffSteer`, we need to update `ds.left.freq` and `ds.right.freq` whenever `ds.linear.vel.freq` or `ds.angular.vel.freq` updates. This is easy, since we can put this logic in the call back functions `ds.linear.acc.cb_func` and `ds.angular.acc.cb_func`.

However, you need to track the sign/direction of each motor separately. It is not necessary to track this as a separate property. You can use `ds.linear.vel.freq`,

`ds.linear.v_sign`, `ds.angular.vel.freq`, `ds.angular.v_sign` to determine the direction of each motor.

6.3.9 Implementational Alternatives

Instead of using unsigned types for `sum` and `freq`, you can consider using signed types. If you do this, you don't need to track the signs of acceleration and velocity separately, and it simplifies comparisons. However, this also means that you need to change the way you compare `sum` with `base_freq`, as well as how you reset `sum` once it exceeds the limits.

6.4 Coding Tips

6.4.1 Efficiency issues

Don't worry about efficiency until you have the algorithm working. Once you have the algorithm working, you can start to consider these issues. On a powerful processor, you may need to worry much about code efficiency. However, with an 8-bit processor that has no division instructions, some hassle in optimization can mean dramatic improvement of performance.

Let us review algorithm 1. $\frac{v^2}{2a_{\max}} \geq D$ is a fairly expensive operation. First, v^2 involves multiplication. The killer part is division by $2a_{\max}$. Division is a bit more expensive than multiplication.

Let's deal with one problem at a time. To avoid division, we can change the comparison as follows:

$$\frac{v^2}{2a_{\max}} \geq D \Rightarrow \quad (6.7)$$

$$v^2 \geq 2Da_{\max} \quad (6.8)$$

This is true because a_{\max} is a positive quantity.

But, doesn't this still involve two multiplications in $2Da_{\max}$?

In general, the answer is yes. If a_{\max} is a constant, then we can roll $2a_{\max}$ into a single constant to get rid of one multiplication. D is a variable, so we cannot roll this into a constant.

However, we can still do some tricks to avoid having to multiply D to $2a_{\max}$ every time. Remember what D represents: it is the number of steps to perform. This means *most* of the time, D is incremented or decremented by 1. This is useful, because now we can use a variable for a scaled D . Whenever we decrement 1 from D , we subtract $2a_{\max}$ from the scaled version. Note that we still need to multiply when we reset D to an arbitrary number.

How about the computation of v^2 ? It certainly appears that we cannot easily get rid of the multiplication here.

Once again, think of what v represents. This is the frequency of stepping f_{step} . Most of the time, if not all, f_{step} is only incremented or decremented

by 1 (in the acceleration logic). This is a very useful fact because we can now eliminate multiplication.

If we know $(v - 1)^2$, we can compute v^2 as follows:

$$(v - 1)^2 = v^2 - 2v + 1 \tag{6.9}$$

$$v^2 = (v - 1)^2 + 2v - 1 \tag{6.10}$$

$$= (v - 1)^2 + v + v - 1 \tag{6.11}$$

See, no multiplications! Using a similar technique, you can also derive $(v-1)^2$ using no multiplication from v^2 .

Although this optimization can save you a lot of processing time, you do need to track quite a few more properties in the structures. As a result, I recommend the use of abstract data types so that you use a single function to increment/decrement v . This same function is also responsible to track a separate variable for v^2 . The same applies to D and its scaled version.

Chapter 7

Project 1

In this project, you need to write a program that controls a differentially steered robot using two stepper motors. I realize that we do not have any actual robots. On the other hand, it is probably easier that way because on-controller debugging is difficult for mobile robots.

7.1 Final Objectives

One of the final objectives of this project is a piece of code that works for practically all differentially steered robots using stepper motors. To make your program easier to debug without robots, it must product suitable output to standard output (`stdout`) so that you can use another program to analyze the output.

The following describes the output of your program:

- Any text on a line after a pound (`#`) symbol is ignored, use this for commenting (and debugging).
- An event is recorded on one line. A motor performing a step is an event. Two events at the same exact time takes two lines.
- Time is measured in ticks. Use an unsigned long integer (32-bit) to count ticks.
- Each line begins with a tick count, expressed as a decimal unsigned number. Use a space character to separate fields.
- Use the letter “L” (uppercase) to indicate a step of the left motor.
- Use the letter “R” (uppercase) to indicate a step of the right motor.
- Although your program doesn’t need to know the physical characteristics of the robot, it may be helpful to know this if you want to visualize the path of the robot. Assume the following physical characteristics:

- the base frequency is 5kHz (5000 times per second)
- one stepper motor step moves 1mm linearly for the attached wheel
- each wheel is 20mm from the center of the robot
- For example,

```
14334 L
```

means that the left motor performs a step in time slice 14334.

The following describes the input of your program:

- Each line of input is one setting, there can be multiple settings per time slice.
- Everything after the pound symbol (#) on a line should be ignored.
- Fields are separated by spaces.
- The following names are used for specifying parameters:
 - `lin` is a prefix for linear parameters
 - `ang` is a prefix for angular parameters
 - `vel` sets the desired top velocity (in number of steps per second), this is a magnitude, so it should be an unsigned integer.
 - `acc` sets the maximum acceleration (in number of steps per second squared), this is a magnitude, so it should be an unsigned integer.
 - `D` sets the remaining number of steps. This can be positive or negative.
- A parameter is set by the parameter name, followed immediately by a possibly signed decimal integer for the value
- For example,

```
5621 lin.D 234
```

means that at time slice 5621, reset linear displacement to 234.

7.2 Step by Step

Write your program step by step. Test and debug as necessary in each step before moving one. Also, copy and paste code from my notes as you see fit. Copy and paste from sources other than my notes is not permitted.

7.2.1 Set up the Tools

My recommendation is use `gcc`. This makes sure your code is relatively portable to `avr-gcc` when you are all done. You can use `gcc` in Linux, FreeBSD or as a component of `Cygwin`.

If you don't want to use `gcc`, Borland C (installed at the lab) probably works okay. However, you need to make sure your program compiles in `gcc` when you turn it in. I will not port programs to `gcc`, but I can help you diagnose problems at the lab.

If you use Windows and don't want to install a new OS just for this class, you have two main alternatives. First, you can use the Knoppix CD (or some other live Linux/FreeBSD live CD) so you can run Linux/FreeBSD without installing it.

Second, you can install Cygwin from <http://www.cygwin.com>. Cygwin is a collection of files that emulates Linux on a Windows platform.

In other case, I can burn CDs for Knoppix or Cygwin.

For a project this big, it may be worthwhile to look into version control tools. RCS (revision control system) is available with Linux, FreeBSD, Cygwin and also native Win32. It is relatively easy to set up and use. This is not required, and it does incur a little bit of overhead initially. However, I think it will save you time in the end, and it's a good skill set to have for the rest of your programming career.

If there is enough interest, I'll write a small section about RCS and include it in this "book".

7.2.2 Input/output

Write simple subroutines to read from a sample input file and write to a sample output file. The input file is probably more difficult to deal with because it requires some parsing (minimal). The output is easy, it should take you about 3 minutes at the most.

7.2.3 Frequency Division and Velocity

Implement and test your frequency division code first. If this part isn't working, don't bother with the rest. After you get frequency division to work, implement velocity control. Don't worry much about displacement at this point. Set the initial displacement (remaining number of steps) to a huge number. You probably also want to limit the number of time ticks in the simulation.

Focus on just the linear parameters at first.

7.2.4 Acceleration and Deceleration

Once you get velocity to work, implement acceleration and deceleration. This requires the ability to parse velocity change input lines (and also the initial acceleration setting). Again, don't worry about displacement by setting it to a huge number.

Be sure to test your logic by setting the target velocity high and low. Your program should respond by increasing and decreasing the velocity to the target velocities using the specified acceleration.

7.2.5 Displacement Control

After you have acceleration logic implemented, you can incorporate displacement control. Now, your program needs to know when to decelerate as the remaining number of steps becomes less than the distance required for deceleration. Furthermore, if the displacement is changed (increased) again, make sure your program responds by speeding up to the target velocity.

At this point, assume displacement is always positive and get your program to work first.

7.2.6 Handle Negative Displacement and Velocity

After your program works perfectly for linear control for positive displacements, extend it so it can also handle negative displacement.

7.2.7 Add Left and Right Motors

At this point, your program only understands linear parameters. Add the concepts of left and right motors so that you can command the motors to spin independently. Because there is no angular parameters, your motors should be fully synchronized.

7.2.8 Add Angular Control

Angular motion profile control is identical to linear motion profile control, as far as logic is concerned. The settings is probably different. Since the settings (maximum velocity and acceleration) are read from standard input anyway, this should not be much of an issue.

When you incorporate angular control, the left and right motors should start to spin differently when there is angular motion involved.

7.3 Due Date and Grading

This project is due three weeks from the assignment date, and it is worth 300 points (probably about a quarter of all the project points).

Your program is graded by running feeding test input files and comparing output files with the proper answer. I will make some sample test cases available as soon as possible. The actual test cases I use for grading will not be disclosed. I'll probably also write a simple program that plots the course graphically from the output of your program so you can visualize the path.

The score of your project is based on the proportion of test cases that work. This means a program that works for all test cases get 100% of 300 points (plus

or minus early or late adjustments). A program that works for half of the test cases get 50% of 300 points, and a program that does not work for any test cases get 0% of 300 points. Note that a program that does not compile automatically gets 0%.

7.4 How to Turn it In?

Turn in this program as an attachment by email. If you use multiple files (*highly* recommended to keep yourself sane!), use either the ZIP format or TAR format (with or without GZIP or BZIP2 compression) to combine files into one archive.

Send your email with the following subject line (wrong subject line gets 40 points deducted):

CISP299 Project 1 by your name

Send your email to tauyeung@drtak.org.

Chapter 8

R/C Servo Motors

8.1 Concepts and Rationale

R/C servo motors are self contained units used mostly for remote control vehicles (airplanes, boats and cars). An R/C servo motor has a very simple physical interface consisting of three wires. Two wires supply ground and power, while the third one carries the control signal.

8.1.1 Components of an R/C Servo

Interface: three wires. As mentioned, one for power, one for ground and one for control.

Motor: R/C servos use tiny DC motors internally to convert electrical energy to motion. There is a wide variety of these DC motors used in R/C servos. The inexpensive ones are quite crude, while expensive ones use coreless high efficiency DC motors.

Control circuit: a small circuit board holds all the electronic components necessary to control the DC motor of an R/C servo.

Gears: R/C servos have very fine and precise gearing. Inexpensive ones have plastic gears, while expensive ones have metallic ones.

Potentiometer: a potentiometer is used to encode the angular position of the output shaft of an R/C servo.

8.1.2 Control Mechanism

The control signal idles at 0V. A square pulse peaking at 5V with a width of 1ms to 2ms (for most R/C servos) specifies the desired angle. 1ms specifies one extreme, while 2ms specifies the other extreme. The angular difference corresponding to the 1ms is about 180 degrees. In other words, if we consider the angle at 1.5ms is 0, 1ms specifies an angle of -90 degrees, while 2ms specifies an angle of +90 degrees.

Upon reception of the falling edge of this 1ms to 2ms pulse, the R/C servo control circuit compares the potentiometer output with the command. If there is any difference, the control circuit turns on the DC motor to correct the error. The correction lasts about 20ms, then the R/C servo control mechanism idles again.

Note that the correction may be strong or weak, depending on the difference between the control signal and the potentiometer output.

If an R/C servo is under sufficient load that the angular position changes without active correction, the pulses should be spaced no more than 20ms apart. If the control pulses are too far apart, it is possible for the R/C servo to creep a little bit before correction is made by the next control pulse.

8.1.3 Why use R/C servos?

For any robot that needs limited angular control, R/C servos are good choices. This is because R/C servos can be inexpensive (less than \$10 each, as of 2004), and they come in all sizes and capabilities. It is relatively easy to mount a GP2D12 sensor (or any other ranging sensor) on an R/C servo for scanning purposes.

R/C servos are also commonly used in smaller legged robots for actuating the legs and other limbs.

Interestingly, R/C servos are also used in some small wheeled robots. Most R/C servos can be modified for full rotation (instead of -90 degree to +90 degrees).

The main advantage of robots utilizing modified R/C servos for full rotation is packaging and size. The standard size R/C servos pack lots of torque in a small package. Furthermore, despite the high gear ratio, most R/C servos have little play. The low voltage requirement of R/C servos is also an attractive features.

8.2 Modification for Full Rotation

As mentioned, R/C servos are designed to rotate from -90 degrees to +90 degrees, and the control signal is meant to control the angle of the output shaft. This means that R/C servos are not useful for wheeled robots that require gear motors that rotate 360 degrees.

As it turns out, most R/C servos can be easily modified for full rotation. Regardless of the brand and basic design, here's the general method:

- open the case, disassemble the gears
- on the output (final) gear, remove the tab that stops it from free rotation
- gain access to the printed circuit board
- desolder the potentiometer, you have two options to go from here:

- replace the potentiometer with a series of two 5k Ohm resistors. The mid point connects to the pad/through hole of the center tap of the removed potentiometer, while the two ends connect to the pad/through hole of the two poles of the removed potentiometer.
- use wires to connect the pad/through hole to the removed potentiometer so that the potentiometer is external to the R/C servo for tweaking.
- reassemble

After you perform this modification, send 1.5ms pulses to the control signal. If you bring the removed potentiometer out with wires, you can tweak the “center” position until the motor stops spinning.

With the modification, the R/C servo is tricked to think it is physically at the center position (0 degree). When you send pulses of 1.75ms, for example, the R/C servo control circuit tries to correct the “error” by turning on the DC motor to spin one way. When you send control pulses of 1.25ms, on the other hand, the R/C servo control circuit rotate the DC motor the other way to try to correct the error.

8.3 PWM Control

Although one can use software to produce the pulse signal required by R/C servos, it is much better to use dedicated hardware to do so.

For example, Parallax BASIC Stamps use the software approach. While it is easy to specify the pulses, it is difficult to send the pulses and perform other tasks at the same time. You *can* control multiple R/C servos at the same time with a BASIC Stamp because the pulses for each R/C servo need to be spaced out by 20ms.

In the ATmega128 (and some other AVR variants), the built-in PWM component of the timers can be used to generate the necessary pulses for R/C servo control. Timer 1 and Timer 3 each has 3 channels of PWM, providing a total of 6 channels of PWM waveforms.

To utilize a timer and its associated PWM ability to control R/C servos, the timer must be configured as follows:

- period set to about 20ms
- enough resolution for pulse widths from 1ms to 2ms

This means we want to have as many counts as possible in 20ms. Because timer1 and timer3 both have 16-bit timer counters, the largest count value is 65536. $\frac{20\text{ms}}{65536}$ is approximately $0.3\mu\text{s}$. Assuming the master clock is 16MHz, this means we need a prescale constant of $0.3\mu\text{s} \times 16000000$, which is approximately 4.88.

The closest prescale constant is 8. Each counter increment then takes $\frac{8}{16000000\text{Hz}}$, which is $0.5\mu\text{s}$. The resolution of the pulse is then $\frac{1\text{ms}}{0.5\mu\text{s}}$, which

is 2000 steps over 1ms. This provides sufficient precision over 180 degrees of rotation.

The period of the pulses should be 20ms. The count value for 20ms is $\frac{20\text{ms}}{0.5\mu\text{s}}$, which is 40,000. Register ICR can be initialized to 40,000 to control the period of the pulses.

The timer should also specify fast PWM operation for each output compare (OC) output pin.

8.4 Issues and Tips

Although R/C servos (especially ones modified for full rotation) have many advantages, they do have problems.

