Programming Aloft and Below

Robert Swan

Contents

Preface

(For people who already know about programming)

I'm mad. Obviously.

It can be the only explanation. Do I really think beginners are going to be able to cope with an assembly language when many beginners already fail to get anywhere with a colourful and friendly graphical IDE?

Well... yes, I do.

Let me explain what led up to my creating ArdEx and writing this book.

At dinner with a friend and former colleague, now a high school teacher, conversation turned to a task she had recently been set: to come up with a course in robotics for the next year. The school had already bought hundreds of Arduino UNOs and it was thought she might use these in the course. She gave me a couple of the Arduinos to try, to see if I had any suggestions.

I had been dabbling with hardware for a fair while, but had never used an Arduino. I tried a few sketches. Easy enough, but the friendly environment shielded the user from nearly everything. Almost a paradox that here you are with the hardware in your hand, but all your contact with it is through other people's work.

Compounding this was that the bulk of the work happened on the PC. You edited and compiled the code there and only transferred it to the Arduino at the last step. If you wanted to change anything it was back to the PC editor. How do you explain to your students that they're working on Arduinos when it's obvious they're working on PCs?

I looked around at various ways to make the environment less numbing. Tried a couple of flavours of Forth (weirder than usual) and found a program called *Bitlash* that wasn't too wide of the mark. Still not really what I was looking for.

Why not write my own friendly command shell talking to the hardware directly? Good question. Ok, but what exactly to write?

After a few days with this in the back of my mind, I was thinking wistfully of my first home computer, the venerable TRS-80. Unlike the card-eating monster I had been using at uni, the TRS-80 powered up to a cheery `Ready>` prompt, all set for interactive BASIC. A few minutes after getting it home I had written my first program:

```
10 PRINT "HELLO"
20 GOTO 10
```

Typed `RUN` and away it went. That's the sort of experience I was looking for. Immediate. But BASIC wasn't the answer. TRS-80 BASIC had `PEEK` and `POKE` commands, and that's about as far as its exposure of hardware went.

That's when I asked "Why not an interpretive *assembly* language?". Heresy, of course, but why not? The parser would be pretty easy, and so would the execution loop. I invented a suitable language (based approximately on the TI MSP430's) and wrote an Arduino sketch to interpret that language. It made all the Atmel `ATmega328/P` I/O registers accessible at their true addresses and, very soon, I was typing in my first interactive assembly language program ever.

Yes, the blinking LED, but I went on experimenting, consulting the Atmel manual and exploring its hardware. It was surprisingly reminiscent of the old TRS-80 times. You know, fun.

My friend has tried it for herself, and with a few of her students. They typed away at `MOV` and `BIS` and `JMP` instructions and gave every impression of enjoying themselves. The idea might have some promise.

I'm sure this approach won't be for everyone. I am not a teacher, but my impression is that different people learn in different ways. I have aimed this book, albeit a few decades late, at my 14 year old self. It takes the "get your hands dirty" approach to learning, which has always worked well for me. It is not a collection of facts and figures to commit to memory, nor is it a clinical presentation of theory. This is less a computer geography text, more a travel guide, pointing out interesting areas to explore.

If that appeals to you, bon voyage.

Chapter 1

Introduction

Admiral Nelson, Captain Cook and other famous naval commanders started their careers as ordinary sailors and rose through the ranks to command. One reason they commanded so well was their earlier experience up in the rigging and down belowdecks; they knew how to get the best out of both crew and ship.

This book aims to give some comparable experience to a programmer on the path to being a "great commander" by showing what computer software really does at its lowest level.

It might seem a stretch to liken today's computer programmer to a naval commander, but on at least one point the two are alike: on board, the captain's rule was absolute; in the computer, what the programmer commands, the computer carries out to the letter.