8.4.1 Poor Speed Control

Most R/C servos modified for full rotation are used as on/off devices. In other words, a controller either request full speed forward, stop, or full speed backward.

While some R/C servos exhibit somewhat proportional responses when the control pulse is close to 1.5ms, the response is entirely dependent on manufacturer and model. In other words, there is no standard like 1.55ms means 50% of full speed.

A *correct* implementation of R/C servo control should not even have a predictable proportional response based on pulse width. As a result, proportional response can only be expected from lower end R/C servos.

8.4.2 Uneven Full Rotation

Some R/C servos do not rotate evenly after modified to rotate 360 degrees. This is hardly surprising because an R/C servo is not designed for smooth full circle rotation! In fact, some miniature R/C servos only has one half of a full gear at the drive axle.

8.4.3 Fragile Plastic Gears

At least in inexpensive R/C servos, the plastic gears are fairly fragile. A competition robot can easily break these gears. Although replacement gears are available, it is still a hassle to have to repair a servo. This is especially the case when the mechanism is difficult to access.

More expensive R/C servos use metallic gear and ball bearing for the main shaft, which significantly improves the longevity and reliability of the device.

8.4.4 Not Meant for Continuous Use

Most R/C servos are designed for steering, flapper control and other intermittent tasks. Although an R/C servo needs to exert a large amount of torque to actuate

something, it needs relatively little torque to hold position due to friction and the high gear ratio.

When R/C servos are modified for full rotation, they often have to operate continuously. This is not a problem when the continuous motion does not require much torque. For example, desktop robots or robots designed to run on smooth and level surfaces require little torque to maintain a fixed speed.

However, for robots that run on rough surfaces or push a high load (such as sumo-type competition robots), R/C servos may overheat and burn off winding insulation. This causes shorts in the motor, which can potentially blow components on the control circuit board.

Part IV

DC Motor and Closed Loop Motion Control

Chapter 9

DC Motor Control

This is a relatively short chapter on DC motor control. It is short because there is not much to discuss! Throughout this discussion, we assume a DC motor is just the usual brushed kind. Brushless motors require a bit more complicated control.

9.1 DC Motor Direction Control

To control the direction of rotation of a DC motor, a controller only needs to control the polarity of the electrical potential applied to the terminals of a DC motor. This is typically done by a device called an H-bridge. The construction of an H-bridge is out of the scope of this class. From the software perspective, we use one bit to control the direction of torque of a DC motor, that's it!

9.2 DC Motor “Strength” Control

I put quotes around strength because the torque of a DC motor is controlled by the current passing through the device. This is, generally speaking, difficult to achieve. As a result, we only control the *average* voltage applied across the terminals of a DC motor.

9.2.1 Analog Electrical Potential Control

You can change the continuous voltage across a DC motor by a transistor. Without going into details, this can be done with a N-channel MOSFET with an opamp, or a bipolar junction transistor (BJT) with a variable bias current.

In the past, it is difficult to adjust the continuous voltage from a controller without a digital-to-analog converter (DAC). This is much easier these days because of digitally controlled potentiometers. In other words, it is possible to control a DC motor by controlling the continuous voltage.

However, this approach has a problem. Power dissipation is $P = VI$. In other words, the power that needs to be dissipated as heat is proportional to the product of the voltage drop across a device and the current passing through the device. Let us consider a modest DC motor that draws up to 4A at 12V. In order to produce 6V across the motor, the other 6V must drop across the transistor (N-MOSFET or NPN BJT). The current consumption at 6V is probably about 2A. This means the power dissipated by the transistor is 12W.

12W does not seem to be much. However, if this much power needs to be dissipated by a small device like a transistor, it is a lot! If you try to control the continuous voltage across a DC motor, you'll need lots of heatsink area and even forced air cooling (use a fan to blow on the heatsink).

9.2.2 PWM

Pulse width modulation means a device is either fully on (100%) or completely off (0%). The period of PWM is the length of a “cycle”, and this is fixed at some relatively small amount of time. For DC motor control, a period of 5ms is suitable for smaller motors, while bigger ones can have periods up to 20ms.

Within a cycle, a controller controls the proportion of time in which the device is on. In other words, for a motor not to do anything, the duty cycle is 0%. To utilize all the torque available, the duty cycle should be 100%. A controller can also adjust the duty cycle to any point between 0% and 100% to change the *average* voltage applied to the motor.

How is this better? Let's try to produce an average of 6V across a motor, just like in the previous subsection.

The available voltage is 12V, this means we want to have a 50% duty cycle. In other words, half of the time, the transistor is fully on. When a transistor, be it N-MOSFET or NPN BJT, is fully on, the power dissipation on the device is at a minimum. A modest N-MOSFET has a resistance of about 0.1Ω . This means the power dissipated at the transistor is $P = I^2R = 16 \times 0.1 = 1.6W$ when the transistor is turned on at 100% duty cycle. At 50% duty cycle, the power dissipation is exactly one half at 0.8W. 0.8W can be dissipated by a TO-220 package without external heatsink at room temperature. We just reduced our power dissipation from 12W to 0.8W!

9.3 Are we done?

Since most controllers can generate PWM waveforms, shouldn't we be done?

Unfortunately, DC motors require feedback for precise control. What we have discussed up to this point is merely “throttling”. You cannot drive a car blind-folded with just a gas pedal and a brake pedal! Similarly, just being able to PWM a DC motor does not mean we can control the rotational speed. The actual rotational speed of a motor depends on many factors, including internal friction (it changes over time), terrain, battery output voltage (also changes over time), switch efficiency and many other factors.

In short, we have little idea of how fast a motor is actually rotating!

Chapter 10

Motion Encoding

The previous chapter introduces PWM control for DC motors. We also realized that we cannot deduce the rotational speed of a DC motor just using the PWM duty cycle. In fact, there is no way to theoretically compute the rotational speed of a DC motor precisely!

We need to rely on feedback to let a controller know exactly how fast a DC motor is rotating. More precisely, we need some feedback to know how fast a robot is moving.

10.1 Optical Encoding

Optical encoding relies on an encoding disc and an encoding sensor. The encoding disc is usually an opaque disc with slots that let light pass through. An encoding sensor has two sides. One side has an Infrared Emitting Diode (IrED), and the other side has a phototransistor. The encoding disc is placed between the IrED and the phototransistor. This way, as the disc rotates, the phototransistor sees transitions of light and darkness because the slots of the disc have motion relative to the sensor.

Optical encoders are commonly found in devices that operate in clean environments. Some printers use DC motors with optical encoding instead of stepper motors because it is less expensive. Small robots that operate in controlled (indoor) environments also use optical encoders.

However, optical encoding cannot be used in dirty environments. Dust, sand, oil droplets, gunk can all easily render optical encoders useless. Most larger robots that operate in outdoor environments cannot use optical encoding unless the encoder is concealed (air-tight). This is still possible because motion can be clutched by magnets so that there is no mechanical linkage from the axle to be encoded to the encoder axle.

10.2 Magnetic Encoding

Recent semiconduction development has made Hall-effect sensors very cost effective. There are several kinds of hall-effect sensors. The most common type is effectively a semiconductor “reed-relay”. It is a transistor that is turned on if and only if the magnetic field perpendicular to it exceeds a threshold.

With hall-effect sensors, we can encode motion by placing small magnets on a wheel or a gear. The encoding resolution is somewhat limited with this approach because magnets are fairly big (compared to a slot on an optical encoder disk). Furthermore, because the magnet is on the rotating part, it can fall off when the rotational speed is high.

An alternative is to place a “bias” magnet near the sensor, then use a metallic (with iron component) gear or slotted disc for encoding. The teeth of the gear serves as magnetic field concentrators. As the gear turns, the magnetic field experienced by the sensor changes, and these transitions indicate motion.

See [http://rocky.digikey.com/WebLib/Melexis Web Data/MLX90217.pdf](http://rocky.digikey.com/WebLib/Melexis%20Web%20Data/MLX90217.pdf) for more information.

Magnetic encoding is not affected by most dirt, dust and oil. It is suitable for robots that need to operate in dirty or outdoor environments.

10.3 Quadrature Encoding

Regardless of the sensor type, if it outputs a 50% duty cycle waveform, quadrature encoding can be used.

In quadrature encoding, two sensors of identical properties are used. In other words, both sensors must have the same 50% duty cycle output waveform. The two sensors are spaced out by a quarter period. In other words, if it takes x rotation for one period (one half on, one half off), the two sensors are spaced by $(n + 0.25)x$ rotation. n can be any integer, as multiples of periods does not change the shape of the output.

Let us call one sensor A and the other one B. Once A and B are spaced out by a quarter period, we get four transitions instead of two per period. Furthermore, we can also tell the direction of rotation. If waveform A leads waveform B, it indicates rotation in one direction. If waveform B leads waveform A, it indicates rotation in the other direction.

This means that with two sensors, we can keep the same encoder disc (optical or magnetic), but we end up with:

- twice the resolution
- the ability to tell direction of rotation

10.4 Implementation

10.4.1 General Approach

Regardless of the encoding technology (optical, magnetic or even contact based), a program needs to know when a transition occurs.

Without any hardware peripherals to assist, a program can still rely on a timer ISR to sample an encoder input line. The logic to do this was already discussed when we discussed general purpose input. Debouncing is usually not needed nor desired.

Detecting edges (transitions) this way is general enough that it'll work on most platforms. However, it is also limited in terms of performance. An interrupt frequency of 5kHz is considered fairly high. Even at this frequency, a program can only detect 2500 transitions per second. The frequency is divided by 2 because it takes two samples to detect one transition.

With a fine optical encoder disk with 500 slots, one revolution yields up to 2000 transitions because there are 4 transitions per period with quadrature encoding. As a result, we can only correctly interpret transitions up to a speed of 1.25r/s, which isn't exactly fast.

10.4.2 External Interrupts

The ATMega128 has many external interrupt lines. An external interrupt pin is a general purpose I/O pin as far as configuration is concerned. However, an external interrupt pin also has the ability to cause an interrupt when it senses a transition. This is great for encoding purposes, because we only need to pay attention when there is a transition.

Different members of the AVR have different external interrupt designs. On an ATMega128, an external interrupt can be set up to sense *either* a rising edge or a falling edge. Although this appears to be a problem, it is not.

Except for the first transition, all other transitions (rising or falling) can be captured correctly. This is because the ISR can flip the edge sensing polarity! In other words, after sensing a falling edge, the ISR can flip the polarity to sense a rising edge. This makes sense because a falling edge *must* be followed by a rising edge before another falling edge occurs.

Although external interrupts only occur when there is a transition, there is still a maximum number of transitions that the processor can handle. For very high speed or high resolution encoding, ISRs should be written in assembly language to minimize the overhead (mostly for saving and restoring registers).

10.4.3 Algorithm for Quadrature Encoding

The general logic for interpreting a quadrature encoder input pin is as follows (assume we are reading channel "A"). Also assume that this code is only invoked when we know there is a transition at channel "A".

if "A" has falling edge **then**

```

if "B" is high then
    "A" leads "B", increment motion tick counter
else
    "A" follows "B", decrement motion tick counter
end if
else
    if "B" is high then
        "A" follows "B", decrement motion tick counter
    else
        "A" leads "B", increment motion tick counter
    end if
end if

```

While this code may seem difficult to implement, the AVR has some instructions that can read a pin and branch in two clocks. For those who are interested in assembly programming, these instructions are `sbic` and `sbis`, for "skip if bit of I/O is cleared" and "skip if bit of I/O is set", respectively.

With these instructions, assuming channel "A" is pin X of port Z, and channel "B" is pin Y of port Z, the corresponding assembly code is as follows:

```

    sbis Z,X
    rjmp falling_edge
    ; my state is high, rising edge
    sbis Z,Y
    ; leading
    rjmp increment
    rjmp decrement

falling_edge:
    sbis Z,Y
    ; following
    rjmp decrement
    rjmp increment

increment:
    ; code to increment counter
    ret

decrement:
    ; code to decrement counter
    ret

```

This code only takes about 8 clocks to decide whether it needs to increment or decrement. This translates to $0.5\mu\text{s}$. The code to increment or decrement a counter takes about 10 clocks. As a result, the entire operation should be done within $2\mu\text{s}$ given a master clock of 16MHz.

I knew assembly programming is good for something.

10.5 Electronic Interface

Some optical encoders have built-in conditioning circuits so that the output is TTL/CMOS compatible. In other words, the output is a signal of high and low voltages with clean and vertical transitions.

Other optical sensors do not include such conditioning circuits. This means that the output of these sensors can be a little difficult to be interpreted by a TTL/CMOS input. There are several reasons for the difficulty.

10.5.1 Problems with a plain phototransistor

What this section describe are problems that is common to all phototransistors.

Transition edge

The output of a phototransistor (in a transmissive sensor) does not have sharp vertical edges when the sensor transitions (from light to dark or vice versa). This is because of physical properties of a transmissive sensor.

First, the IrED (infrared emitting diode) is not a point source of infrared. Instead, from the perspective of the phototransistor, it looks more like a disc of infrared. This means that during the transition from opaque to clear (from the encoder disc), a phototransistor experiences a continuous change from full IrED disc to no IrED disc. This continuous change of IrED disc exposure translates to a continuous change of the output voltage.

For a normal CMOS/TTL input, this is a problem. This is a problem because the amount of time to transition from full exposure to the IrED disc to no exposure is longer than the response time of a CMOS/TTL gate. Most CMOS/TTL input have specifications similar to the following:

- High-level minimum input voltage: $V_{IH} = 3.15V$
- Low-level maximum input voltage: $V_{IL} = 1.35V$

This means that the device guarantees to register a one when the input voltage is 3.15V or more, and guarantees to register a zero when the input voltage is 1.35V or less. But what if the input voltage is, say, 2.5V? Is that a one, or is that a zero?

Many CMOS devices bounce the interpretation between 0 and 1 when the input voltage is between the high-level minimum and the low-level maximum. This leads to false transition readings.

Rail to rail range

As described in the previous section, a TTL/CMOS device only guarantees to register a zero when the input voltage is below about 1.35V, and only guarantees to register a one when the input voltage is above 3.15V. Unfortunately, some transmissive sensors cannot produce an output voltage range that exceeds this range of 1.35V and 3.15V.

The problem is usually due to two physical properties of an optical encoder. First, some optical encoders have “weak” phototransistors. This means the phototransistors cannot let much current through the device, even when it is 100% on. As a result, it is often necessary to use a large value pull-up resistor. This, in return, change the response time of the phototransistor. As a compromise, sometimes it is necessary to use a less-than-optimal resistor value, making it impossible for a phototransistor to yield a low voltage when it is fully turned on.

The other limiting factor, surprisingly, is the encoder disc. Industrial encoder discs are machined from thin aluminum sheets. A thin foil of aluminum is 100% opaque to infrared. However, it is far more cost effective to create encoder discs from transparency sheets (for projector use). A stripe pattern can be easily generated by various programs, and the pattern can be printed to a transparency sheet using a laser printer.

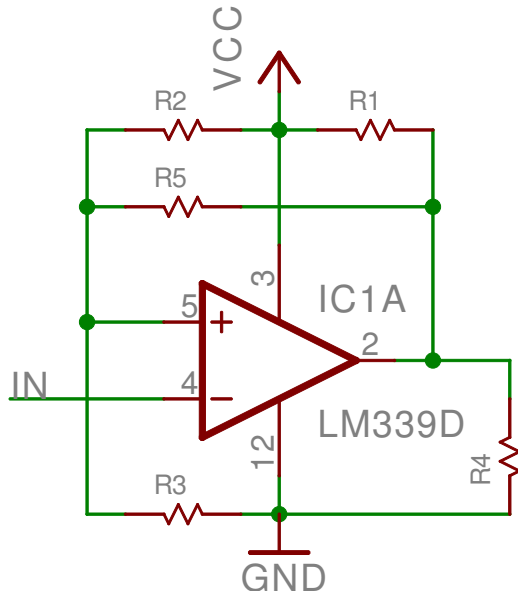
Although the color black from a laser printer is dark enough for the human eye (in the visible spectrum), it is not guaranteed to be opaque enough in the infrared spectrum. This is because infrared has a longer wavelength than visible light. As a result, a toner density that is sufficient to block the transmission of visible light is not necessarily sufficient to block the transmission of infrared.

As a result, a cheap homemade encoder disc may be somewhat transmissive even at the opaque (black) area. This infrared leak through means that the phototransistor is always on a little bit, and it is never completely off. The corresponding output of the phototransistor never reaches the high end of the rail.

10.5.2 Solution

Comparators may be used, combined with a variable voltage divider bridge, to generate the correct output. The LMV339 (a low voltage variant of the LM339) is one option. It is inexpensive at less than \$1 each (at single quantity). The datasheet from National Semi detailed how to add hysteresis. A single chip is enough for two quadrature encoders, although we will need a number of resistors (3 per comparator, yielding a total of 12 for the whole set up).

The basic circuit is illustrated as follows:



Here's how it works. When the output of the comparator is high, the voltage at the non-inverting terminal is as follows:

$$V_h = VCC \frac{R3}{R3 + \frac{1}{\frac{1}{R2} + \frac{1}{R5}}} \quad (10.1)$$

When the output of the comparator is low, the voltage at the non-inverting terminal is as follows:

$$V_l = VCC \frac{\frac{1}{\frac{1}{R3} + \frac{1}{R5}}}{R2 + \frac{1}{\frac{1}{R3} + \frac{1}{R5}}} \quad (10.2)$$

This means the input voltage must drop below V_l for the output to switch to high, and the input voltage must rise above V_h for the output to switch to low. This logic appears to be inverted, hence the name “inverted”.

Note that the “pull-up” resistor, R1, is necessary because the comparator has an open-collector type output. Without a pull-up resistor, the device cannot drive the output high (pin 2 of IC1A). R4, the load resistor, is not actually there. It is merely a symbol that denotes the impedance of the load (in our case, it is in the range of mega Ohms).

10.5.3 Example

Let's say the voltage swing (of the phototransistor output) is between 1.8V and 4V. We want to make the switch-over point centered at 2.9V with 0.5V of hysteresis. This means that we want the comparator to output high when the voltage drops below 2.4V, and output low when the voltage exceeds 3.4V. There should be no change when the voltage is between 2.4V and 3.4V.

This can be achieved by 130k for R2, 200k for R3 and 330k for R5. Once one of the three resistor values is fixed (R5, for example), the other two can be solved in closed form. It is the ratio among the resistor values that is important.

Chapter 11

PID Loop

11.1 The Theory

The drive system of a robot is control system. In order for this system to behave predictably, it should be a *closed-loop* control system. This means some feedback should be used to adjust the output to the motors. This section discusses the theory of a well-known method of closed loop control: PID (proportional integral and differential).

For any control, there is a set point. The set point is what is the feedback *should* read. Let us use $s(t)$ to represent the set point (reference) at time t . The actual feedback, $f(t)$, however, is very unlikely to be at $s(t)$. As a result, there is an error term, which is simply the difference of the two, $e(t) = s(t) - f(t)$.

Based on the error term, a control system adjusts the output to the system.

In continuous terms, the output of a control system, $k(t)$ is expressed as follows:

$$k(t) = K_p e(t) + K_i \int_0^t e(x) dx + K_d \frac{de(t)}{dt} \quad (11.1)$$

11.1.1 The Proportional Term

The proportional term is merely a scaled error $e(t)$. This term makes perfect sense. When $e(t)$ is large, it means our set point is far away from the feedback. As a result, the output needs to be “very high” in order to bring the feedback closer to the set point. As our feedback gets closer, $e(t)$ becomes smaller, and it makes sense to reduce the output so we don’t overshoot.

In a lossless system, just relying on $K_p e(t)$ should get us to the set point, eventually. However, because all practical systems are not lossless, $K_p e(t)$ never gets us to the set point. This is because at some point, the term $K_p e(t)$ is merely enough to counter the loss, and the system enters a steady state in which the input (from the heater or motor) cancels the output (heat loss or friction).

Despite the fact that $K_p e(t)$ is not sufficient by itself, it is the main component during the first phase of approaching the set point. Physically, $K_p e(t)$ is the bulk of the response to changes.

11.1.2 The Integration Term

The integration term, $K_i \int_0^t e(x) dx$ is the area between the set point and the feedback in a plot against time. As the proportional term $K_p e(t)$ “runs out of gas” as the feedback gets closer to the set point, the integration term $K_i \int_0^t e(x) dx$ continues to increase. As a result, the summation of $K_p e(t) + K_i \int_0^t e(x) dx$ should eventually become large enough so that any loss of the system is balanced by the contribution of $K_i \int_0^t e(x) dx$.

Note that the integration term is not entirely independent from the proportional term. As the integration term helps to bring feedback closer to the set point, the proportional term gets even smaller. Eventually, in a tuned system, as the feedback becomes the same as the set point, the entire output comes from the integration term.

Physically, $K_i \int_0^t e(x) dx$ is the amount of output necessary to balance loss of the system. This term is very dependent on how the other two terms are set up, although it will, eventually, get the feedback to match the set point.

11.1.3 The Derivative Term

The derivative term, $K_d \frac{de(t)}{dt}$ only contributes when there is an abrupt change of $e(t)$. This change can be due to a change of set point, or it can be due to a change of feedback. A sudden change of feedback can be due to external factors such as a bump on the road.

This term is also called the dampening term because it dampens the effect of the other two terms. As $e(t)$ gets smaller, $K_d \frac{de(t)}{dt}$ is negative. In fact, for motor control, $K_d \frac{de(t)}{dt}$ is very negative to begin with because the response of a motor is stronger at lower speeds.

So why do we need the derivative term?

One use of the derivative term is so that we can crank up K_p . With a dampening term, we can adjust the other two terms so that the output does not “fizzle” out as the feedback gets closer to the set point. Note that $K_p e(t)$ fizzles out by itself anyway, but $K_i \int_0^t e(x) dx$ contributes more in time. Also, the response of a motor decreases as speed increases.

11.2 From Continuous to Discrete

In a discrete system, we can transform the previous equation to a discrete form (t is a natural number):

$$k(t) = K_p e(t) + K_d (e(t) - e(t - n)) + K_i \sum e(t) \quad (11.2)$$

Note that the division of n for the differential term is absorbed by K_d .

Why choose n instead of 1? To get a differential, we just need to compare the error of time t to the error at time $t - 1$. This issue will be addressed later.

It should be noted that the sum term should be reset every time the reference is changed. This is because the integration of errors should be particular to a particular reference.

11.3 Practical Concerns

The main equation is simple, now comes the gritty details.

11.3.1 Resolution of Output

First, there is the resolution of the output. Since most robots are controlled by pulse-width-modulation (PWM), the output is an adjustment of the duty cycle of the PWM signal. The real question is, then, the resolution of PWM.

Obviously, a resolution of 2 (1 bit) is not sufficient. That would be the same as on-off control. On the other hand, a resolution of 1024 (10 bits) is too much because most mechanical systems have errors that exceed 0.1%. For a mechanical system that has an inherent error of 1%, 128 divisions for the PWM is enough. We will use r_k to represent the number of bits to represent the output.

11.3.2 The Resolution of $e(t)$ and K_p

The proportional term is the main component of the output of a PID loop. As a result, K_p is an important coefficient, as is the error term itself.

We need to determine the resolution of the error term, $e(t)$. Let us consider a system where the speed can be up to x ticks/s. This means the error term can be up to $2x$ because the motor can be spinning full speed backward and ask to spin full speed forward.

The next factor is how often to run the PID loop logic. The frequency of the loop f helps to determine the resolution of the error term. Although the actual error can be up to $2x$ ticks/s, between each PID invocation, the difference only be up to $\frac{2x}{f}$. For example, if the top speed of a reversible system is 1200 ticks/s, and the PID loop is invoked at 50Hz, $e(t)$ can only range from -48 to 48, which can be represented by a 7-bit signed integer.

The number of bit required for $e(t)$ is, then,

$$r_e = \frac{\log \frac{4x}{f}}{\log 2} \quad (11.3)$$

When r_e is close to r_k , the magnitude of K_p is small. This means you need to rely on fractions in K_p to fine tune the terms. Instead of using floating point numbers, many small MCUs benefit from the use of more efficient integer/fixed number computations.

For example, even if we know that K_p does not exceed 4, we can use 8 bits to represent it. The least significant 6 bits become fractional terms. Given an integral bit pattern of 10110101_2 , the actual represented value is $\frac{10110101_2}{2^6}$. The division by 2^6 is inexpensively performed by right shift operations.

11.3.3 The Resolution of the Integration Term

The integration term is important because it allows a system to reach the reference. Without the integration term, a PID loop can never achieve the reference signal.

The integral term relies on the summation of errors. This sum can become much larger than the error term itself. It is advisable to put limits on this term (caps). The range of the integral term should be about 20 times that of the range of the error term. Due to the magnitude of the error term, there is no problem with resolution in the integration term.

Because the integration is a large number to start with, the coefficient K_i must be a small fraction. This is easily done by assuming the binary point to the left of the most significant bit of the binary integer representing K_i . The trouble, however, is that we now have a large number multiplied by another large number. This can take a bit of time for an 8-bit system. As a result, some control systems choose to ignore some of the less significant digits to speed up computation.

11.3.4 The Resolution of the Differential Term

The differential term is the *change* of error. While this term can have the same magnitude as the error term itself, most of the time it is very small (one tenth the magnitude of the error term itself or smaller). In a continuous system, this is not a problem because there is *infinite* precision. However, in a digital system, this poses the problem of digitizing the contribution from the differential term.

Instead of just relying on the number of ticks between PID invocations, we can also rely in the period between ticks. Although period is proportional to the inverse of speed, it is still more accurate compared to the number of ticks at slow speeds.

Consider a realistic case where a system expects about 1000 ticks per second. This translated to one tick of millisecond. However, as the system slows down to rely on friction to come to a complete stop, the speed can be down to a small number (such as one) of ticks per second. It becomes impossible to know the actual speed. Using a timer, however, we can track the period between ticks and use the period to deduce the actual speed.

If we consider the worst case of a 16 count timer for a 50Hz PID invocation frequency, the lowest frequency is 50Hz, while the next lowest frequency (due to digitization) is $\frac{1}{15/16*20\text{ms}}$, which translates to 53.3Hz. This resolution is better than 7%, even when the tick count per time method fails. While 7% may not sound like a lot, it makes a lot of difference when there is nothing else to rely on.

11.4 Improving Feedback Resolution

11.4.1 The Problem

The feedback and set point are both in units of “ticks per PID period” up to this point. This is okay for most cases, assuming the encoding has sufficient resolution. However, for robots with low resolution encoders, and also for low-speed situations, just counting ticks between PID invocations produces coarse and quantized results.

Let us assume the PID control loop is invoked at 20Hz (20 times per second). Let us also assume the encoder disk has 1000 transitions per revolution. At a very low speed, such as 0.1 revolutions per second, we only get 100 ticks per second. This is divided by 20 because the sampling frequency is 20Hz. As a result, we only have, on the average, 5 ticks per period.

The error term is going to be even more quantized because it is the difference between the set point and the feedback. As a result, the control signal has very poor resolution because it depends on the error term.

11.4.2 A Solution

A solution is to compute the period between ticks, and use its reciprocal as the speed. This method is not suitable when there is sufficient encoding resolution because reciprocal is difficult to compute (compared to counting ticks), and the resolution of the period-oriented approach drops as speed increases. However, at lower speeds, this period-oriented method yields much better resolution than tick counting.

In our previous example, using the tick count method, 0.1r/s cannot be distinguished from 0.09r/s because both yields about 5 ticks per PID invocation. However, using the period measurement, 0.1r/s results in a period of 10ms, whereas 0.09r/s results in a period of 11.1ms. Even with a timing resolution of 1ms, we can still distinguish 0.09r/s from 0.1r/s.

Note that timing resolution can be improved by hardware peripherals. The ATmega128, for example, has an “input capture” device associated with timer 1. It can record the duration of a pulse at resolutions down to single clock periods.

11.4.3 Practical Considerations

Although we can improve the resolution of low speed encoding, the benefit may not be useful because of other problems. For example, due to the nature of DC motors, they are not smooth at lower speeds. As a result, it may not be practically useful to use period measurement to improve the encoding resolution at lower speeds.

Chapter 12

Motor Driver Circuit Design

12.1 TPIC0108 Interfacing

The TPIC0108 is the perfect chip to use because it supports all four modes. However, it does not have an enable input. This can be provided using a PAL/CPLD. The ATF750C-15SC (SO24 package) is a good option because it has enough pins, and cheap enough at \$2.14. The Needhams EMP-20 programmer can program this chip, provided that I get an SO-DIP convertor first (SO330-28p). Atmel provides WinCUPL that is free to create files for these devices.

12.1.1 TPIC0108 Logic for DC Motors

The logic to control the two lines of the TPIC0108 uses the following truth table:

| PWM | DIR | BRK | IN1 | IN2 | comments |
|-----|-----|-----|-----|-----|----------------------|
| 1 | ? | ? | 0 | 0 | quiescent hi-Z (off) |
| 0 | ? | 0 | 1 | 1 | brake |
| 0 | 1 | 1 | 0 | 1 | one way |
| 0 | 0 | 1 | 1 | 0 | the other way |

The equation for IN1 is as follows:

$$IN1 = \overline{\overline{PWM}} \times \overline{\overline{BRK}} + \overline{\overline{PWM}} \times \overline{DIR} \quad (12.1)$$

Whereas the equation for IN2 is as follows:

$$IN2 = \overline{\overline{PWM}} \times \overline{\overline{BRK}} + \overline{\overline{PWM}} \times DIR \quad (12.2)$$

This allows a PWM line be used for each channel. Note that the PWM

signal and the BRK signal are both active-low so that the default state can be pulled-up when the controller is in reset state.

This plan uses 3 input channels per H-bridge for maximum flexibility. Since this is often only useful for DC motors, we only need to use 6 lines from the MCU to control two DC motors.

12.1.2 TPIC0108 Logic for Stepper Motors

For stepper motors, we can use only two lines per motor for step and direction. This requires that the logic chip maintain its current state, and also to have an initial reset state.

The truth table of the stepper half-stepping state machine is as follows (coils A and B, each has its own IN1 and IN2 signals):

| A.IN1 | A.IN2 | B.IN1 | B.IN2 | comment |
|-------|-------|-------|-------|------------------------|
| 0 | 1 | 1 | 1 | A forward, B off |
| 0 | 1 | 0 | 1 | A forward, B forward |
| 1 | 1 | 0 | 1 | A off, B forward |
| 1 | 0 | 0 | 1 | A backward, B forward |
| 1 | 0 | 1 | 1 | A backward, B off |
| 1 | 0 | 1 | 0 | A backward, B backward |
| 1 | 1 | 1 | 0 | A off, B backward |
| 0 | 1 | 1 | 0 | A forward, B backward |

Because we need to use a “previous” state to determine the “next” state, the programmable logic needs to be registered. Fortunately, the ATF750C device has a product clock for each macro cell. This means we can dedicate a clock (STEP) for each stepper motor.

Next, we need to express the next state as an equation of the current terms, as well as the clock and direction. “ar” is async. reset state, “ck” is the clock product term, “oe” is output enable and “d” is the D-latch input term. The following is the equation for coil A IN1.

$$A.IN1.oe = 1 \quad (12.3)$$

$$A.IN1.ar = 0 \quad (12.4)$$

$$A.IN1.ck = STEP \quad (12.5)$$

$$A.IN1.d = DIR \cdot \overline{A.IN1} \cdot A.IN2 \cdot \overline{B.IN1} \cdot B.IN2 + \quad (12.6)$$

$$\overline{DIR} \cdot \overline{A.IN1} \cdot A.IN2 \cdot B.IN1 \cdot \overline{B.IN2} + \quad (12.7)$$

$$A.IN1 \cdot \overline{A.IN2} \cdot + \quad (12.8)$$

$$DIR \cdot A.IN1 \cdot A.IN2 \cdot \overline{B.IN1} \cdot B.IN2 + \quad (12.9)$$

$$\overline{DIR} \cdot A.IN1 \cdot A.IN2 \cdot B.IN1 \cdot \overline{B.IN2} \quad (12.10)$$

The other output pins can be determined similarly. Using this technique, we only need two lines to control two stepper motors.

12.1.3 TPIC0108 Chopper Drive Logic

To perform software controlled chopper drive, it is necessary to use a PWM signal to enable and disable the device. Each coil requires its own PWM signal.

We can build on top of the equations in the previous section. The trick is to utilize the output enable line and a pull-down resistor for each output. This way, we can connect the PWM (output compare) signal from the MCU directly to the output enable signal. When a pin is not driving, the signal defaults to 0, and the corresponding H-bridge goes to Hi-Z state.

The modification is as follows:

$$A.IN1.oe = A.PWM \quad (12.11)$$

$$A.IN2.oe = A.PWM \quad (12.12)$$

This approach requires the use of four output compare channels. In addition, the duty cycle needs to be controlled dynamically (the overflow interrupt needs to change the duty cycle).

12.2 Layout Considerations

The TPIC0108 is a thermal-enhanced SO20 IC. Although it has the same footprint as other SO20 (small-outline 20 pin) ICs, it has an extra heatsink area that is under the IC. This pad allows the chip to be soldered on a large area of pad on a PCB, and provide heatsinking capacity so the chip can be fully utilized.

With a reflow oven, this does not pose any problem because the solder paste placed under the chip will flow as the temperature increases. However, for hand soldering, it is much more difficult to get solder *under* a chip to flow.

This is why I included four large vias under the chip. A fine soldering tip should fit in such a via, and so we can solder the chip at least to the vias. This design also allows me to put a large pad on the top side for additional heatsinking.

As far as the traces are concerned, the current (20040322) layout has two main “trunks” for ground and supply. The design is also chainable, making it easy to add three more H-bridges without adding to ground and power routing problems. The relatively large ground and supply traces also provide additional heatsinking resources.

Because of the use of a PAL/CPLD, routing control signals is easier (than without a PAL/CPLD). As long as the PAL/CPLD is placed between the MCU and the H-bridges, I can use APL/CPLD equations as routing resources.

Part V

Reading Sensors

Chapter 13

Analog to Digital Conversion

Before we discuss any sensor topic, it is important to first discuss the analog-to-digital converter (ADC). On the ATmega128, there are 8 channels of ADCs, each channel has a resolution of 10 bits.

Due to the flexibility of the ADC component, we cannot cover all possible configurations in this chapter. However, we can cover enough so that we can make this discussion directly applicable in following chapters.

13.1 Configuration

13.2 The ADC clock

The *successive approximation* nature of the ADC (used in the ATmega128) requires its own clock. Instead of using another oscillator that is asynchronous to the system clock, the ATmega128 allows software to configure the ADC clock based on the main clock. This is documented on page 232 of the ATmega128 datasheet.

The prescaler is controlled by ADPS0-ADPS2, bits 0 to 2 of I/O location ADCSRA. The exact division factor is shown on page 244 of the datasheet. Note that a high speed ADC clock cannot fully utilize the 10-bit resolution capable of the device.

13.2.1 Voltage Reference

An ADC only computes the *ratio* of some input voltage with respect to a supplied voltage. The ATmega128 ADC has a programmable voltage reference selection. Bits REFS0 and REFS1 (bit 0 and 1 of ADMUX) selects the source of voltage reference. Refer to page 242 of the datasheet.

For most practical use, one can either use AVCC (analog supply voltage) or the internal 2.56V reference (modes 1 and 3, respectively). Note that both modes require that an external capacitor be connected between the pin AREF and ground.

13.2.2 Multiplexer Selection

Although there is really only one ADC in the MCU, and only 8 external pins for analog input purposes, the ADC peripheral has 32 possible channels to select from. This is because the ATMega128 can operate in differential mode and have internal amplifiers to amplify the voltage. The channel is selected by MUX0 to MUX4 (bits 0 to 4 of ADMUX). Refer to table 98 on page 242 for a full list of the channels.

13.2.3 Free Running versus Manual

The ADC can be set to either free running mode or manual mode.

In free running mode, the ADC samples and updates automatically and continuously. You can get the highest sampling frequency using this mode. Note that the ADC must be fixed to a channel in free running mode. In other words, you cannot set up the ADC to “scan” channels in the free running mode.

In manual mode, the ADC only samples and updates after it is “started” manually in software. After one conversion, the device stops. This mode is useful if you need the best resolution (you need to put the rest of the MCU to sleep during conversion), or if you need to reduce power consumption.

Free Running mode is selected when ADFR is set to 1. ADFR is bit 5 of ADCSRA.

13.2.4 Interrupts

Because it takes a significant amount of time for a conversion to complete, most programs do not poll for completion. Instead, most programs enable ADC interrupt so that the corresponding ISR is called only after a conversion is completed.

Bit ADIE (bit 3 of ADCSRA) controls if interrupts should occur after each conversion. ADIF (bit 4 of ADCSRA) is set when an ADC conversion completes. It is cleared either by executing the ISR of ADC completion, or when a 1 is written to this bit.

This flag is useful when AD conversion is handled by polling.

13.2.5 Enabling and Starting

ADEN (bit 7 of ADCSRA) controls whether the ADC is enabled or not. A one in this bit means the device is enabled. ADSC (bit 6 of ADCSRA), on the other hand, starts an AD conversion. A one should be written to this bit to start the next conversion.

13.2.6 Bit Positions

The ADC also allows you control where the significant digits are located. Bit ADLAR (bit 5 of ADMUX) selects whether zeros are inserted to the most significant bits, or the least significant bits.

If ADLAR is 0, then the most significant bits are zeros, whereas bit 0 to bit 9 are the results. If ADLAR is 1, then the least significant bits are zeros, whereas bit 6 to bit 15 are the significant bits.

13.2.7 Reading Results

After each conversion, the result is stored in ADCL and ADCH. ADCL is the least significant byte, whereas ADCH is the most significant byte. In order to keep both bytes synchronized, ADCL should be read first, then ADCH. This is because reading ADCL latches the value of ADCH into an 8-bit buffer, then reading ADCH afterwards ensures both bytes belong to the same sample.

13.3 Common Techniques

This section discusses common techniques that can be used with ADCs. Most of these techniques are not specific to the AVR MCUs.

13.3.1 One-shot ADC Read

Some embedded applications do not need to read the ADC continuously. For example, in a room temperature control application, you only need to read temperature every 30 seconds or so.

In these applications, you can use the following approach:

```
set up ADC
start ADC
while ADC is not done do
    nothing to do here
end while
```

This approach is only suitable if an application has no time critical operations when it is waiting for the ADC to finish. The advantage of this approach is that it is easy and universal. On the other hand, this approach makes the program wait for ADC completion.

13.3.2 Free-running Background Update

For ADCs that has a free-running mode and interrupt capability, you can set up the ADC to free-running and interrupt when a conversion completes. In the ISR, do the following:

```
copy from ADC value I/O location to global variable
```

Then, you write a function to read the global variable. You'll need a function to do this because interrupt should be disabled when you take a snapshot of the global variable:

```
unsigned ADC_getsnapshot(void)
{
    unsigned result;
    char s = SREG;
    _CLI();
    result = ADC_value;
    SREG = s;
    return result
}
```

Then, whenever you need to read the current ADC value, you just need to call this function:

```
if (ADC_getsnapshot() < 512)
    // turn on the heater
    ...
```

This method is better than the previous one because there is very little overhead to read the current ADC value. A program does not need to stop and wait for a conversion. However, one drawback of this approach is that the value can be “stale”. In other words, the snapshot can be a value that is t old, whereas t is the period of free-running conversion.

With the AVR, you don't even need to set up an interrupt because the value register of the ADC is already buffered. In other words, the function to return the current ADC value can be as simple as follows:

```
unsigned ADC_value(void)
{
    unsigned result;

    result = ADCL;
    result |= (ADCH << 8);
    return result
}
```

What if we need to read values of different ADC channels?

13.3.3 Server-client Approach

To get the latest value of different ADC channels without busy waiting, we need to set up a server-client architecture.

Although this may sound difficult, it is not. Let us assume that we are using a multithreading kernel. When a thread needs to read the value of a channel, it calls a function that has the following logic:

```
queue the request and current thread
```

```

if ADC is inactive then
  activate (set up the ADC)
else
  nothing to do
end if
block the current thread
  The ADC ISR has the following logic:
locate the filled request (head of queue)
copy ADC value to a return value for the request
remove the request from the queue
if there is more in the queue then
  set up the ADC for the requested channel
end if

```

This approach is only useful for multithreading programs because when one thread is waiting for an ADC conversion, other threads can continue. The actual code to do this is a little messy because a multithreading kernel must be involved.

Instead of using a dedicated queue for ADC requests, it is also possible to use a semaphore for queuing purposes. When a thread needs to start an ADC conversion, it can call a function that does the following:

```

P(adc semaphore)
set up ADC
P(adc completion)
copy the ADC value to return value
V(adc semaphore)

```

The ISR is then modified to do the following:

```

V(adc completion)

```

My real-time kernel supports both multithreading and semaphores, so it can easily implement this approach.

Chapter 14

IR Ranging Sensor

This chapter discuss the Sharp GP2D12 sensor.

14.1 General Information

The Sharp GP2D12 sensor (and its siblings) are triangulation-based sensors. In other words, a beam of infrared is emitted, and the reflection is sensed by a linear CCD (charged-couple device). Objects at different distances project reflections at different positions of the linear sensor. This technique is similar to the focusing mechanism of point-and-shoot cameras and older range-finder cameras.

While this technique has excellent near-range distance resolution, the performance (accuracy) degrades rapidly as the distance increases. This is because distant objects yield very small reflection angular differences, making them more difficult to distinguish using a digitized triangulation device.

You can purchase the GP2D12 sensor from many distributors, <http://www.digikey.com> is one of them. Note that you need to order a relatively large number of units from these distributors to get a good price. At 25 quantities, Digikey wants about \$8.50 for each sensor. This is a reasonable price.

For more information about the sensor, refer to <http://rocky.digikey.com/WebLib/Sharp/Web%20Data/gp2d12.pdf>.

14.2 Interface

The GP2D12 sensor outputs an analog signal that maps to the location of the CCD device detecting the IR reflection. This means that, in theory, this means that the voltage you measure is not the distance.

You can get the distance using an equation. Refer to <http://www.acroname.com/robotics/info/articles/irlinear/irlinear.html> for a discussion of how to calibrate and compute the distance sensed by the sensor.

To use an GP2D12 sensor, a microcontroller should have an analog-to-digital converter. We discussed this topic in the previous chapter. The specification of

GP2D12 also indicates that the voltage range of the output ranges from 0V to about 2.5V. This is perfect, because we can set up the ATmega128 to use its internal reference voltage of 2.56V.

14.3 Important Notes

The GP2D12 sensor is easy to work with. It only requires 5V, ground, and outputs an analog voltage that can be converted to distance.

However, you must understand some of the weaknesses of this sensor to utilize it optimally. Otherwise, you will be disappointed because it is not likely to live up to your expectations.

14.3.1 Stabilizing the Supply Voltage

The sensor draws relatively big amount of current in pulses when it operates. The *average* is only about 33mA to 50mA (from the specifications). This is not much. However, the instantaneous current consumption can reach 200mA (according to some unofficial experiments) or even more.

This instantaneous current will upset most voltage regulators if a design does not have provisions for it. Some controller will even reset because this much current will make the output of the regulator drop below its threshold to reset. Other controllers will have a ripple superimposed on VCC, which can then affect the effective resolution and accuracy of the ADC.

To minimize the effect of this large current, the easiest solution is to put a large capacitor across the VCC and GND pins of the sensor. You need to put the capacitor as close to the sensor as possible to reduce the side effects. The capacitor value can range from 20 μ F to 200 μ F. It is also recommended that you use low-ESR (low equivalent series resistance) capacitors that are capable of large ripple current.

14.3.2 Debouncing

Due to an internal design flaw, the GP2D12 sensor can, occasionally, give you spurious values even when the distance to object remains the same. This problem can compound on the current consumption problem. However, even with the power consumption problem fixed (using capacitors), the internal design flaw still make the sensor output spurious readings that are incorrect.

You can “fix” this problem by software debouncing. First, you need to experiment can find out the range of these spurious errors. The magnitude of this error is the same across the entire range. Once you find out, in terms of counts of the ADC value, then you can keep a circular queue of values, and throw away ones that are at least this much from its neighboring values.

In other words, you can use the following algorithm to filter the buffer (you need to wrap around correctly using the modulus operator, not shown in the following code):

```

t ← 0
while t < n do
  if (|vt-1 - vt| ≥ ethres) ∧ (|vt+1 - vt| - vt ≥ ethres) then
    vt ←  $\frac{v_{t+1} + v_{t-1}}{2}$ 
  end if
  t ← t + 1
end while

```

14.3.3 Sampling Frequency

The GP2D12 is not a continuous analog sensor. Instead, it measures every 38ms or so, but maintains the output of the previous measurement continuously. This means that if you measure more frequently than 38ms, every two values will be “identical”.

Does this mean that you should not sample more frequently than every 38ms? To facilitate debouncing, you should read at least twice for each measurement. That means you probably want a sampling frequency of 80Hz or so. Although the *average* sampling period is 38ms, it can be as short as 28ms. Given that you want to read at least twice for each reading, you need to read every 14ms. A sampling frequency of 80Hz translates to 12.5ms between samples.

Does it help to read at a higher frequency, like 200Hz? In terms of processor overhead, reading at 200Hz is still very easy. At a higher sampling rate, you can “average” out and get a more accurate result from within the same measurement. This is because the output of the device can fluctuate even for the same measurement.

However, there is also a noise between measurements. To remove the inter-measurement noise, increasing the sampling frequency will not help.

14.3.4 Inter-measurement Filtering

In order to get a higher resolution, especially for distant objects, you need to do inter-measurement averaging. This means you have to average the values measured over some multiples of 38ms.

In theory, assuming inter-measurement noise is uniformly distributed, if you average over about 80ms, the resolution is improved by one bit. If you average over about 160ms, the resolution is improved by two bits.

To determine the number of bits you need to “resuce” by averaging, you have to fix the distance of an object, then see how the ADC values fluctuate. Because the distance is fixed, you *should* be getting the same values all the time. However, if you observe a fluctuation of up to 8 counts even after debouncing, then you have an inter-measurement noise of 3 bits. To get you the best resolution, you need to average samples collected over 320ms (about 8 times 38ms).

In the previous example, averaging over 320ms will not help because you are already bound by the limitations of the ADC. To get more information out of the sensor, you’ll need to use the differential mode of the ADC, and provide

a programmable precise voltage reference source. In general, it is not worth doing.

Note that resolution improvement by averaging only works when the distance to object does not change. If you are tracking a moving object (which is often the case for mobile robots), you cannot afford to average over 320ms.

What you can do is to maintain a circular queue that is large enough for 30 to 40 samples. Depending on the distribution of the samples, you can determine what time frame to use for averaging. If all the samples are within 8 counts from each other, average over the entire circular queue. If the most recent 16 samples are very different from the rest, only average over the most recent 16 samples.

In other words, from the most recent sample, work your way back in time until either you are the entire circular queue, or up to a point where the difference is more than x counts from the most recent sample. This automatically trades responsiveness for accuracy.

14.3.5 Distance Noise Prediction

All robots should know how much it can count on the sensors. In other words, if you already know that the noise level is up to 2 counts, you need to translate that to distance noise so that the robot knows that the actual distance (to an object) is between x and y , although the sensor indicates v .

Because the noise level is linear to the voltage (ADC value), you need to go through some computations to figure out distance noise. You can use the same equations for computing distance from voltage.

Part VI

Communication

Chapter 15

General Communication

While the virtue of a robot is autonomy, it is often necessary for a robot to communicate with another computer, robot or human operator. This chapter discusses the various reasons for communication, as well as common devices to do so.

15.1 Reasons

15.1.1 Program update

Most robots are now controlled by reprogrammable controllers. This means, in theory, the program running in a robot can be changed. This feature is very handy because it allows a development team refine the software of a robot without having to physically disassemble (most of) a robot.

15.1.2 Remote control

In the initial phase of developing a robot, the development team needs to debug low-level component, including power electronics and mechanical components. While a robot can be equipped with traditional remote control circuits and receivers, it can just as easily use more computer-oriented means of communication for remote control.

15.1.3 Debugging and Logging

Debugging a robotic application is not as easy as debugging a desktop application program. The main reason is that events are asynchronous to the execution of the program. The other reason is that we cannot mount a monitor or keyboard to the embedded controller running on a robot. Even if this is possible (some robots use laptop computers as their controllers), it is difficult to read or type when a robot is in motion.

As a result, developers rely on remote debugging techniques. This means a debugger runs on a “host” machine, which is most likely a notebook computer (for portability) or a desktop in some cases. The debugger then relies on some means to communicate with a bare-bone monitor program on the robot controller. This way, a programmer can debug a program that is actually running on a robot using a computer that may not be physically connected to the robot.

Although software-based debuggers are helpful, they are often not 100%. The reason is that when a program is being debugged using single stepping or stopped, the dynamics of the program changes. As a result, some “bugs” that are present when a program executes at full speed disappear when the program is being debugged.

To facilitate software bugs that are sensitive to real-time, sometimes developers rely on logging. Since logging events tend to have very little impact on the overall execution speed of a program, most real-time sensitive bugs remain undisturbed by logging. The information contained in a log entry is flexible, and is entirely up to the programmer to decide what needs to be logged.

It is possible for some controllers to keep the log on themselves until such time that the programmer wants to download it. However, on “thin” controllers, there may not be sufficient storage to log events. As a result, the log is transmitted, at real-time, to another computer that receives the information and writes everything to a file. This file can be viewed at real-time, or it can be examined manually or with the help of filter programs after the fact.

15.1.4 Environmental Interaction

In some environments, a robot may need to communicate with other devices. For example, a vacuum robot may want to communicate with occupancy sensors already installed in a house so it does not enter rooms that are currently in use.

In other environments, beacons can be set up at known locations. When a robot gets close enough to a beacon, the robot can communicate with the beacons to get the ID of the beacon, along with the current time (according to the beacon) and other data. At the same time, the robot can also upload information to the beacon. Such information may include states of the robot, the ID of the robot, images it has captured and etc.

The beacon approach is particularly useful in indoor environments where an existing wired network is already established. Not only does it help a robot confirm its location, it also helps to eliminate the need for a robot to have full time connectivity to the network.

15.1.5 Inter-robot Interaction

Robots can also communicate with each other. This is particularly important when robots need to coordinate with each other to accomplish a common goal. Robot soccer is a robot sport that requires the coordination of several robots in order to score.

In emergency handling, rescue robots also need to coordinate with each other to handle a crisis. Otherwise, a robot may block the exit of another robot, or robots may even collide by themselves.

15.2 Devices

15.2.1 RS-232, asynchronous

The EIA-232 standard specifies voltage levels for 1 and 0. It is the asynchronous protocol that specifies timing of the 1s and 0s for the transmission and reception of data. However, most people associate RS-232 with the asynchronous protocol. In this section, we will use this more popular definition of RS-232 communication.

The nice part of RS-232 is that it is widely available on notebook computers and desktop computers. Many controllers also have the hardware to handle RS-232 communication. Even with modern sub-notebook computers without any built-in RS-232 ports, they can be added inexpensively using USB-serial converters.

The maximum speed (bandwidth) of RS-232 is 115200bps, although some devices are capable of up to 250000bps. By today's standard, this bandwidth is not impressive. 250kbps translates to about 30kB/s because one byte takes 10 bits to transmit (including the start and top bits).

A major drawback of RS-232 is that it relies on the absolute voltage of signals. This can be easily disturbed because of a long cable, or interference from electro-magnetic devices (such as motors). As a result, RS-232 usually only has a recommended range of 12 feet or so.

RS-232 is also an end-to-end protocol. This means an RS-232 port on a notebook computer can only talk to one controller. While this may not appear to be a significant limitation for "usual" applications, it is somewhat limiting for robotics. It is not uncommon that a relatively complex robot have several controllers.

15.2.2 RS-485

The EIA-485 protocol specifies a *differential* signal instead of an *absolute* (RS-232). This difference makes it possible for RS-485 to have a much greater range (up to 1 mile) because it is not sensitive to ground differences. Using twisted wires, RS-485 is also insensitive to usual magnetic interference.

RS-485 specifies a half-duplex device, whereas RS-422 specifies two RS-485 channels for a full-duplex device. In reality, RS-485 is often sufficient because many communication protocols are half-duplex, hence not requiring the full-duplex capability of RS-422.

The bandwidth of RS-485 depends on many environmental factors. The length of the cable, for example, has a large impact on the available bandwidth.

For short distance (12 feet or so), RS-485 has the same bandwidth as RS-232. For up to a mile, RS-485 drops down to 2400bps to 4800bps.

One major drawback of RS-485 is that it is seldom available on computers. Some companies make expansion cards or converters, but they are usually quite expensive. RS-485 is suitable for lower-bandwidth communication within a large robot.

15.2.3 Ethernet

More and more often, embedded controllers are now equipped with ethernet connectors and drivers. As we all know, ethernet has a minimum bandwidth of 10Mbps.

Ethernet is extremely suitable for “dry-dock” development work due to the high speed. In fact, for embedded controllers that has operating systems, a developer can even telnet into the controller and debug programs on the controller itself. Because of the higher bandwidth, a developer can also quickly update the OS on an embedded controller over ethernet.

Unlike RS-485, which only requires two twisted wires, ethernet requires a thicker cable. This means that ethernet is not suitable when a robot is in motion.

15.2.4 Wifi

For embedded controllers with a PCMCIA, Compact Flash or PCI slot, we can use Wifi (wireless fidelity) for wireless communication. Even the older standard, IEEE 802.11b, has a maximum bandwidth of 11Mbps. Of course, to utilize a Wifi expansion card, we also need to run a fairly modern operating system with proper drivers.

Wifi offers many advantages over the previous means. For example, it is wireless. This means there is no tangling of wires even when the robot is in motion. Wifi is also omnidirectional. There is no need to point a sensor/transmitter to the robot. To extend the range of wifi, however, it is possible to design antennae that are highly directional to get a better range. But then it is necessary to point the antennae to the direction of the robot.

Wifi also has enough bandwidth for live on-target debugging and even development. This simplifies the set up of the tool chain. Most operating systems that can handle wifi also have full TCP/IP stacks, which means the robot can be its own web-server, database server and etc.

The main disadvantage of Wifi is that it requires a bit of resources. Typically, this is only useful if a controller is based on the mini-ITX, PC104 or other embedded PC form factor. Furthermore, it also requires a fairly modern operating system (such as Linux, WinCE and etc.) to support the device.

15.2.5 Bluetooth

Lately, there is a wireless solution for thin controllers that cannot utilize Wifi. Bluetooth is a relatively low-end wireless standard for “appliance” devices such as cell phones.

There are converters that converts RS-232 to/from bluetooth. These devices are not inexpensive, but they are easy to use. Bluetooth also consumes less power compared to Wifi.

Besides the lower bandwidth (57600bps) of bluetooth, it is also less secure than Wifi because there is no protocol level encryption. It is, typically, not a major problem for development work.

Chapter 16

Low-Level Asynchronous Communication

16.1 Timing Details

The adjective “asynchronous” means that the transmitter and receiver are not synchronized by a common clock signal. Instead, the transmitter and receiver *agrees* on a common bit rate for information transmission. Common bit rates are 1200bps, 2400bps, 4800bps, 9600bps, 14400bps, 19200bps, 28800bps, 38400bps, 57600bps and 115200bps.

While nothing is being transmitted, an asynchronous communication signal idles at a logical-high state. “Logical high state” means that from the CMOS/TTL point of view, the voltage is high (typically 3.3V or 5V).

When a byte is to be transmitted, the following happens:

- 1+ start bit
- up to 8 data bits
- an optional parity bit
- 1+ stop bit

Let us examine each component:

16.1.1 Start Bit(s)

Each word begins with at least one start bit. A start bit is a logical low signal for the duration of one bit. In other words, for 19200bps, the signal is low for $52\mu\text{s}$.

The main purpose of a start bit is to indicate the beginning of a byte. Without a start bit (that has the opposite polarity of the idle state), it is impossible to indicate when the transmission of a byte begins.

16.1.2 Data Bits

Data bits are bits to be received. A bit value of 1 is transmitted as logical high, while a bit value of 0 is transmitted as logical low. Most devices handle up to 8 data bits.

Within a byte, data bits are transmitted LSb (least significant bit) first.

16.1.3 Parity Bit

Older devices may require the use of parity bits. There are two options. An even parity bit is 0 if and only if there is an even number of 1s in the data bits, where as an odd parity bit is 0 if and only if there is an odd number of 1s in the data bits.

For example, the 8-bit value 0x56 has an even parity bit of 0 and an odd parity bit of 1.

Note that a parity bit is optional.

16.1.4 Stop Bit

A stop bit is driving the signal logical high for the duration of one bit. It marks the end of a byte (or smaller packet). The minimum number of stop bit(s) can vary from 1 to 2.

16.2 The USART

On the ATmega128, there are two USARTs. USART stands for “Universal Synchronous/Asynchronous Receiver Transmitter”. It is a hardware device on the MCU silicon die that can both transmit and receive information. For now, we’ll focus on the asynchronous part of this device.

Note that the ATmega128 is a fairly modern and complicated MCU. As such, the USART device is highly configurable, which also leads to the complexity of getting the correct bit values right.

There are two identical USARTs on the ATmega128. As a result, all I/O locations must be postfixed by a 0 or 1 to indicate which USART. In our notation, *n* indicates where either 0 or 1 should be specified.

16.2.1 Clock and Transmission Speed

- UMSEL in UCSRnC should be 0 to select asynchronous operation.
- U2X (bit 1) of UCSRnA selects between normal (0) or double speed (1) asynchronous operations. Double speed is not free, as it sacrifices some of the USART’s ability to resynchronize transmissions that are not accurate in receive mode, and it can also lead to outgoing (transmitter) error for recipients that are not sampling quickly enough.

- **UBRR** (**UBRRnL** and **UBRRnH**) needs to be initialized to the correct period (of a bit), refer to page 172 of the datasheet for an equation to compute the value of **UBRR** from the desired bit rate

16.2.2 Frame Format

The USART has the following flexibilities:

- start bit: this must be one bit.
- data bits: this can range from 5 to 8. This is controlled by **UCSZ** (bit 2 of **UCSRnB** as bit 2, bit 2 of **UCSRnC** as bit 1, and bit 1 of **UCSRnC** as bit 0). As a 3-bit number, it adds a value to 5, and the sum is the number of data bits. All value of 4 to 6 reserved. A value of 7 specifies 9 (yes, nine!) data bits.

The first 8 bits are specified by **UDRn** both for reception and transmission.

For transmission, bit 8 is specified by **TXB8** (bit 0 of **UCSRnB** . For reception, bit 8 is stored in **RXB8** (bit 1 of **UCSRB**).

- parity bits: this can be none (no bit used), even or odd It is controlled by **UPM** (bits 0 to 1) of **UCSRnB**.

Table 78 on page 190 of the data sheet explains the possible modes.

- stop bit(s): this can range from 1 to 2. It is controlled by bit **USBS** in **UCSRnC**.

16.2.3 The Ninth Bit and Networking

On page 185 of the datasheet, Atmel explains the multiprocessor communication mode. Here is a simplified version.

The **MPCM** (multi-processor communication mode) bit of **UCSRna** is used to enable (iff set to 1) this mode of operation. The **MPCM** is a useful feature in a multidrop network using the asynchronous protocol. Multidrop in this context means multiple network nodes all listen to the same physical signal.

In such a network, it is common that a networking protocol (above physical level, in the link level) uses a byte to identify the destination (recipient) address. This byte is normally followed by the “payload” of a message.

Without **MPCM**, a node must receive and interpret every byte transmitted on the network, even though a message may not be directed to this node. This is because if a node does not interpret the entire message, it does not know when the message ends, and as a result it does not know when the next message (which may be directed to this node) begins.

MPCM utilizes a ninth bit to alleviate the average processing load of nodes on a multidrop network. If and only if the ninth bit is a one, the transmitted byte (the first eight bits) is an address. As a result, if the ninth bit is a zero, the payload is some data byte of a message.

This scheme helps to reduce the average processing requirements. Each receiving node only needs to listen for bytes with a one for the ninth bit. If the address does not match the node's own address, then the following data bytes can be ignored. Otherwise, a node needs to switch mode and receive/interpret all following data bytes.

16.2.4 Errors

Page 181 describes the possible error flags set by the USART.

- FE (frame error): essentially, when an expected stop bit is a zero instead of a one, this bit is set. Note that it does not catch all possible cases.
- DOR (data overrun): this happens when the MCU cannot remove data from UDR fast enough. UDR can buffer one byte when the following is still begin received. However, when the following one is received, it overwrites UDR so the previous byte is lost if UDR is not read.
- UPE (parity error): when the parity bit of a byte does not match the payload, this error flag is set.

16.2.5 Interrupts

The USART is designed so that the processor only needs to pay attention when something happens. The sources of interrupts are as follows:

- Reception complete: this is enabled by RXCIE (bit 7) of UCSRnB. An interrupt is generated when RXCIE and RXC (bit 7 of UCSRnA) are both set. This is cleared by reading UDR.
- Transmission complete: this is enabled by TXCIE (bit 6) of UCSRnB. An interrupt is generated when TXCIE and TXC (bit 6 of UCSRnA) are both set. The interrupt request is cleared when the corresponding ISR executes. When TXC is set, it means the USART is done with transmitting everything asked of it (and that the UDR for the transmitter is empty). This interrupt is generally useful to indicate that *a packet is fully transmitted*. The corresponding ISR typically needs to turn around the bus, or at least turn off the transmitter hardware in a multidrop network.
- UDR empty: this is enabled by UDRIE (bit 5) of UCSRnB. An interrupt is generated when UDRIE and UCRE (bit 5 of UCSRnA) are both set. When UCRE is set, it only means that the transmitter is ready to accept another byte in UDR. It is possible that the previous byte is still *being transmitted out of the USART*. This interrupt is generally useful to keep filling the UDR so that the transmission of data is continuous and uninterrupted. The corresponding ISR typically gets the next byte of the current message to transmit. When there is nothing else to transmit, UDRIE should be cleared because there is no way to clear UDRE.

16.2.6 Enabling and Disabling

RXEN (bit 4) and TXEN (bit 3) of UCSRnB should be set if and only if the receiver and transmitter needs to be enabled. When the receiver or transmitter are disabled, the corresponding pins return to act like general purpose I/O pins. Note that the receiver and transmitter can be enabled independently from each other.

It should be noted that when the transmitter is disabled, it still clocks out any remaining bits not yet transmitted. This means it is safe to disable the transmitter as soon as the last byte of a message is written to UDR.

In contrast, disabling the receiver stops reception instantaneously. The value of UDR (for reading) as well as the error flags become invalid as soon as a receiver is disabled.

16.3 Circular Queues as Software Buffers

Most UARTs or USARTs on inexpensive MCUs only has a one byte buffer. This means that it can only store one byte while another byte is arriving. As soon as the other (new) byte is received, the buffered byte is overwritten. For output purposes, an inexpensive UART has a buffer to hold a byte, while the “shift register” shifts out another byte.

Without any additional buffering, this means that an application must be able to read and process a received byte within the transmission time of a byte. This may sound very reasonable. After all, even at 57600bps, the actual transmission rate is about 5kB/s, or 200 μ s between each byte.

In reality, there may be important matters for the processor to attend to, rather than interpreting a received byte. For example, in a robot, it is more important to avoid collision with an object than to reply to a query from an end user.

As a result, it is common to separate the “read” operation from the “process/interpret” operation. We want to read a byte from the UART before it is overwritten by the following byte, but we may want to delay the processing of this byte until we are cleared from a collision threat.

In order to do this, we need to implement a buffer in software to store the received-but-not-processed bytes. This section describes the concept of a circular queue (in pseudocode).

16.3.1 An Example: Circular Queue

A “queue” is a British term for a line of persons waiting for something. It is also the *official* term for any data structure that has the first-in-first-out characteristics.

Let us consider the example of a carousel in-slot that is responsible for taking all incoming work to be done at a worker’s desk. For our discussion, let us assume the carousel has 8 slots to handle up to 16 buffered incoming work requests.

The worker uses a post-it to indicate the “next to process”. As the worker removes an incoming request from a slot, the post-it is moved clockwise by one slot. There is also a post-it to indicate “next request”. Anyone who wishes to place a work order must put the work request at the “next request” post-it, then move this post-it clockwise by one slot.

Figure 16.1 illustrates an initial configuration where there are two buffered requests (to be processed). Note how the “next to process” post-it is tagging the first filled (gray) slot clockwise, and how the “next request” post-it is tagging the first unfilled (white) slot clockwise.

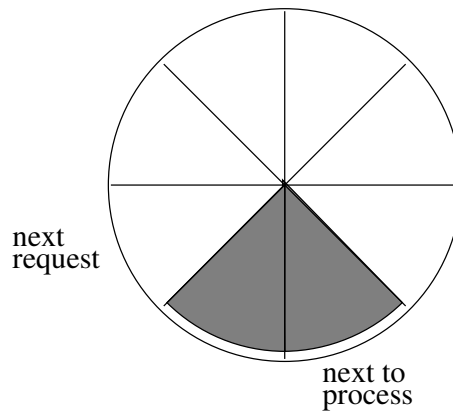


Figure 16.1: Initial configuration with two requests.

After we process a request, we move the post-it to the next request (clockwise). Figure 16.2 illustrates the carousel after one request is processed.

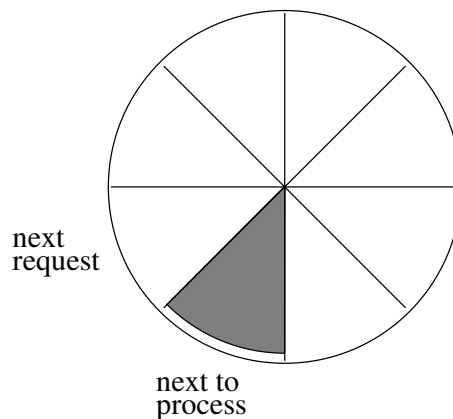


Figure 16.2: One request is removed and processed.

If the worker process the only remaining request before any new requests come in, it becomes the one illustrated in figure 16.3. Note that in this configuration, both the “next to process” and “next request” post-its are on the same slot. This is how the worker knows there is no request to be processed at this point.

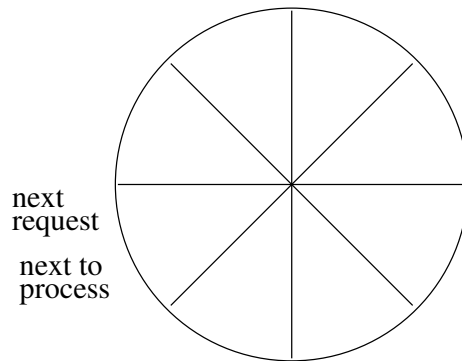


Figure 16.3: No request is remaining.

Let’s say lunch hour is over, and suddenly many requests come in at about the same time. More specifically, let us assume eight requests are received at about the same time before the worker can remove any one for processing. Our carousel is now filled. The “next request” is moved clockwise eight times. Figure 16.4 illustrates our 100% filled carousel.

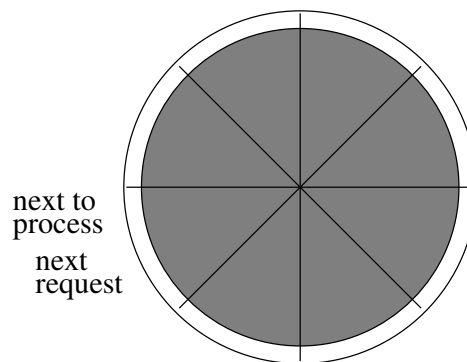


Figure 16.4: Eight requests are received.

Figure 16.4 also demonstrates how we know there is no more room for one more request. When the “next request” post-it is at the same slot as the “next to process” post-it, our carousel is filled up.

This also means it is not sufficient to just keep two post-its. We need to actually count the number of filled slots.

16.3.2 Where is the Array?

The carousel is our array! Imagine that we assign indices to each slot in the carousel. Figure 16.5 illustrates the numbering of the slots.

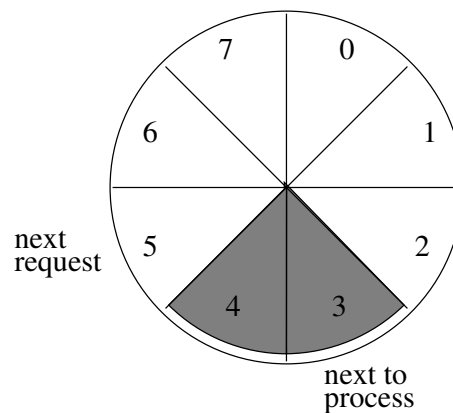


Figure 16.5: Carousel as an array, numbered slots.

This means instead of using a post-it to tag a slot, we can use two indices (integers) to indicate which slot is to be processed, and which slot should receive the next request. That is, “next to process” and “next request” become two variables of integer type.

16.3.3 The Record Structure

In order to represent a carousel, we can use the following record representation, after we rename “carousel” to its formal name, “circular queue”, “next request” to tail and use count to count the number of filled slots:

```
record circular queue
  a : array of 8 slot
  head : integer
  count : integer
end record
```

Note that we did not say specifically what a “slot” is. In other words, “slot” can be a simple type like integer, or it can be a complex record or array all by itself. As far as the logic of a circular queue is concerned, the type of a slot is not important.

16.3.4 Logic for a Circular Queue

You can perform several operations for a circular queue. For example, we can initialize it, check if it is empty, if it is full, to remove the item at the “head” and to insert an item at the “tail”.

We will write some subroutines to handle each operation.

Initialize an Empty Circular Queue

The following subroutine initializes a circular queue c .

```
define subroutine initialize circular queue
given and provide  $c$  : circular queue
 $c$ .head  $\leftarrow$  0
 $c$ .count  $\leftarrow$  0
```

Check for Empty

As discussed earlier, checking for an empty circular queue only needs to check if the head and tail are the same. The following is the subroutine to do so:

```
define subroutine is circular queue empty
given  $c$  : circular queue
provide  $isempty$  : boolean
 $isempty \leftarrow (c.count = 0)$ 
```

This subroutine is rather simple. To invoke this subroutine to check if a particular circular queue x is empty, we use the following pseudocode:

```
invoke is circular queue empty  $x \rightarrow c, result \leftarrow isempty$ 
if  $result$  then
  print "the circular queue is empty"
else
  print "the circular queue is non-empty"
end if
```

Check for Full

The logic to check for a full circular queue is similar to the one to check for empty. We only need to compare the “count” field to 8 instead of 0.

Add to Tail

The algorithm to add to the tail slot is as follows:

```
define subroutine add to circular queue tail
given and provide  $c$  : circular queue
given  $s$  : slot
provide  $result$  : boolean
invoke is circular queue full  $c \rightarrow c, result \leftarrow result$ 
 $result \leftarrow \neg result$ 
```

```

if result then
   $c.a_{(c.head+c.count) \bmod 8} \leftarrow s$ 
   $c.count = c.count + 1$ 
end if

```

The **invoke** statement may look a little confusing. This is because the subroutine being called, “is circular queue full”, also has formal parameters c and $result$. The negation of $result$ is necessary because “add to circular queue tail” provides a $result$ of true if and only if it successfully add to the tail of a queue. If $result$ is true from the **invoke** statement, it means the queue is already full and therefore the add operation fails. After the negation of $result$, $result$ is true if and only if there is room to add to the circular queue.

The statement $c.a_{(c.head+c.count) \bmod 8} \leftarrow s$ also deserves a little explanation. $c.head$ is the index of the head of the queue. $c.head + c.count$ is the index of the tail (next available slot). However, this sum can exceed 8 (e.g., when $c.head = 7$ and $c.count = 3$). $(c.head + c.count) \bmod 8$ ensures the number is between 0 and 7. This number is then used as the index for the array “a” (as a field) of a circular queue record c . The indexed element of this array is finally used as the destination so we can copy s (the intended contents of a slot) to a slot in the circular queue.

Remove from Head

The algorithm to remove contents from the head of a circular queue is as follows:

```

define subroutine remove from circular queue head
given and provide  $c$  : circular queue
provide  $s$  : slot
provide  $result$  : boolean
invoke is circular queue empty  $c \rightarrow c$ ,  $result \leftarrow isempty$ 
 $result \leftarrow \neg result$ 
if  $result$  then
   $s \leftarrow c.a_{c.head}$ 
   $c.head \leftarrow (c.head + 1) \bmod 8$ 
   $c.count \leftarrow c.count - 1$ 
end if

```

This subroutine is similar to “add to circular queue tail”. $result$ is a formal parameter to indicate whether a slot was successfully removed from the head of the provided circular queue. s is a formal parameter to contain the contents of the removed slot.

The only part that requires explanation is $c.head \leftarrow (c.head + 1) \bmod 8$. After we remove one slot from the head, the head must advance to the next slot in the carousel. However, we may be advancing from index 7 to index 0. This is why we need to use the modulus operator to ensure the resulting number (the next head index) is still between 0 and 7 inclusively.

16.4 Circular Queue Implementation

Circular queues are very common data structures, there is no need to reinvent the wheel. This section describes what I have already implemented.

16.4.1 cq.h

cq.h is the header file of circular queues. It describes everything that an application programmer needs to know about a circular queue.

```
\input {../samples/cq.h}
```

Let us examine the three functions that you need to access a circular queue.

```
void cqInitialize(struct Cq *p, void *, unsigned size);
```

`cqInitialize` is used to initialize a circular queue. The first parameter `p` is a pointer to a `struct Cq`. The second parameter (no name in the prototype) is a pointer to a chunk of memory for actually storing bytes. This is usually a piece of memory that is statically allocated. The third parameter, `size`, is an unsigned integer that indicates the number of bytes available in the second parameter. This subroutine does not return any value.

```
int cqPutByte(struct Cq *p, char byte);
```

`cqPutByte` is used to add/insert a byte into an initialized circular queue. The first parameter, `p`, is a pointer to the circular queue (to which a byte will be added). The second parameter, `byte`, is the byte value to put into the circular queue. This subroutine returns 0 if and only if there is no room left in the circular queue to store the byte.

```
int cqGetByte(struct Cq *p);
```

`cqGetByte` is used to remove/retrieve a byte from a circular queue. The only parameter, `p`, is a pointer to the circular queue from which a byte will be removed. The subroutine returns a value from 0 to 255 if the retrieval is successful. It returns a negative value if and only if there is nothing in the circular queue to remove.

16.4.2 UART ISR Logic

Given that we have a circular queue structure, we can now define the logic to handle various interrupts generated by the UART.

When the receive buffer has a received byte, the following should be done:

```
b ← UDR
if cqPutByte(uart_rcq, b) == 1 then
    we're done!
else
```

```

    flag software buffer overrun error
end if

```

When the transmit buffer UDR is empty, the following should be done:

```

i = cqGetByte(uart_tcq)
if i < 0 then
    turn off the UART transmitter
else
    UDR ← i
end if

```

When transmission is completed, the following should be done:

```

disable transmitter driver

```

16.4.3 UART API

We now define how the rest of the system sees the UART.

First, we need to define a subroutine to add a byte to the transmitter circular queue:

```

save global interrupt enable
disable global interrupt
given b to add to the queue
if uarttcq has space then
    call cqPutByte(uart_tcq,b)
    enable transmitter driver
    enable UART transmitter
    enable UART transmitter interrupts
    restore global interrupt enable
else
    restore global insterrupt enable, return error, there is no space left
end if

```

The subroutine to receive a byte is simpler:

```

save global interrupt enable
disble global interrupt
i = cqGetByte(uart_rcq)
restore global interrupt enable, return i

```

Note that this is a very simple API to use an UART. It lacks certain nice features. For example, if the transmit circular queue is full, the requesting code must try to transmit later. Likewise, if the receive circular queue is empty, the reading logic has to attempt to read later. This easily leads to polling code, which is inefficient and makes program logic more difficult to understand.

Part VII

Robot Behavior

Chapter 17

Threading

This chapter discusses multithreading, a concept more commonly found in larger operating systems.

17.1 Life without Multithreading

To understand why multithreading is helpful, we first have to understand what it was like without.

Most robots require parallel algorithmic logic of some kind. For example, a rescue robot needs to handle the following tasks in parallel:

- Maintain communication with the rescue command.
- Monitor its own vital status (temperature, battery voltage, etc.).
- Process information from sensors.
- Control motion to get to a certain destination.
- Plan the most effective course to handle the current crisis.

Without multithreading, we need to implement each parallel task as a state machine. A state machine is often coded as follows:

```
struct SomeSM
{
    unsigned state;
    // state machine private properties
};

void SomeSMTick(struct SomeSM *p)
{
    switch (p->state)
    {
```

```
    case STATE1:
        // ...
        p->state = STATE2; // change state
        // ...
        break;
    case STATE2:
        // ...
        break;
    // more states
}
}
```

Part VIII
Project

Chapter 18

Project

This is your final project.

18.1 Robot Objectives

The first section of your project states the objectives of your robot. Use examples to illustrate *what* your robot is supposed to do. For example, the following describes what a fire fighting robot does:

- patrol the interior of buildings
- identify potential fires
- notify the fire department
- put out the fire with internal extinguishers

18.2 Robot Specifications

In this section, flesh out more details, in technical terms. In our example, the specifications may look like the following:

- to patrol:
 - optimal speed to be about the same as fast pace walking speed, just 4 miles/hour.
 - obstacle avoidance to avoid injuring people or causing property damage.
 - robot base to have about the same width as a person, about 24 inches.
- to identify fires:
 - be able to detect open fire from properties of flames

- be able to detect fire from smoke
- be able to detect fire from carbon-monoxide
- be able to detect using infrared thermo-sensors
- to notify fire department:
 - on-robot siren and flashing lights
 - radio transmission via wireless/cellular technology to 911
- to extinguish fire:
 - elevating extinguisher head with tilt and swivel control.
 - sensors on the extinguisher head for aiming, logging and remote control purposes.
 - equipped with usual fire extinguisher canisters.
- self maintenance:
 - low battery detection.
 - automatic recharge mechanisms.
 - dual battery for hot swapping.
 - self diagnosing on critical components
- user friendliness:
 - LCD screen and keypad for on-robot configuration.
 - wireless communication for on-the-run configuration, logging or debugging.
 - total mass not to exceed 40 pounds.
- safety:
 - optional remote kill switch
 - optional remote control unit
 - automatic shutdown if it fails self diagnostics.
 - audio/visual alarm if it fails self diagnostics.
 - radio/audio/visual beacon.

18.3 Design

In this section, describe how the robot is designed. This can get very long, tedious. I am looking for analyses and explanations. Why do you choose a particular design/component? How does the chosen one compare to alternatives? What are your selection criteria?

In our example, the design section may look like the following:

18.3.1 Drive Platform

- motors:
 - choose to use auto windshield wiper motors
 - only requires 12V, commonly available from car batteries
 - quiet and plenty of torque
 - not very efficient, but since we can recharge, it is not a major issue
 - inexpensive and reliable, built for continuous operation
 - easy to mount wheels
- wheels:
 - small 8" to 10" wheels designed for hand trucks or go-karts
 - pneumatic tires for traction and bump absorbing.
 - commonly available

18.3.2 Fire Extinguisher Subsystem

18.3.3 Sensors

18.3.4 Communication Subsystem

18.3.5 Battery and Power Subsystem

18.3.6 Control Units

This section is particular important for our class, since software runs on the control units.

- main controller
 - responsible for coordination, communication
 - choose Mini-ITX based controllers
 - include cooling (thermo-electric combined with fans)
 - less expensive than PC104s
 - powerful enough to run Linux
 - use compact flash as solid state file system
 - use USB-blue tooth for wireless communication
 - use USB-cell phone for communication with fire department
 - use USB-serial to communication with other controllers
- motion controller
 - MCU based design for compactness and cost reduction

- communicates using serial protocol, RS-485
 - accepts high level commands
 - monitors motor temperature and current consumption
 - monitors output voltage of batteries
- sensor controller

Part IX

Robot Design Log

Chapter 19

Robot Design Log

19.1 Design Requirements

19.1.1 Physical

- up to 10cm by 10cm (for mini-sumo rules)
- up to 1.1 pounds, including all components

19.1.2 I/O

- able to drive 2 bi-polar driven stepper motors
- able to drive 2 DC motors
- able to control 2 R/C servos
- 2 quadrature encoding input
- 2 to 3 ADC input for GP2D12 sensors
- 2 to 3 phototransistor sensor ports
- programming port
- LEDs
- DIP switch
- battery connector
- on/off switch
- push-buttons
- RS-232 interface

- RS-485 interface (added 20040307)
- RS-485 break detection by hardware, tie to reset circuit (added 20040307)
- jumper to select auto/run/program modes (added 20040307)

19.1.3 Software (added 20040307)

- gdb remote serial protocol bootloader
- supports gdb remote serial protocol
- bootloader to support the following modes:
 - run: always run application program do not enter program mode
 - program: always enter program mode, do not enter run mode
 - auto: after resets, listen for RSP traffic for 10 seconds (make this programmable?). If there is no RSP traffic in 10 seconds, enter run mode. Otherwise, enter program mode.

19.2 Components

19.2.1 Pin count

- 2 L293Ds, one with PWM, the other one without, 10 pins
- (alternative to above, added 20040307) 4 mc33186
- (alternative to above, added 20040307) 4 tpic0108
- 3 ADC channels, 3 pins
- 2 quadrature inputs, 4 pins
- RS-232, Rx/Tx, 2 pins
- RS-485, Rx/Tx and enable, 3 pins (added 20040307)
- run/program mode, two pins (added 20040307)
- programming port, clk, in, out, 3 pins
- 3 phototransistor ports, 3 pins
- 2 LEDs, 2 pins
- 4 position DIP switch, 4 pins
- 2 push buttons, 2 pins

A total of 33 I/O pins are needed. This rules out the DIP packages. For flexibility, might as well use the ATmega128.

19.2.2 Component Selection

- 1x ATmega128, TQFP version
- 2x 0.1uF decoupling cap for ATmega128
- 1x 10uH inductor for ATmega128
- 1x 16MHz ceramic oscillator
- 2x Ti 754410 (equiv. to L293D)
- 1x 5V LDO regulator
- 1x jumper to select whether to use regulator reset or not
- 1x push button to reset board manually
- 2x software readable push buttons
- 1x RS-232 switched cap level convertor
- 4x 1uF cap for RS-232 level convertor
- 1x RS-485 driver chip
- misc. resistors for RS-485 termination
- 1x DE9 connector for RS-232
- 1x resistor pack for IrED current limit
- 1x resistor pack for phototransistor pull-up
- 1x resistor pack for quadrature encoder IrED current limit
- 2x LED (different colors)
- 1x LED power indicator
- 1x 6-pin programming header
- 2x 3-pin servo connectors
- 2x 2-pin DC motor connectors
- 2x 4-pin bipolar stepper motor connectors
- 2x 6-pin quadrature encoder connectors
- 3x 3-pin GP2D12 connectors
- 3x 20uF cap for GP2D12 connectors
- 3x 1uF cap for GP2D12 connectors

- 3x 4-pin IrED/phototransistor connectors
- 1x 4-position DIP switch
- 2x EE-SX1131 for quadrature encoder

19.2.3 Mini-Sumo Chassis Design

-

19.3 Design (added 20040307)

19.3.1 Bootloader (added 20040307)

The bootloader is to be loaded with a monitor capable of handling the gdb remote serial protocol (rsp). RSP should listen to an UART connected to RS-485 for long-range noise-tolerant communication. Due to the small page size of the ATmega128, we should be able to debug a program via this interface.

A special hardware circuit is used to detect the break signal. A break signal longer than a threshold is used to reset the board. This feature is jumper selected so that we can disable this resetting for an application.

This special hardware can be done with a small signal transistor, an RC circuit and a comparator (or schmidt trigger) and a diode (so that reset is always sunk and never driven).

The use of a bootloader also means that the vector table is relocated. Special linker scripts must be used instead of the default avr-gcc and avr-binutils ones.

19.3.2 H-Bridges

The new surface mount H-bridges are efficient, but they require bottom soldering to pads with vias for optimal heat dissipation. This poses a problem with hand soldering because it is generally speaking difficult to do this without specialized tools.

The motorola TSOP package is the best option because at least there is some exposed area from the bottom heat sink plate for external hand soldering. The package is also relatively small (15mm by 16mm). This part is capable of handling 5A continuous!

Due to the high current ability and requirement of a large heat sink area, it may not be the best option to put these H-bridges on the main board. A separate board with separate connectors for power and control signals makes a design more versatile.

An alternative is to use discrete transistors (high efficiency N-MOSFET) and a switch cap voltage doubler. This approach makes the design more flexible in terms of component selection, but it is somewhat expensive in terms of soldering.

In order to limit the gate voltage, it may be necessary to use a zener diode that connects from gate to ground. A pull-up resistor than connects from gate to the doubled voltage.

Update 20040319: found a new way to use the TPIC0108. use a large via on the underside, and solder the heatsink to the via from the underside.

Update 20040324: assigned pins from PAL to the H-bridges. The assignment is mostly based on routing needs.

19.3.3 (20040322) RS232

A usual MAX3221 chip will be used for RS-232 PC Interface. It requires four 0.1uF caps for the voltage doublers, but otherwise a easy chip to use.

Will use a SSOP RS-232 transceiver chip that only needs four 0.1uF caps. This chip is tiny, doesn't take up too much board space. The land pattern is entered in EAGLE.

Update (20040324): Decide to use TXD0/RXD0 for the RS-232 port. The rationale is that the TXD1/RXD1 are external interrupt capable. To resolve the conflict between the RS-232 driver and the in-system programming port, we can use the negation of the reset line to disable the RS-232 chip. We can also use the PAL to select one of the two sources to drive the pin: ISP or RS-232 ROUT. Haven't decided yet.

Another reason to preserve TX1/RX1 is that other external interrupt lines dual-purpose with output compare. For maximum flexibility, we need four output compare (for software chopper drive). As a result, I want to use TX1/RX1 for external interrupt purposes.

19.4 PCB Design Log

19.4.1 20040410

Yes, the whole break is over, and I am not done with the design yet (sigh!). On the other hand, it's fortunate that I am not done with it yet. Read on.

The major hang up was the routing of the pulse IrED circuit. It is still not completely done, but the remaining portions should be relatively easy. Still need to add the AD5245 digital potentiometer to the library.

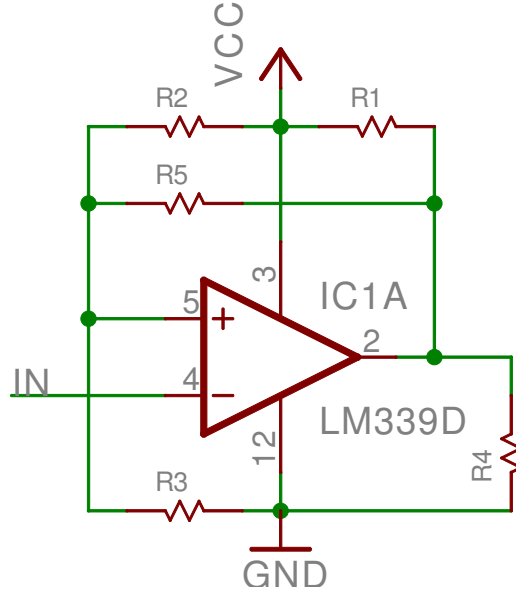
I also found out yesterday that I'll need to include a comparator on the PCB. Most quadrature encoders output real output from the phototransistors. This means we'll see a smooth and continuous voltage swing that is not friendly to CMOS inputs. If the extremes of the swing are close enough to 5V and 0V, we can easily use a schmidt trigger so the signal doesn't bounce. However, this is not always possible.

The reason is that "black" is not necessarily black enough. The density of toner from a normal laser printer is not dense enough in the IR frequency. As a result, *enough* IR leaks through the "opaque" portions, making the so-called "off current" not really close to no current.

As a result, comparators must be used, combined with a variable voltage divider bridge, to generate the correct output. The LMV339 (a low voltage variant of the LM339) is one option. It is inexpensive at less than \$1 each

(at single quantity). The datasheet from National Semi detailed how to add hysteresis. A single chip is enough for two quadrature encoders, although we will need a number of resistors (3 per comparator, yielding a total of 12 for the whole set up).

The basic circuit is illustrated as follows:



Here's how it works. When the output of the comparator is high, the voltage at the non-inverting terminal is as follows:

$$V_h = VCC \frac{R3}{R3 + \frac{1}{\frac{1}{R2} + \frac{1}{R5}}} \quad (19.1)$$

When the output of the comparator is low, the voltage at the non-inverting terminal is as follows:

$$V_l = VCC \frac{\frac{1}{\frac{1}{R3} + \frac{1}{R5}}}{R2 + \frac{1}{\frac{1}{R3} + \frac{1}{R5}}} \quad (19.2)$$

This means the input voltage must drop below V_l for the output to switch to high, and the input voltage must rise above V_h for the output to switch to low. This logic appears to be inverted, hence the name “inverted”.

Note that the “pull-up” resistor, R1, is necessary because the comparator has an open-collector type output. Without a pull-up resistor, the device cannot drive the output high (pin 2 of IC1A). R4, the load resistor, is not actually there. It is merely a symbol that denotes the impedance of the load (in our case, it is in the range of mega Ohms).

19.4.2 20040406

Just realize that we don't need a small signal N-MOSFET. The 4051 "tri-states" the outputs when enable is negated. This means that a simple pull-down resistor (10kOhm or so) on each power MOSFET should disable the output!

Because the opamp has an output capacity that matches the input of the mux (about 20mA at 5V), we can use a relatively small current limiting resistor (220Ohm or so).

For cost and soldering considerations, a Darlington array (such as the ULQ2003 or ULN2801) may be used. The main tradeoff is efficiency, as a Darlington device has close to 2V of drop at 500mA. On the other hand, a single SO18 package is much smaller than four SO8 packages.

A tiny transistor is NDS331N (or NDS351AN). It is a SOT-23 3 pin package that is capable of up to 10A pulsed (at a very low duty cycle). This part is also very cheap. Compared to the SOT-6 package, it is easier to solder because of more space between the pins.

The NDS355AN has the same package, but it has a specification of 1.7A continuous and a lower on resistance. The price is \$0.48 each, and \$0.27 at 100 quantities. This is probably the best part to use for driving the IrEDs.

19.4.3 20040404

Yet another revision.

Instead of using a negative supply, I am switching back to use a single rail opamp for current control. This is because I already have a way to shut down the output completely.

The Burr-Brown OPA340 or Microchip MCP601T opamp is tiny (SOT23), and it is rail-to-rail. Combined with the IR7301 (SO8 for two N-MOSFETs), the IrED pulsing circuit has the following components:

- an AD5245 for current selection
- an OPA320 for current feedback and gate control
- a 2.2kOhm resistor from the output of the opamp to limit control current
- an 2N7002 sot-23 N-MOSFET to disable the drive voltage
- a 4051 to mux the gate control voltage
- a 10kOhm resistor network to pull power FET gates to ground
- a common 0.5Ohm or 1Ohm current sense resistor
- four IR7301 power FETs

19.4.4 20040331

Need to consider adding a negative supply to the board. This is necessary for the op-amp for current control. BB1 had a similar circuit based on a single rail op-amp, but the result is somewhat mixed due to the near-rail (0V) nature. A negative supply removes the requirement of the op-amp to operate near rail, hence making it possible to use less expensive non R2R op-amps such as LM324s.

The Maxim1044 SO8 (also known as ICL7660) chip seems to be a good choice. Although not as cheap as some other switched capacitor charge pumps, this one has a very high efficiency.

Combined with a digital pot, the board can adjust the reference voltage applied to the non-negated input of the opamp to set the current. To turn off the device, we can simply ground the non-negated input using one I/O pin from the MCU. We can also ground the output of the opamp (via a resistor of about 5kOhm) to ground so that the MOSFET is disabled. In order to make sure that the MOSFET has a safe reset state, however, it is better to drive the input to the MOSFET through another small signal MOSFET. The small signal MOSFET is enabled by an output of the MCU that has an external pull-up resistor. This way, the small signal MOSFET has a reset state of on, which turns off the MOSFET to be controlled by the circuit.

A good digital pot to use is the AD5160 from Analog Devices. This one has the A-B terminals not tied to power and ground. Unfortunately, it uses an SPI interface (instead of a multidrop I2C bus). The AD5245 is a similar device using I2C, which makes it more flexible. I am inclining to use the AD5245 instead of the AD5160.

(20040402) bad news. The AD5245 cannot handle negative voltage relative to ground, we need to switch to a DS1666. This part has a -ve bias so the low terminal can go below 0V.

In order to have good control over the reference voltage, We should use the series of 20k-5k(pot)-24k from positive supply to negative supply. This ensures that we have most of the 256 divisions of the digital pot working over the useful range of reference voltage. A 24kOhm 0805 resistor with 0.5% precision is about 15 cents each (RR12P24.0KDCT from Digikey).

Once the pulse current control circuit is set up, we need work on a multiplexer that directs the pulse current to different series of IrEDs.

To make things easier, a device like SN74CBT3251 (1-of-8 demux FET) can be used. However, this type of device is relatively inefficient. The on-resistance is in the range of 20 Ohms or so. The CD4051 can also be used to mux the drive signal (after the current limiting resistor from the op-amp) to 8 individual MOSFETs.

19.4.5 20040322

bb2-20040322.pdf illustrates the current board layout. The H-bridge layout is almost done, except that I have not put in the top ground plate for heatsinking purposes. This layout is chainable so that I can add additional H-bridges next to

the current one easily, while preserving excellent ground and Vdd connectivity.

19.5 Drive System

C&H Sales has a \$60 motor geared down with encoder. Stock# DCGME2201. Output shaft is 12mm by 7/8 inch, 3mm cross drilled hole. Encoder has 500 steps per revolution.

C&H Sales #DCGM2103PR L/R pair of wheelchair motors, 17mm diameter shaft tapped for 8mm by 1.25mm screw, 6mm wide key slot

#DCGM2003 12VDC, shaft is 3/4 inch diameter, standard 3/16 inch straight keyway. \$140 each.

Tank tread set at robotmarketplace.com, \$180/set. Drive sprocket has a 3/4 inch inner diameter.

NPC-PH804 from robotmarketplace.com is a hub to fit 3/4 inch bore with 3/16 inch keyway.

3/16 inch key stock is also available from robotmarketplace.com.

Hub with 3/4 inch inner diameter and 2.75 inches bolt diameter is available at robotmarketplace.com (NPC-PH804).

Chapter 20

A.N.T.

ANT (autonomously navigated transportation) is an experimental/vapor project that is a side effect of this course. You are not required to read this chapter. You may find some of the topics interesting, but I cannot guarantee that anything is accurate or correct in this chapter.

In other words, read at your own risk.

20.1 GPS

20.1.1 Interface

A serial port should be fine to interface with most GPS units. We can get handheld GPS units that come with computer ports, or get GPS units that are intended to interface with computers. It is not worth it to get GPS modules that require housing because they are usually more expensive than GPS units that have the circuit board, antenna and housing all integrated.

20.2 Stereo Vision

20.2.1 Theory

With two or more high resolution optical imagers, it is possible to use triangulation to provide stereo views of the road and terrain. It is important to emphasize “high resolution” because low resolution images do not contain enough information to determine the size and distance of distant objects. It is critical that a robot be able to see and determine the terrain for as far as possible.

To perform stereo vision interpretation, two images are taken at about the same time. The only critical part for terrain mapping is that the two images be taken from different points, and the robot does not move between the time of the two pictures taken. This means that one camera can be sufficient, as long as

the robot does not move and an articulated arm is used to change the camera position and view.

Once two images are acquired (from different points and views), software can start to identify common objects. Because of the change of view and angle, objects will not be identical in the two images. As a result, identifying the same object in the images requires some processing.

Once objects are identified, triangulation can be used to determine the distance to the objects. Note that many hints can be used. For example, images taken earlier when the robot is at a different location can be used, combined with the encoded distance traveled. This can help to confirm the location of known objects. Once the distance to known or tracked objects is confirmed, the two frames can be correlated and distances to new objects can be deduced.

This method is rather useful because it does not rely on extreme accuracy of placing and aiming the cameras. Instead, it uses clues already embedded into the pictures to help resolve distances.

In order to help calibration, laser beams can be used to help identify objects with little contrast.

Object Id

The key to interpret stereo-vision is to id objects. This is easy for people because our eyes and brains have massively parallel processing to recognize objects. For a computer, however, this is not an easy or natural task.

The general question is: given this region of pixels in this image, where is this region in the other image? We are looking for a spatial displacement of a region of pixels. Although this may seem like a very difficult problem, we do have some clues that can be helpful.

- it cannot be too far away because there is only so much the cameras can be spaced apart, and only a particular range of distance to objects
- for vertically displaced images, the two regions must align vertically, for horizontally displaced images, the two regions must be aligned horizontally.
- for a tracked object, we already know approximately where a region of it should be located
- objects that are at a far distance should not be displaced at all (use them as anchor points)

To do this, both images should be pre-processed so that instead of looking at pixels, we can focus on larger macro constructs. Basic straight lines, curves, shapes and etc. can be recognized with parameters (angle, length, orientation, curvature and etc.). This facilitates the comparison of macro constructs between two images.

With high resolution scans, there can be *a lot* of details in an image. It is unnecessary to identify each piece of small rock. As a result, the analysis

algorithm should use a divide and conquer approach to first identify larger macro constructs, then break the larger constructs into smaller ones as necessary.

20.2.2 Equipment

Although one can spend much money, there are existing off-the-shelf solutions. Most digital cameras can be operated remotely via an USB connection. A 4 megapixel digital camera has sufficient resolution for stereo vision. In Linux, a program like `gphoto2` can be used to acquire images from digital cameras remotely.

It may be necessary to protect consumer-grade digital camera from harsh environment. A dust-free sealed box with temperature control may be sufficient for some applications.

Chapter 21

Project Log

21.1 20050525

For larger robots, we can use suspension equipped bicycle seatposts. We can also consider using elastomer rubber bung.

An interesting option is elastomer pellets. At eBay, US\$20 gets four pounds. It needs to be placed in some sliding cylinder/piston structure. I have no idea how much of this stuff a robot will need as a shock absorber.

Minerelastomer.com seems to have some useful bumpers.

21.2 20050525

At vetcosurplus.com, they have sealed aluminum boxes. These are perfect as an enclosure for oil-filled electronics for better cooling.

Have finally decided to use pre-built DC-to-DC converters. This saves space and the trouble of debugging switching regulators. Digikey has some. This clears the path to use the HIP4082 FET driver chip to drive N-MOSFETs. The whole circuit board is going to be submerged in oil inside an aluminum sealed case. This way, there is no need to mount individual air heatsinks to the chips.

The only remaining problem is to get the cables through. We can always drill a hole, get the cable in, then use sealant to plug the hole. However, oil can leak through capillary effect. A solution is to use a block to separate inside wiring from outside wiring.

One option is to use a Molex connector pair. The crimp pins can be coated with epoxy to stop oil leaks. We can also consider soldering the crimp connectors, then coat just the part between the conductor and the insulator with epoxy. The molex connector housing is then fitted in a opening cut just for it. The connector needs to be completely sealed with sealant, and fixed to the opening. JB Weld may be used.

For the main regulated power supply, a power supply designed for car may be used. The only concern is whether such supplies can deal with the voltage

dip when the motors draw current. There is one that has a specification of 10.8V to 20V (from powerstream.com). It has quite a bit of power for 5V and 12V applications.

21.2.1 Interface to oil submerged box

The oil submerged box only needs to deal with high power devices that generate heat. It'll need the following interfaces:

- main power from battery and ground
- conditioned (after TVS protection) power to other components
- + and - terminals to motor 1 (2)
- + and - terminals to motor 2 (2)
- control signals to each N-FET (8 total)
- thermister terminals (2), may be combined to the same header as the control signal
- regulated 5V and paired grounds (5 pairs)

21.2.2 Components

- 8 high current N-FETs
- 1 high current 14V bi-directional TVS
- 1 5V switching regulator in DIP format
- 1 12V switching regulator in DIP format (maybe)

21.3 20050129

Happy new year (after 9 months of no activities!).

Got a consulting application that requires the use of windshield wiper motors. So I am now looking into big-and-hefty electronics. After a little bit of research, I have decided to use N-channel MOSFETs for the switches, and use some high-side FET drivers to handle the high-side switches.

The windshield wiper motors are modified to work at 24V (instead of 12V). This improves torque and speed at the same time. Furthermore, from some review on the web, this modification also makes the forward and backward speeds more comparable.

As far as parts are concerned, I am inclining to use the National Semiconductor LM9061M as the high-side FET driver. Compared to the Linear Technology, Maxim and International Rectifier alternatives, the LM9061M is inexpensive, and it will get the job done. It does require at 7V for the supply

(bootscrap) voltage. Ideally speaking the Intersil isl6801 is even better because it only requires 5V as a bootstrap. The full-bridge driver Intersil HIP4082 is even better because it can interface with four N-FETs directly. The HIP4082 does require a 12V supply, which should not be a problem. The same applies to the HIP4080 and HIP4081, which are more expensive due to their internal charge pump. Note that the HIP408x family offers shoot-through protection and programmable dead time, which are extremely important safety features that should not be implemented in software. The HIP4081 has a better interface for software control.

For the actual switch, any N channel FETs that handles up to 12V at the gate can be used. For example, the International Rectifier IRFZ48V can handle V_{GS} up to 20V, and the FET can switch up to 72A (continuous) with a breakdown voltage of 60V. This part is also quite inexpensive at US\$0.84 each at quantity 10.

In order to support passive braking, we need a hefty diode that can be used to loop back current. Ideally speaking, we want all of the energy be dissipated by the motor itself. However, this is not possible due to the forward voltage drop of a diode. Diodes that are rated at 40A or more are quite expensive and big. The only exception is the On Semiconductor MBR6045WT (\$3.29 at quantity 10).

Note that the loopback diode needs to handle the maximum motor current continuously because the energy to dissipate is not due to the collapsing magnetic field of the coil winding, but rather the mechanical momentum of the robot itself. We know that this current cannot exceed the stall current of the motor because of coil resistance.

The loopback diode may not be needed because the N-FET has a specified continuous source current of 72A (max.). According to the diode forward voltage curve, the forward voltage is about 1V at 72A. This means the package has to dissipate 72W. With proper heat sinking, the package has a thermal resistance of about 1.5C/W. This means that at 72W, the temperature is just about at the maximum (150W).

If we don't have to put in external loopback diodes, the set up can be much cheaper (because the diodes are 3 times the price of FETs!).

Without loopback/flyback diodes, we still need a TVSes to help protect the switches against inductive electrical potential. This is a lot cheaper, though. For example, the Diodes Corp. 1.5KE series are typically less than \$1 per unit (for bidirectional, C suffix). For a 24V system, the 27V or 30V parts are appropriate (for a little margin). The exact part numbers are 1.5KE27CA and 1.5KE30CA.

With the TO220 package, there are a few heat sinking options. Although a wiper motor only has a current draw of about 4A to 5A continuously, it is best to design the board to accommodate for more current capability.

The relatively cost effective Aavid 529902b02500 has a rated 4.5C/W thermal resistance. With a total 6C/W, we can still easily switch 20A with the aforementioned inexpensive MOSFET (with no airflow). The only drawback is that the size of the heatsink is a bit big, requiring a foot print of 42mm by 25mm.

A different technique is to mount the TO-220 transistors on the case using

thermal adhesive. This gives us more freedom as far as heat sinking is concerned. However, because the large heat sinking pad is tied to drain on the MOSFET, we cannot use a single heat sink.

21.4 20040326

Finally got batteries and got them charged. The wheelchair still won't move. When it is turned on, the control box beeps. The voltage is good, though. Check all the connections and they are all good. Don't know why the control box beeps.

The steering mechanism does work. The drive wheel turns left and right. However, forward and backward does not work at all.

I am guessing that either the control box is shot, or the motor itself is shot. I am going to ohm out the motor terminals later (somehow) and see if they are shorted. Oh, when the control box is not connected to the motor, the beeping stopped. Battery voltage is verified to be good (all the way to the green side).

I *hope* that the problem is only due to the power component in the control box is shot. I can probably either fix the circuit or design a replacement circuit. However, if the problem is with the motor, then I'll have to replace that.

Oops, it was my own operator error. After reading the manual of the MPV4 (which, by the way, is quite a bit more modern than the original MPV), I realized that I had to switch from "brake" back to "run" to make it go. Duh!

The wheelchair now operates just fine. It propelled me without any problem. The next challenge is to decode the signals coming out from the hand control unit and replicate that using a controller. It seems easiest to decode from the user interface box after I open it up.

21.5 20040319

The charger that came with the battery doesn't seem to charge the battery. I think the connector is wrong. Anyway, I am now using my automobile battery charger, set to 2A charge rate (instead of 12A). Each 12V battery that came with the wheelchair is rated at 33AH, it'll take a little while to fully charge both.

I think I have already determined to use a mini-ITX computer as the main controller. It'll be running Linux with most peripherals connected by USB or serial (via USB-serial convertor). The one from <http://www.jkmicro.com> seems to be a good choice, with a switching power supply and two included CF slots, it has all the components that I need. Well, almost. I may want to add my 802.11b WiFi card to the PCI bus, or a CF version so I can program it remotely.

The wheels on the wheelchair are not encoded. Because all three wheels are mounted fairly close to some chassis component, I don't think it'll be too difficult to mount some hall effect sensors and magnets (on the wheels). For

outdoor use, magnetic encoding is the only choice because optical encoders are too sensitive to dirt.

The back steering mechanism also needs to be encoded. This is a little more tricky because I need to know the exact angle. I can put one extra magnet so that it can indicate the “home” position, then I can track the current angle using the other sensors. This also means that when the bot resets, it needs to first control the steering mechanism to find the home position.

My first task, after the batteries are charged, is to test the wheelchair using the user controls. If all works out, then I can start to reverse engineer the control mechanism and see how I can interface that with a microcontroller. I am suspecting the control mechanism (joystick) is really just two potentiometers, and I replace them with two digital pots easily.

Just tested the batteries. One doesn't charge, but the other seems okay. Now I need to go find a replacement so I can power up the vehicle for the first time. eBay has U1 batteries sold for 55 dollars for two, so that's the way it'll go.

21.6 20040318

Wheelchair finally arrives! The condition is as expected, but I have no way to test it yet. Need to get two 1U 12V batteries first. Can't spend any more money on it now, must wait...

Looked up USB GPS for Linux. Only the ones with Prolific PL2303 should be used. UC-232 and IO-Data USB to serial adaptors may also work. To be safe, may need to get a serial GPS, then get an USB to serial converter so I don't use up one serial port.

GPS 18 LVC from GPSCity seems to be a good choice. It has barewire output/input for maximum flexibility.

Part X

Work In Progress

Chapter 22

DC Motor Encoding

Since we can use optical encoding, this topic may appear to be quite dead. However, optical encoding has many disadvantages. First, it is somewhat difficult from the mechanical perspective. Second, it has fairly limited resolution unless the encoding is performed near the motor axle.

This chapter discusses a method that can potentially perform encoding at axle resolution.

22.1 The EMF

Whenever a current circuit is broken, the collapsing magnetic field produces enough voltage to push current through initially, then the current falls off according to the LR formula. Most of the time, this EMF (electro-magnetic feedback) is undesirable because it can potentially break semiconductor devices.

On the other hand, this EMF can also be used for encoding purposes. A DC motor has an armature that has several contacts that touch the brushes on the stator. As the contact opens, the EMF generated can potentially be detected by a circuit. As a result, we can encode motor rotation based on EMF detection.

22.2 The Problem

The main problem of this approach is noise. Although in theory the contact remains closed until it moves out of the reach of the brush, it is noisy in reality. In other words, the brushes bounce as it makes contact with the contact on the rotor. This generates a lot of EMF noise spikes that interfere with motion encoding.

22.3 The Solution

If we know the bouncing characteristics, we can potentially use a software approach to filter out the bounces. We do need to assume that the time between bounces is significantly shorter than the delay due to contact rotation.

22.4 Detecting Circuit Open

This is easy. As a circuit opens, it generates an EMF spike.

22.5 Detecting Circuit Closure

This is harder than the previous case. The only detectable clue is the ramping up of current due to the inductance of the rotor coil. We can potentially use a current sensing circuit to detect this transition.

22.6