Indeed it is even more important that the programmer give sensible commands. There *were* limits to a captain's absolute authority. If he ordered his entire crew to jump overboard without good reason, he would have faced a mutiny. A programmer's instructions are always dutifully obeyed, no matter how silly they might be. A programmer with experience in "thinking like a computer" is less likely to write foolish programs.

The *only* langauge a computer handles directly is *machine language*. It is made up of electrical patterns of 1s and 0s. Consider it the "spoken form". This book uses the written equivalent: *assembly language*.

Many smart people think assembly language a subject to avoid. It is widely considered obscure and difficult, but it is neither. What it *is*, is primitive. Like a stereotypical caveman (or a toddler), you have to put your more sophisticated thoughts in very simple terms. This process clarifies thinking which is exactly why it *should* be one of a programmer's first languages.

Assembly language is also condemned because it is not portable; you learn it for this machine, you have to learn it again for that machine. Correct, in a hair-splitting way, but it misses the point. In a car, the windscreen wiper control can be in various places, but the steering wheel and pedals are much the same in all cars. In the same way, the crucial elements are common to all assembly languages. Learn these elements and you have a good understanding how computers work. *That* is the point.

Intended Audience

Anybody interested in how computers work and how to put them to work.

This book presents ideas which, it is hoped, will make youe think. You should come away with a good feel for what the basic elements of a computer are, and how to make use of them to solve problems.

The book is intended to be instructive and conversational, trying to take some of the mystery out of computer programming. It is not a textbook, but might be used to supplement a course.

People with more programming experience will be able to cover the material more quickly, but people with no programming experience should be able to cope with it all too, and come away with a pretty good picture of what computers really do.

As for age, a smart ten year old should be able to understand most of it, but some of the material in Chapter 7 and Appendix B is quite esoteric.

How To Read This Book

Plan to work at it. While writing this book hasn't been easy, getting the most out of it will take a fair effort on your part. Mind you, it should be a *fun* effort. If it becomes drudgery, this approach is probably not for you.

Skim through the appendices before you start. There is material there that can be referred to as needed. The Command and Instruction References are short, but should be leafed through from time to time to get the most out of ArdEx.

It is probably best to work through the chapters in order. The book is short, but the ideas are fairly densely packed and there isn't much repetition. You don't want to miss anything on the way through. It would be reasonable to read each section from beginning to end, then reread it, pausing this time to load each example into ArdEx.

Take your time trying each example out. *Experiment*. Tweak delays and other instructions to really understand how the program works. Use the `S` and `SS` commands to run the program step by step.

Try the exercises, but bear in mind that they are just a starting point. Come up with your own ideas to modify the example code. Do that a few times and you'll soon be ready to write your own ArdEx programs from scratch.

Hardware

You'll need an Arduino UNO with ArdEx installed. The Arduino environment shields you from the specifics of the hardware, making one Arduino much the same as another. However, ArdEx bypasses the Arduino environment, accessing the hardware directly. Consequently, ArdEx is only sure to work properly with an Arduino UNO (i.e. one based on an Atmel `ATmega328/P` microcontroller).

Many of the programs were developed with the *ThinkerShield* board installed. This board was developed for and sold by the N.S.W. Museum of Applied Arts and Sciences. It includes six LEDs, a button, a buzzer a potentiometer and a photoresistor and is an open hardware project published on GitHub. The ThinkerShield is a convenient companion to this book, but is not essential. The same programs can work by connecting the Arduino to the appropriate parts on a breadboard instead.

The ThinkerShield schematic is included in Appendix A.

Stepper motor TODO
7-segment LED TODO
74HC595 DIP 16 chip TODO
MCP9808 module TODO
resistors and breadboard TODO

Software

Loading the ArdEx software onto the Arduino.

Software for your computer

- Windows PC TODO
- Apple Mac TODO
- Linux TODO

1.1 First Program. What Else: Blinky!

Before getting into the details of programming, it is worth making sure that your equipment is working correctly. When trying out a new computer or language, the usual first program would print ”Hello”, or ”Hello, World!!” or such like. The equivalent for today’s microcontroller gadgets is to blink an LED.

Here is one way to do it with ArdEx:

```
0       MOV     #0x20,DDRB
1       XOR     #0x20,PORTB
2       WAIT    5000
3       JMP     1
```

Example 1.1: Blink LED

Type in the lines* as shown (including the ”step number” in the left column), then type `RUN<Enter>`. The LED will blink repeatedly: on for half a second, off for half a second. When you have seen enough press `<Ctrl>-C` and the ArdEx prompt will come up.

For now, don’t worry about what exactly is happening in the first line. The second line toggles the LED – if it’s on, it turns it off; if it’s off, it turns it on. Again, don’t worry too much for now exactly how that’s happening. The third line simply pauses for an interval counted in tenths of milliseconds; 5000 here is half a second. The fourth line jumps back to the second. And it repeats these few instructions until you interrupt it with `<Ctrl>-C`.

Try changing the delay. Just retype step 2 with a different interval, say 1000, and type `RUN` again. Now it blinks 5 times per second.

Before going on, a few notes on the ArdEx environment. The three columns are called the *step number*, the *opcode* and the *argument(s)*. The opcode and arguments together are the *instruction*. The step number is a value between 0 and 255, each representing a place where an instruction can be stored.

ArdEx allows you to include a comment after the arguments. Anything following a ‘;’ (semicolon) will be discarded. Indeed, if you type a semicolon, nothing more will appear on the screen until you hit `<Enter>`. Comments are intended to be used in files stored and edited on the computer before transferring to ArdEx.

You can start a program at any step number by putting that after the `RUN` command, e.g. `RUN 3` will start running at step 3.

After interrupting a program with `<Ctrl>-C` you can run it one step at a time using the `S` command. Just keep typing `S<Enter>` and you’ll see that the `XOR` instruction is the one that toggles the LED.

You can list the instructions stored in steps using the `LIST` command. Typing `LIST 0,9` will show you the instructions stored in the first ten steps.

If you don’t include a step number before an instruction, the instruction is executed immediately. If you type:

```
XOR #0x20,PORTB
```

*you don’t have to line things up perfectly; as long as you have a space or tab between columns will be fine

The LED will toggle as soon as you press `<Enter>`.

Look at Appendix C for the complete list of commands. They may not make much sense if this is all new to you.

Here's another version of ”Blinky”:

```
0       MOV     #0x20,DDRB
1       BIS     #0x20,PORTB
2       WAIT    5000
3       BIC     #0x20,PORTB
4       WAIT    5000
5       JMP     1
```

Example 1.2: Less symmetrical LED blinking

Here, the `BIS` instruction turns the LED on (”sets” it) and `BIC` turns it off (”clears” it). Now you can experiment with the two different `WAIT` intervals so the time spent on can be very different from the time spent off.

Blinky isn't the most exciting program in the world, but here are some things to try out and think about.

Exercises 1.1

1. The human eye can't detect flicker if something is turned on and off rapidly enough. Try testing your threshold by experimenting with the WAIT intervals. Which WAIT instruction determines how long the LED is off? Most people see flicker at an off period of 1/50th of a second, but not if the off period is 1/100th of a second or less. Where are you on the scale?

2. With the second example, hold the off period short enough that you can't see any flicker. Now vary the on period from about half that to about 5 times the off period. What do you notice about the LED?

1.2 Programming

Though it might seem trivial, there is a world of difference between the commands:

```
XOR #0x20,PORTB
```

and

```
1 XOR #0x20,PORTB
```

In the first command, you are executing an instruction, immediately toggling a bit in PORTB. In the second command, you are storing that instruction for *future* use. PORTB isn't affected at all until the program is run. The first command is “playing with a light switch”, the second is programming a computer to play with a light switch.

That is the final product of computer programming: a set of instructions to be carried out later on. The computer is wrapped around the very simple idea of repeating: