

Robotics Programming in C

COLLABORATORS

	<i>TITLE :</i> Robotics Programming in C		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		May 30, 2011	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Goals of this Tutorial	1
2	Why C?	2
2.1	C is an open standard	2
2.2	C is an all-purpose language	2
2.3	C is fast	2
2.4	C can do anything	3
3	Vex Programming Options	4
3.1	MCC18 and MPLAB	4
3.2	easyC	4
3.3	RobotC	4
3.4	SDCC and OpenVex	5
4	Getting Started with Vex Programming	6
4.1	Robot	6
4.2	Programming Computer	6
4.3	Programming Tool Chain	6
4.4	Programming Hardware	7
5	The Compilation Process	8
6	The Basics	9
6.1	Unix Basics	9
6.2	Vex Beginner Code.	9
6.3	Editing the Code	14
7	Basics of Embedded C Programming	15
8	Streams	16
8.1	What is a Stream?	16
8.2	Standard Streams	16

9	Standard stream I/O functions	18
10	Data types, Variables, and Expressions	20
11	Functions and Macros	22
12	Programming Exercises	24
12.1	Attaining Wisdom	24
12.2	Maximizing Available Resources	24
12.3	Basic Motor Control	25
12.4	Basic Sensor Input	25

List of Tables

2.1	Language Speed Comparison	3
9.1	printf() place-holders	19
10.1	C Data Types	20

Chapter 1

Goals of this Tutorial

This tutorial is meant as a quick-start guide to robotics programming in C, using the Vex robotics kit and SDCC. It assumes familiarity with a C-based language such as C, C++, Java, Perl, PHP. As such, it does not attempt to teach basic programming skills or C syntax. Instead, it focuses on highlighting the differences between C and similar languages, and topics important to embedded programming.

Much of this tutorial can also be followed using other compilers or robotics systems. The code examples provided here are specific to the OpenVex API (Application Programming Interface), but the concepts and exercises are common to all robotics programming. If you do not have a Vex robotics kit, but do have a Lego RCX or NXT, you could follow this tutorial without too much difficulty by adapting the example code for Lego systems using NQC (Not Quite C) with RCX, or NXC (Not eXactly C) with NXT. For NXT programming, I recommend purchasing "NXT Power Programming" by John Hansen. This book covers the NXC compiler and libraries in detail, as well as teaching some principles of robotics programming.

If you have never programmed before, you should first go through a basic textbook that explains C from the ground up, such as [The C/Unix Programmer's Guide](#). Get some practice using basic constructs such as if blocks, loops, and functions. For embedded programming, it is also important to learn number bases such as binary, octal, and hexadecimal. Once you've learned these basics, you should be able to complete this tutorial in a relatively short time.

Chapter 2

Why C?

2.1 C is an open standard

C is an industry standard language, that can be used to program all kinds of devices. In the embedded world, C is the only truly portable programming language. I.e., it's the only language that will allow you to use the code you write on many different devices.

Most other programming languages are proprietary, and can only be used to program one vendor's hardware. Examples include PIC BASIC, National Instruments Labview, and any type of assembly language. None of the code you write in PIC BASIC can be used to program a National Instruments Microcontroller Unit (MCU), or vice versa.

The motivation for hardware vendors to offer their own proprietary programming systems is simple. Learning to use a language and tools represents a significant investment of time. Once you invest your time into the learning curve for a proprietary system, you will have an incentive to continue using their hardware.

C code, on the other hand, is highly portable. Much of the C code you write is not hardware specific. For example, string manipulation functions, sensor input filters, high-level motor drive routines, etc. generally work the same way regardless of the hardware, and can be reused with different hardware and even different micro-controllers.

In short, proprietary languages benefit vendors, while open-standard languages benefit developers and their customers.

2.2 C is an all-purpose language

C can be used to program anything from embedded devices to major application programs. While no language can be the best for every purpose, it's nice to know that one language can serve *most* of your programming needs.

Hence, if you become proficient at C programming, you'll be well equipped to write fast programs for a wide variety of purposes. It's still a good idea to pick up a few other languages, particularly scripting languages that are well suited for quick automation of tasks where speed is not critical.

2.3 C is fast

The only way to write a program that will outrun a C program is by using assembly language, which is not portable. An assembly language is a human-readable version of a CPU's binary machine language, which is specific to each type of CPU. Hence, C allows you to write the fastest possible *portable* code.

The performance gap between various languages should not be underestimated. Compared to other purely compiled high-level languages such as C++, Pascal, Fortran, etc., the speed difference is generally marginal. Compared to interpreted languages, such as Perl, Visual BASIC, etc., the difference can be orders of magnitude.

The table below shows run times for a selection sort of 50,000 integers on a 2.0GHz Core Duo Mac running OS X Tiger.

Language	Run time (seconds)
C	4.01
C++	4.07
Java with JIT	6.14
Java without JIT	64.74
Perl	589

Table 2.1: Language Speed Comparison

All languages are written in what we call *source code*. The source code of true compiled languages like C and C++ are translated to the native machine language of the CPU before the program is executed. This results in the maximum possible run-time performance, and eliminates the need to have an interpreter on every machine that runs the program.

The source code of interpreted languages such as Perl, PHP, BASIC, Python, Ruby, etc. are not translated to machine language. Instead, the source code is parsed and executed by another program called an interpreter (often after being reduced to a simpler form for faster parsing). This results in a order of magnitude loss in run-time performance since the rough equivalent of compilation happens every time a statement in the program is executed. This isn't necessarily a problem, since interpreted languages are not meant for use in major applications where performance is critical. They are mostly used for small, special-purpose "script" programs. Another drawback of interpreted languages is the need to have an interpreter on every machine that runs a program. A compiler need only be present on the machine that compiles the program. Once compiled, the executable program can be distributed to other machines and executed directly by the hardware.

Java and .NET-based languages are in between true compiled languages and true interpreted languages. They are compiled to a non-native byte code that resembles machine language, but is not understood by the CPU. This byte code must be interpreted by the "Virtual Machine", which works much like any other interpreter, but is much faster since the byte code is much simpler than the source code. The Java JIT (just-in-time) compiler translates the byte code to native machine code while the program is running, so that the second time a statement is executed, it runs at native compiled speed. This allows Java programs to run nearly as fast as an ahead-of-time compiled language, but it still requires the presence of the Java virtual machine. When Java is used in embedded systems, it runs on a stripped-down version of the Java virtual machine which generally does not provide just-in-time compilation.

Other true compiled languages such as Pascal and Fortran will show performance comparable to C and C++. The .NET platform is a "virtual machine" (basically a fancy interpreter) whose compiled languages will perform comparably to Java. Fully interpreted languages like PHP, Ruby, etc. can be expected to perform comparably to Perl.

2.4 C can do anything

The C language is often referred to as a "high level assembler". What this means is, it offers the ease of programming and portability of a high-level language, along with the low-level capabilities and speed of assembly language. There is really nothing you can't do in C. 99% of the Unix operating system (including most of the device driver code) is written in C. Entire embedded applications, including device drivers, are often written in C. The OpenVex firmware for SDCC is 100% C, and the MCC18 version contains only a tiny amount of assembly language. Occasionally, programmers may choose to sacrifice portability and write hand-optimized assembly language to get a slight speed improvement for a highly critical section of code.

Other "higher level" languages often aim to reduce the initial learning curve by making common tasks easier. This is an attractive selling point to inexperienced programmers, most of whom eventually learn that they can't implement their whole program in the language because of limitations in performance or capabilities. This generally leads to software written partly in a proprietary language, and partly in C. I.e., choosing the proprietary language often backfires.

Chapter 3

Vex Programming Options

The standard options available for programming the Vex are all based on C.

3.1 MCC18 and MPLAB

The Vex uses a Microchip PIC 18F8520 micro-controller. Originally, the only supported programming option for Vex was Microchip's MCC18 compiler, with the MPLAB integrated development environment. While there are other commercial compilers for the PIC processor, Vex only offered sample firmware written for MCC18/MPLAB. Since Vex does not release full hardware specs on the Vex controller, it is difficult to create firmware with other compilers.

Unfortunately, the MPLAB environment and the Vex default code are not at all friendly to inexperienced programmers. The MPLAB environment requires a lot of setup, and the Vex default code is difficult to understand, even for experienced programmers.

3.2 easyC

It was quickly recognized that MPLAB is too difficult for the inexperienced to tackle on their own, so Intelitek created the easyC environment, a drag-and-drop C programming system based on MCC18 and the WPI library. easyC lets programmers choose simple, abstract program components from a menu, to perform tasks like starting and stopping motors, checking sensor values, etc.

While easyC is very easy to learn, it is also very easy to outgrow. Once programmers get the basic idea about how to put together a program for the Vex, the drag-and-drop interface becomes a hindrance.

3.3 RobotC

RobotC is a more flexible system where you can code your programs in a text editor, while still using abstract functions comparable to easyC's WPIlib.

Unfortunately, RobotC does not use a true compiler. Instead of compiling the C code to native PIC machine language, it compiles it to a proprietary byte code which runs on a virtual machine that in turn runs on the Vex controller. In other words, the micro-controller doesn't run your program directly. Instead, it runs an interpreter (virtual machine) that runs your program. This virtual machine reduces the speed of RobotC programs much like a Java VM with no JIT (Just-in-time compiler). While it's probably fast enough for most purposes, it may limit your capabilities at some point, leaving you no alternative but to rewrite the program in a truly compiled language.

3.4 SDCC and OpenVex

This is the only free system for programming the Vex, and the only system that works on non-Windows machines. It has been tested on FreeBSD, Linux, Mac, and Windows under Cygwin.

SDCC (Small Device C Compiler) is an open-source compiler based on GCC (the GNU C Compiler), which supports several different microcontrollers. The compiler and standard libraries for PIC processors are very similar to MCC18, but not 100% compatible.

OpenVex is an open source API library comparable to WPILib and RobotC's library. It can be compiled with either SDCC and MCC18.

Using SDCC, you can program the Vex from just about any platform using your favorite text editor. OpenVex provides abstract function calls much like easyC and RobotC, so that inexperienced programmer's can learn to program the Vex quickly using simple function calls to control motors and read sensors. OpenVex is also 100% open source, so more advanced programmers can dig into the library code and make more sophisticated programs.

Chapter 4

Getting Started with Vex Programming

This chapter outlines what you will need to get started with Embedded C Programming on the Vex

4.1 Robot

Start with a basic robot similar the squarebot described in the Vex Inventor's Guide. You can be creative and design a robot according to your own interests. The following are recommended basic specs for a starter robot:

- A drive wheel on each side driven by its own motor.
- A bumper switch on the front of the vehicle.
- At least one analog sensor, such as a light sensor.
- An implement controlled by a motor or servo.

4.2 Programming Computer

To program your Vex using SDCC, you can use any computer capable of running SDCC. This could be a PC running some form of Unix, a PPC or Intel Mac, or a Windows machine. If you're using Windows, Cygwin is recommended for SDCC and OpenVex programming. This will make the experience match the instructions in this tutorial.

I recommend Mac or some form of Unix such as FreeBSD or Linux. These systems are unaffected by the malware such as viruses and spyware that plague Windows machines, and are also much faster and more reliable than Windows on the same machine. By avoiding Windows you can spend most of your time doing robotics programming instead of installing updates and fixing problems.

The OpenVex library and roboctl communication suite are developed on FreeBSD and Mac OS X, and fairly well tested under Debian-based Linux systems (e.g. Debian, Ubuntu) and Cygwin. All of these platforms have convenient ports/package systems for installing the tools you will need. (FreeBSD ports, MacPorts, Debian Packages, and Cygwin Packages)

4.3 Programming Tool Chain

Your computer will need the following software installed:

- The OpenVex library: Available at <http://personalpages.tds.net/~jwbacon/vex.html>.
 - SDCC: Available in most ports/packages systems.
-

- Robotcl: Available as an official FreeBSD port and MacPort. A Gentoo port is also available, but check to make sure it's the latest version. For other systems, the source code can be obtained at <http://personalpages.tds.net/~jwbacon/vex.html>
- A programming editor: APE (Another Programmer's Editor) has built-in support for programming the Vex using SDCC and robotcl under FreeBSD, Mac OS X, Linux, and Cygwin. APE is also available at <http://personalpages.tds.net/~jwbacon/vex.html>.

Note Included with the library are "robotize" scripts to automatically install SDCC, robotcl, and APE on FreeBSD, Mac OS X, Debian, and Cygwin systems.

4.4 Programming Hardware

You will need the Vex programming hardware kit, available at <http://www.vexrobotics.com>. You do not need to buy any of the software kits: the hardware kit can be purchased separately.

Chapter 5

The Compilation Process

Compilation is the process of converting C, C++, Pascal, Fortran, or other high-level language code to the binary machine instructions that drive the CPU.

Machine instructions are very primitive, performing only the simplest operations such as adding two integers, copying a character or number from one memory location to another, etc. Each type of CPU has a distinct machine language. The machine language of Intel and AMD processors are almost identical, which is why a PC can use either one to run Windows software. A PowerPC CPU has a completely different machine language than Intel and AMD, however. These factors make it difficult and disadvantageous to program in machine language, or in assembly language which is simply a human-readable representation of machine language. Coding in machine or assembly language is laborious due to its primitive nature, and if you switch to a different CPU architecture, you would have to rewrite all your code.

The portability of C and other HLLs depends on the availability of compilers for each type of CPU. C compilers are available for virtually every type of CPU, hence C is one of the most portable languages around.

The compilation of C involves three steps:

1. Preprocessing performs a series of text insertions and substitutions according to the *directives* you have inserted into your program. For example, an `#include` directive inserts another file into the program, and a `#define` directive creates a named constant such as `PI`, which can be used to make the program more readable and easier to maintain. Directives are described in more detail later. The preprocessor does not overwrite your source file, but passes edited code to the compilation phase.
2. The preprocessor output is passed on to the *compiler*, which translates the portable C code to the CPU-specific machine code.
3. The C language is actually quite simple and minimal, but extensible. As a result, much of C programming involves using *libraries*, which are pre-written, pre-compiled routines for common tasks such as input and output, common math and trigonometry computations, and so on. Much of what goes into the executable file created by the compiler actually comes from the standard libraries included with all C compilers. In many cases, the code you wrote may be a small fraction of the actual program.

In addition, your own C code may be in more than one source file. You will compile these source files to machine language separately to produce *object modules*. Object modules are files containing machine code from a single source file, and usually have a `.o` extension on their name. You then combine your object modules with machine language from the libraries to produce the complete executable program.

This leads us to the final stage of compilation: *linking*. The linker combines the machine language from one or more object modules with machine code retrieved from the libraries to produce the final *executable* file.

Chapter 6

The Basics

6.1 Unix Basics

OpenVex was built on a Unix system called FreeBSD, but can be used on virtually any hardware and operating system. Unix was once a trademark for a specific operating system created at Bell Labs, but the term is now used to refer to a set of standards that almost all modern operating systems adhere to (Windows being the only mainstream exception). FreeBSD is one such operating system, as are Linux, Mac OS X, Sun Solaris, IBM AIX, and many others. The Unix standard makes our lives easier and promotes competition in the computer industry by allowing us to use the same user interface (UI) and application program interface (API) on a variety of computer hardware and operating systems.

Windows users can use the Cygwin system to simulate Unix. Cygwin is not an emulation layer, as many people think: Rather, it is a set of common Unix tools ported to run directly under Windows. Hence, the Cygwin tools run at the same speed and in the same environment as other programs on the Windows system.

This tutorial will assume that users are working in the Unix shell environment, since this is possible on all operating systems, and limiting the tutorial to one environment keeps things simpler. Whether you're using FreeBSD, Linux, Mac, or Cygwin, the instructions here are the same.

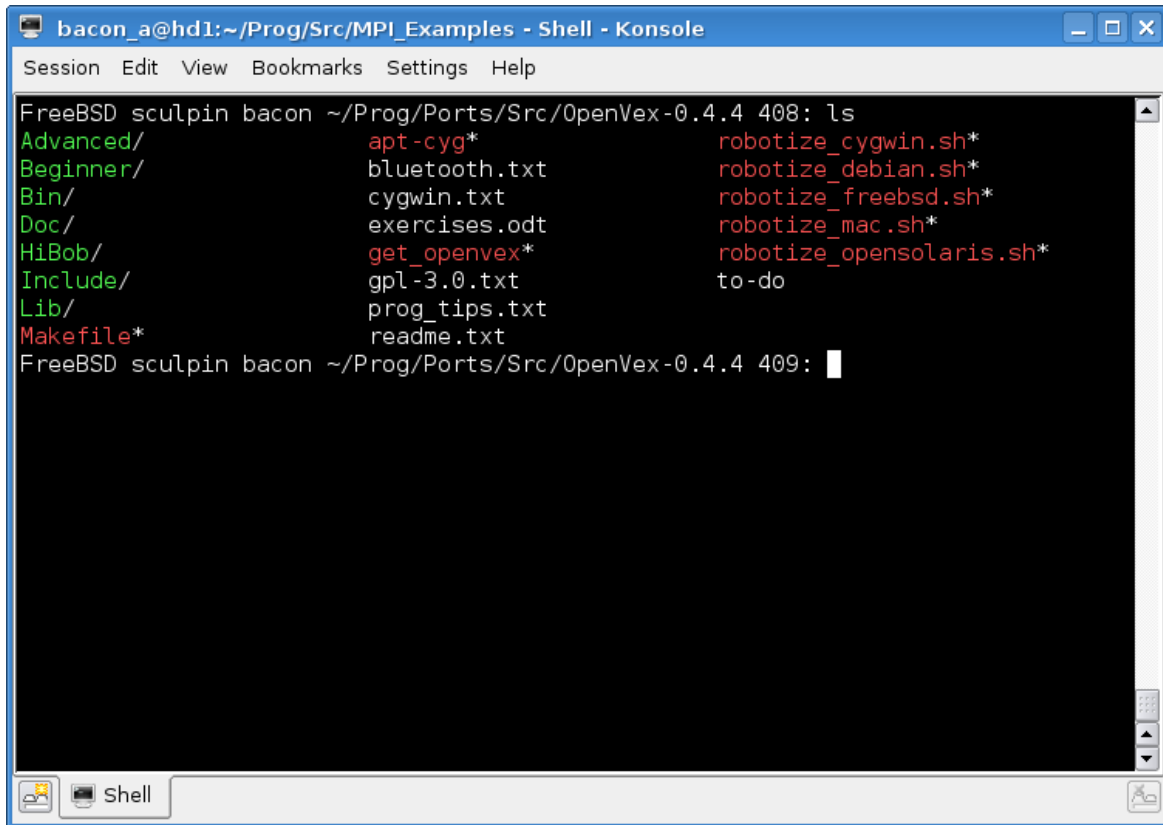
To program with OpenVex, you will need to know a few basic Unix commands. You don't have to be a Unix guru, but you should be comfortable with moving around the directories (folders) from the command line interface, editing files, and a few other basics.

These basics can be learned in less than an hour from one of the many good tutorials available on the WEB. You can find them by entering "c tutorial" in your favorite search engine. Of the few I examined [this one](#) seems to be the best suited for our needs here.

6.2 Vex Beginner Code.

Once you have installed the required software on your programming computer, do the following:

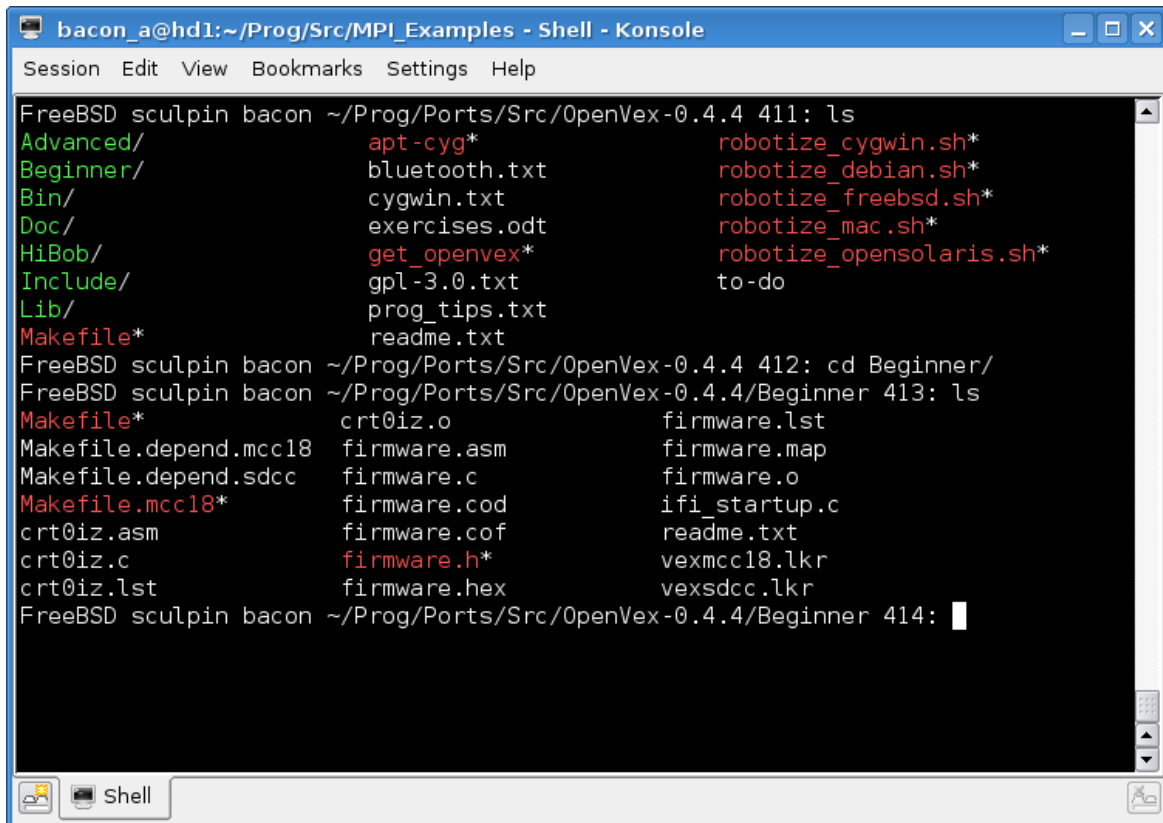
1. Open a terminal window (a Cygwin window if using Windows). You will see a prompt in the terminal window, indicating that the shell is waiting for you to enter a command. Commands that you should type below are indicated by the prompt "shell: ".
 2. shell: cd OpenVex-0.4.4
 3. shell: ls You should see a file listing that looks much like the image below. The window borders and colorization of the filenames will depend on which operating system you use, but the text should be the same. This snapshot was taken on FreeBSD with the KDE desktop.
-



```
bacon_a@hd1:~/Prog/Src/MPI_Examples - Shell - Konsole
Session Edit View Bookmarks Settings Help
FreeBSD sculpin bacon ~/Prog/Ports/Src/OpenVex-0.4.4 408: ls
Advanced/      apt-cyg*      robotize_cygwin.sh*
Beginner/      bluetooth.txt robotize_debian.sh*
Bin/           cygwin.txt    robotize_freebsd.sh*
Doc/           exercises.odt robotize_mac.sh*
HiBob/        get_openvex*  robotize_opensolaris.sh*
Include/       gpl-3.0.txt   to-do
Lib/          prog_tips.txt
Makefile*     readme.txt
FreeBSD sculpin bacon ~/Prog/Ports/Src/OpenVex-0.4.4 409: █
```

4. shell: cd Beginner

5. shell: ls



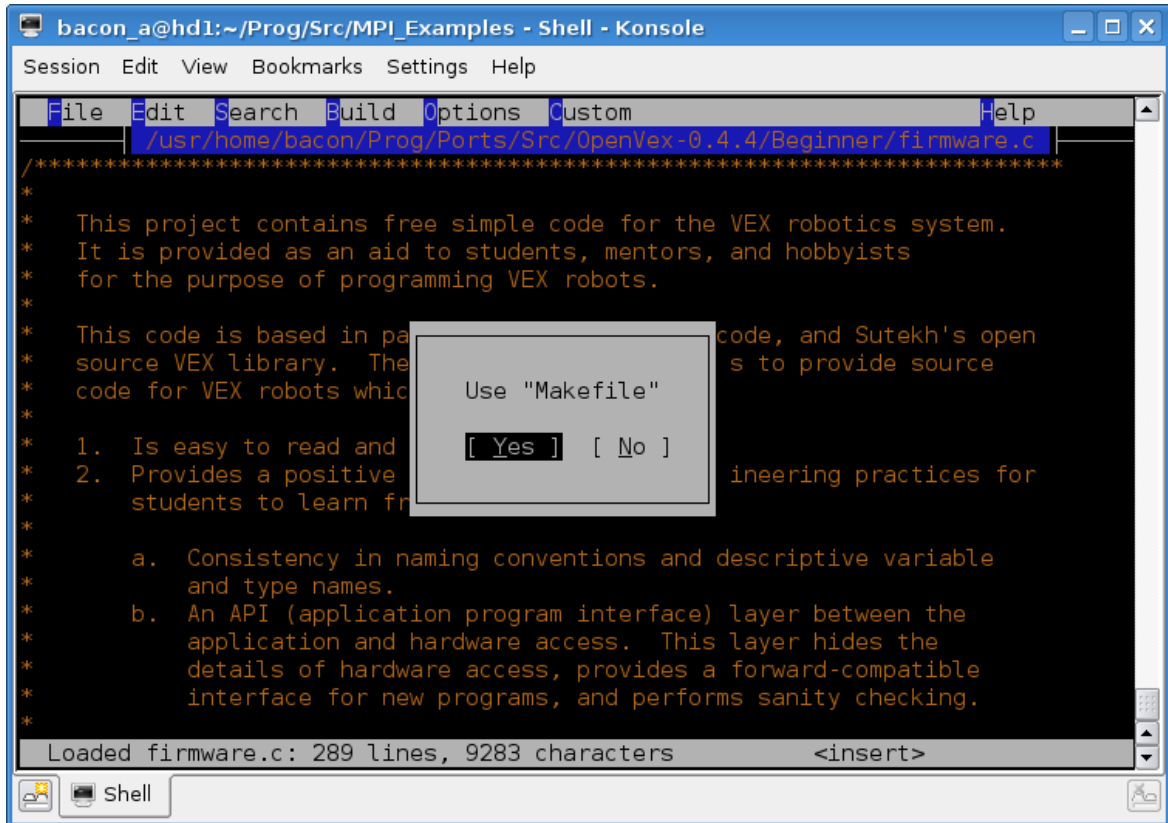
```
bacon_a@hd1:~/Prog/Src/MPI_Examples - Shell - Konsole
Session Edit View Bookmarks Settings Help
FreeBSD sculpin bacon ~/Prog/Ports/Src/OpenVex-0.4.4 411: ls
Advanced/      apt-cyg*      robotize_cygwin.sh*
Beginner/      bluetooth.txt robotize_debian.sh*
Bin/           cygwin.txt    robotize_freebsd.sh*
Doc/           exercises.odt robotize_mac.sh*
HiBob/        get_openvex*  robotize_opensolaris.sh*
Include/       gpl-3.0.txt   to-do
Lib/          prog_tips.txt
Makefile*     readme.txt
FreeBSD sculpin bacon ~/Prog/Ports/Src/OpenVex-0.4.4 412: cd Beginner/
FreeBSD sculpin bacon ~/Prog/Ports/Src/OpenVex-0.4.4/Beginner 413: ls
Makefile*     crt0iz.o      firmware.lst
Makefile.depend.mcc18  firmware.asm  firmware.map
Makefile.depend.sdcc  firmware.c    firmware.o
Makefile.mcc18*      firmware.cod  ifi_startup.c
crt0iz.asm          firmware.cof  readme.txt
crt0iz.c            firmware.h*   vexmcc18.lkr
crt0iz.lst          firmware.hex  vexsdcc.lkr
FreeBSD sculpin bacon ~/Prog/Ports/Src/OpenVex-0.4.4/Beginner 414: █
```

6. Now it's time to edit the program. You can use your favorite editor and then compile and upload the program from the

command line by running "make install".

This tutorial uses APE (Another Programmer's Editor), which is installed by the robotize scripts included with OpenVex. APE allows you to build and upload the program using the menus and/or hot keys within the editor.

shell: ape firmware.c



Select "YES" when APE asks if it should use Makefile. Then accept the default parameters that pop up in the next window.

The Makefile is a script-like file that contains the rules for building the executable file for the Vex controller from the C source code. The process is somewhat complicated, so for now you can simply use the provided Makefile, and focus just on editing the main source file. This is all you should need to do for most Vex programming projects, but you're free to explore and edit the Makefile and library code when you feel adventurous.

```

bacon_a@hdl:~/Prog/Src/MPI_Examples - Shell - Konsole
Session Edit View Bookmarks Settings Help
File Edit Search Build Options Custom Help
/usr/home/bacon/Prog/Ports/Src/OpenVex-0.4.4/Beginner/firmware.c
/*****
*
* This project contains free simple code for the VEX robotics system.
* It
* for
*
* Makefile:      Makefile
* Executable:    firmware.hex
* Directory:     /usr/home/bacon/Prog/Ports/Src/OpenVex-0
* Make Arguments:
* Run prefix:    time
* Run Arguments:
* 1.
* 2.
*
* Enter name of makefile in the current directory.
* Move: (Arrows or TAB), [ OK ] (Enter), [ Cancel ] (Esc)
*
*
* application and hardware access. This layer hides the
* details of hardware access, provides a forward-compatible
* interface for new programs, and performs sanity checking.
*
Loaded firmware.c: 289 lines, 9283 characters <insert>
Shell

```

7. To build and upload the firmware, open the Build menu using Alt+b (hold Alt while pressing b) or Esc-b (press and release Esc and then press b within 1 second). Then select Install from the Build menu.

```

bacon_a@hdl:~/Prog/Src/MPI_Examples - Shell - Konsole
Session Edit View Bookmarks Settings Help
File Edit Search Build Options Custom Help
/usr/home/bacon/Prog/Ports/Src/OpenVex-0.4.4/Beginner/firmware.c
void main(void)
{
    /*
    * Gory hardware
    * to a reasonable
    * Must be done
    */
    vex_init();

    /* Initialize port
    custom_init();

    /*
    * Main loop.
    * that new data
    */
    while (TRUE)
    {
        Build and Run (Esc-r or F5)
        Compile to Object (Esc-i or F6)
        Syntax check (Esc-y or F8)
        Build executable (Esc-x or F7)
        Clean (Esc-c)
        Install
        Translate to Assembly language
        View Preprocessor output

        Goto next error (Esc-n)
        View compiler errors
        Run Debugger
        View function call trace
        View execution profile

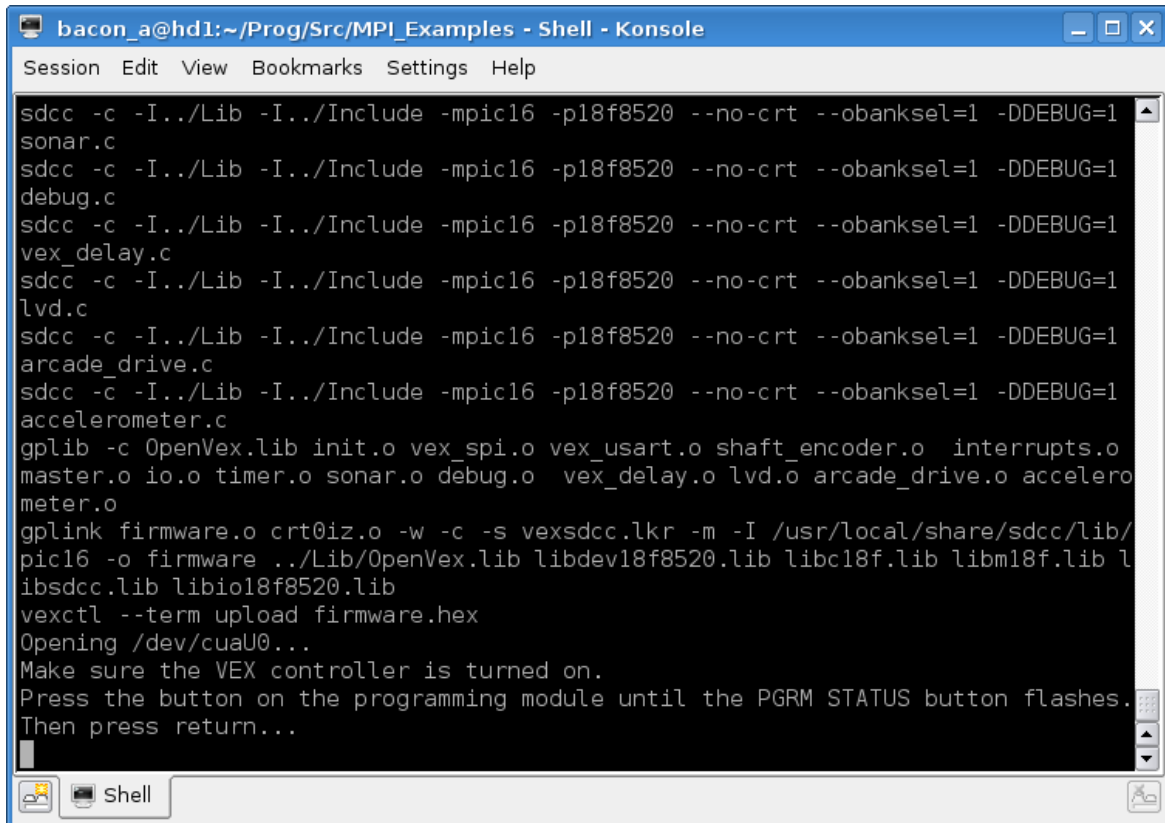
        Select Makefile
        Set Execute command

Loaded firmware.c: 289 lines, 9283 characters <insert> 56 1
Shell

```

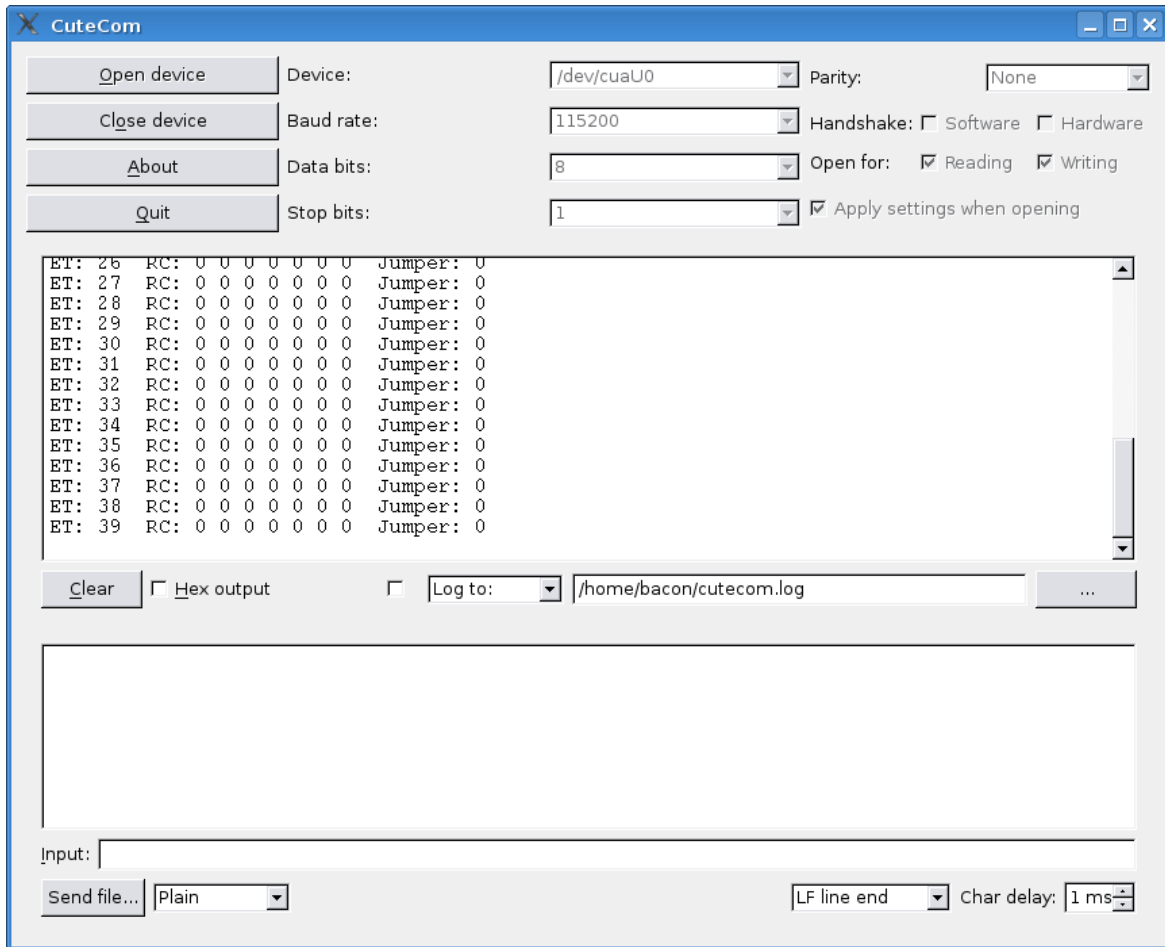
8. The firmware should build, and the Makefile should automatically run *vexctl*, the program that uploads HEX files to the

controller. The `vexctl` program will remind you to press the button on the programming dongle (the orange box in the middle of the cable connecting your computer's USB port to the serial port on the Vex controller).



```
bacon_a@hdl:~/Prog/Src/MPI_Examples - Shell - Konsole
Session Edit View Bookmarks Settings Help
sdcc -c -I../Lib -I../Include -mpic16 -p18f8520 --no-crt --obanksel=1 -DDEBUG=1
sonar.c
sdcc -c -I../Lib -I../Include -mpic16 -p18f8520 --no-crt --obanksel=1 -DDEBUG=1
debug.c
sdcc -c -I../Lib -I../Include -mpic16 -p18f8520 --no-crt --obanksel=1 -DDEBUG=1
vex_delay.c
sdcc -c -I../Lib -I../Include -mpic16 -p18f8520 --no-crt --obanksel=1 -DDEBUG=1
lvd.c
sdcc -c -I../Lib -I../Include -mpic16 -p18f8520 --no-crt --obanksel=1 -DDEBUG=1
arcade_drive.c
sdcc -c -I../Lib -I../Include -mpic16 -p18f8520 --no-crt --obanksel=1 -DDEBUG=1
accelerometer.c
gplib -c OpenVex.lib init.o vex_spi.o vex_usart.o shaft_encoder.o interrupts.o
master.o io.o timer.o sonar.o debug.o vex_delay.o lvd.o arcade_drive.o accelero
meter.o
gplink firmware.o crt0iz.o -w -c -s vexsdcc.lkr -m -I /usr/local/share/sdcc/lib/
pic16 -o firmware ../Lib/OpenVex.lib libdev18f8520.lib libc18f.lib libm18f.lib l
ibsdcc.lib libio18f8520.lib
vexctl --term upload firmware.hex
Opening /dev/cuaU0...
Make sure the VEX controller is turned on.
Press the button on the programming module until the PGRM STATUS button flashes.
Then press return...
```

Press the dongle button, and then press return on your computer to begin the upload. The firmware will begin to run as soon as the upload completes. At this point, the Makefile will run `cutecom` (on real Unix systems) or `Bray++ Terminal` (on Cygwin) to display output from the Vex controller. The output messages from the controller are sent to your computer through the same cable used to upload the firmware. Make sure the serial port settings (baud, data, and stop) match those shown below, and then open the device. After opening the device, you should see the output from the firmware appear on the screen.



6.3 Editing the Code

Now go to APE's File menu, and select open. Use the arrow keys to select `firmware.h` and open it. You now have two files open in APE. You can toggle between them using `Ctrl+t`.

Check the values of `LEFT_DRIVE_PORT` and `RIGHT_DRIVE_PORT` in `firmware.h`. Make sure your robot's drive motors are plugged into the same port numbers set in `firmware.h`. If so, you should now be able to drive your robot using the firmware you just compiled and installed!

From here on, programming will be a process of exploration and experimentation. Try modifying your robot and the code to add additional drive motors, implement motors and/or servos, and simple sensors such as button sensors or light sensors. Study the source code and the [OpenVex API Documentation](#) to see what functions are available and how they are called.

When you feel you've mastered the Beginner code, go to the *Advanced* directory, and begin exploring the code there. The advanced code contains examples of autonomous operation and many different sensors.

Chapter 7

Basics of Embedded C Programming

The following chapters cover the basics of embedded programming in C. As mentioned at the start of this tutorial, it is assumed that you have a working knowledge of C or a similar language.

To become proficient in any language requires more depth than any tutorial can provide, so readers are urged to consult a good textbook on the subject. Any book on C (not C++) will help prepare you for programming in SDCC. If you don't already have one, [The C/Unix Programmer's Guide](#) covers the C language, as well as background information on number systems, Unix basics, and standard C library functions, which are essential for programming with SDCC.

Chapter 8

Streams

8.1 What is a Stream?

Most applications written in C perform I/O operations through a facility known as a stream. Streams do two things for the programmer:

1. Hide the complexity of and differences between various I/O devices, allowing us to view all input and output as simple streams of characters. Our programs send characters to the stream using simple statements such as:

```
putchar('a');
```

and read characters from input streams using statements like:

```
ch = getchar();
```

These stream functions take care of all the complex details of reading or writing the file or I/O device for us.

2. Improve efficiency of I/O by buffering data in memory. Initiating a transaction with an I/O device often takes many times longer than reading or writing the actual data. Hence, we can improve I/O performance by reading and writing data in fewer and larger transactions. Streams accomplish this by saving many characters of data in memory writing all of them in one large operation. I.e., calling `putchar()` usually only places characters into a buffer. When the buffer is full, `putchar()` calls another function to empty it to the actual output device. Similarly, `getchar()` initially calls a function to read in a block of characters to a memory buffer, and then subsequent calls simply take them from the buffer until it is empty.

8.2 Standard Streams

On typical computers such as Unix, Mac, and Windows, every program has three standard streams available at all times, and can open more streams as needed to access files or I/O devices:

- Standard input (`stdin`): Connected to the keyboard by default. Can be *redirected* to take input from a file or another input device if desired.
- Standard output (`stdout`): Connected to the screen by default. Used for normal program output. Can be *redirected* to send output to a file or another output device if desired.
- Standard error (`stderr`): Connected to the screen by default. Used for error messages. Can be *redirected* to send output to a file or another output device if desired.

Stdout and stderr can each be directed to different destinations to separate program error messages from normal output.

On embedded systems, input and output is typically much more limited than on a PC or larger computer. On the Vex there is no stderr stream. The stdin and stdout streams are connected to the same simple serial interface used to upload code to the Vex from the IFI loader or vexctl.

This serial interface is known as a USART (Universal Synchronous/Asynchronous Receiver/Transmitter. The term *serial* means that it transmits all the bits over a single wire, one after another. (A parallel interface, such as an old parallel printer cable or ATA disk cable, typically transmits 8, 16, or 32 bits at the same time over separate (parallel) wires.)

In order to display output from the Vex controller, your PC must be connected to the Vex serial port, and be running a terminal program to read and display the character data the Vex is sending. There is a simple terminal included in the IFI loader. However, I recommend the Bray++ terminal for Windows users, and Cutecom for Mac and other Unix users.

Chapter 9

Standard stream I/O functions

The standard functions for I/O with `stdin` and `stdout` are as follows:

The rule for output is always use the simplest method. For example, use `putchar('a')` instead of `puts("a")`, and `puts("Hi, Bob")` instead of `printf("Hi, Bob\n")`. Using `printf()` to print a simple string that has no place-holders wastes CPU time, since `printf()` combs the string for place-holders as it prints. This may or may not be an issue for a particular program, but it's better to form good habits. If you can eliminate complex library functions such as `printf()` from your code altogether, you'll end up with a smaller executable file.

- `putchar()` - send the character *ch* to `stdout`.

```
putchar('a');
```

The argument to `putchar` can be a `char` or `int` variable (discussed later) or a *character constant*, which is any visible character or *escape sequence* between SINGLE quotes. The most common escape sequences are `'\n'` (newline) and `'\t'` (tab).

- `puts(str)` - sends a simple string (array of characters) `str` to `stdout`. The `puts()` function simply calls `putchar()` for each character in the string. The argument to `puts()` can be a character array (discussed later) or a string constant, which is any set of characters and/or escape sequences between DOUBLE quotes.

```
puts("Hi, Bob.");
```

Note: `puts` outputs a newline (`'\n'`) after the string, so you do not need to place one in the string. If you don't want a newline printed after the string, use `fputs(str, stdout)`. The following is equivalent to the above:

```
fputs("Hi, Bob\n", stdout);
```

- `printf(format-string, additional arguments)` - prints any type of data to `stdout`.

The format string provides an overview of what the entire output will look like. It contains constant character data as would be sent to `puts()`, plus *place-holders* for the additional arguments that follow.

The value of the first argument after the format string is displayed where the first place-holder appears, and so on.

```
printf("%d squared is %d\n", c, c*c);
```

Place-holders begin with a `'%'`. If you actually want a `'%'` to be printed by a `printf()` statement, use `"%%"`. There is a different place-holders for each type in the C language. (Data types are explained in the next section.) Below is a list of the most useful ones in Vex programming:

Place-holder	Data type
%d	int
%ld	long
%p	memory address
%f	float
%c	character

Table 9.1: printf() place-holders

Chapter 10

Data types, Variables, and Expressions

The C language supports all the data types offered by typical computer hardware. Specifically, there are signed and unsigned integer types of size 8, 16, 32, and 64 bits, and floating point types, usually 32 and 64 bits.

The table below enumerates the C data types along with their usual sizes and range. Unlike Java, C data types are allowed to vary in size across different hardware platforms. In particular, the *long* type may be 64 bits on some systems instead of 32.

Type	Size in bits	Signed range	Unsigned Range
char	8	-128 to +127	0 to 255
short	16	-32,768 to +32,767	0 to 65,536
int	platform-dependent	platform-dependent	platform-dependent
long	32 (or 64)	-2,147,483,648 to +2,147,483,647	0 to 4,294,967,295
long long	64 (or 128)	-9.223 x 10 ¹⁸ to +9.223 x 10 ¹⁸	0 to 1.845 x 10 ¹⁹
float	32	+/- 10 ³⁸	Does not apply
double	64	+/- 10 ³⁰⁸	Does not apply

Table 10.1: C Data Types

The *int* type is considered an alias for short or long, and you should never make any assumptions about its size when writing code. The size of *int* is generally the native word size of the underlying processor, which could be 16 bits on a small processor (such as the PIC processor in the Vex controller) or 32 bits on a typical PC. Use *int* only when you don't care whether the variable is a short or long.

The PIC processor in the Vex controller is a 16-bit processor. This means that the largest piece of data that it can process in a single instruction is 16 bits. As a result, using *long* or *long long* in your programs will make them slower, since the PIC must deal with these types in multiple pieces.

Floating point operations are slow and imprecise, and cause the CPU to draw more power and generate more heat. Generally, they should be avoided in embedded systems. If the underlying processor supports floating point, each operation (e.g. addition, subtraction) will take roughly 3 times longer than the same operation on integers.

Floating point types (*float* and *double*) are often not supported at all by the underlying hardware. If this is the case (as it is for the PIC processor), then floating point operations must be handled by software. This will slow your program down by a factor of 20 or more, and also result in a much larger program since it must include another function for each floating point operation.

In addition, the perceived need to work with fractional values is often a false assumption. Any computation can be achieved using integers. In some cases, it can be difficult, but achievable. In most cases, it is a simple matter of choosing the right units. For example, storing monetary values as pennies instead of dollars eliminates the need for fractional values.

Ordering of multiplication and division operations can also be important when working with integer arithmetic. Consider the code snippet below:

```
short  a = 1000,  
      b = 500,  
      c = 50,  
      d;  
  
d = a * b / c; /* Integer overflow */  
e = c / a * b; /* Truncation causes unexpected results */
```

When computing the expression $a * b / c$, the program first computes $a * b$, which is 500,000. This is beyond the range of a short, so the ultimate result assigned to d will be invalid. C programs do not terminate with an error when integer overflow occurs, so you have to check the validity of your output to detect errors like this one. This problem can be easily avoided by rearranging the expression as follows:

```
d = a / c * b;
```

The result of a / c is 20, which is then multiplied by 500 to yield 10,000. All intermediate results and the final result are therefore within the range of a short.

The next expression has a different problem. When computing c / a , the program uses integer division (recall this from 2nd grade or so, where we end up with an integer quotient and a remainder). The result of c / a is 0, and the remainder is discarded. (It can be computed with the `%` operator if needed.) The variable e is then assigned $0 * 500$, which is 0, probably not what we wanted. Again, we can fix this with a simple rearrangement:

```
e = b * c / a;
```

By multiplying first, we avoid dividing a smaller number by a larger one. The expression $b * c$ is 25,000, which is then divided by 1000 to yield 25.

Chapter 11

Functions and Macros

In embedded systems, speed and memory use can be critical. A typical PC as of this writing has a gigabyte or more of RAM, and runs a billion or more instructions per second. The PIC processor in the Vex controller has 30 kilobytes of flash program memory, 2 kilobytes of data memory (yes, that's correct), and runs about 10 million instructions per second.

Your program code must fit into the 30k of flash memory, and all of your variables plus additional things like function return addresses must fit into the 2k of data RAM.

For this reason, you will want to avoid writing programs with too many levels of function calls (e.g. main calls a, which calls b, which calls c, ... on down to z before returning all the way back to main). I.e., you want to avoid "over-abstracting" or "over-factoring" your code. Each function call takes up more space on the *stack* for each argument passed to the function and the return address that gets the program back to the caller. This could cause your program to run out of stack space (known as a stack overflow), or run out of memory in general.

It also generates more instructions to implement argument passing, and branching to and from the function, which could result in running out of code space. (The quickest way to run out of code space is by using floating point in your program.) On non-embedded systems, we generally don't worry about these issues at all, since the overhead they generate is trivial compared to available resources. Embedded systems require some additional care, however.

The way *not* to remedy this is by writing spaghetti code. Do not write a huge main() or other huge functions that serve multiple purposes. Each function should serve exactly one purpose. Also do not use global variables unless they are really necessary. Doing these things will make your program very difficult to debug.

A better way to eliminate function call overhead is by using macros instead of functions. (Using inline functions would have the same effect, but most embedded compilers do not support inline.) By using a macro, we still abstract out common complex operations so that they can be represented by a single line of code, making the program much easier to follow. But the macro eliminates the overhead of a function call by *replacing* the call with the code, rather than a branch to and from the code. Macros are best suited for very small tasks. For example:

```
short  abs(short a)
{
    if ( a >= 0 )
        return a;
    else
        return -a;
}
```

This function has two problems:

1. The overhead of calling it exceeds the useful work it performs.
2. It only works for shorts. We would need additional identical functions for char, int, long, and long long.

Simple computations such as absolute value, min, max, etc. are great candidates for macros. The macro version of abs() would be defined as follows:

```
#define ABS(a) ((a) >= 0 ? (a) : -(a))
```

When the macro is used in a program, the preprocessor replaces the call by the macro body. For example, the statement:

```
y = b * ABS(x+2) / 5;
```

is replaced by

```
y = b * ((x+2) >= 0 ? (x+2) : -(x+2)) / 5;
```

This statement is then compiled into machine code in the compilation phase. Note that without the extensive parentheses in the macro, this statement would not work as expected. In particular, the - sign in $-x+2$ would only apply to x , vs $-(x+2)$. Without the outer parentheses, the division by 5 would take precedence over the conditional operator ($?:$), and therefore would only apply to the else clause, $-(x+2)$.

Also note that the macro name is all upper-case letters. This is a convention used in C programming to alert programmer's that they are using a macro, and not a function. This is important to know in order to avoid side effects. For example,

```
y = ABS(x++);
```

will not work as expected. If you examine the preprocessor output:

```
y = ((x++) >= 0 ? (x++) : -(x++));
```

you see that x gets incremented twice, regardless of the outcome of the comparison! This is not what we intended, and would not occur when using an `abs()` function.

Chapter 12

Programming Exercises

12.1 Attaining Wisdom

Wisdom comes from doing, not from watching. It's important to study the materials in previous sections and in books, and it's helpful to have a good teacher explain things. All this only serves to get you started, though. Most of your real learning happens as you struggle with getting the programs to work.

Don't let your friends or mentors give you the answers to questions. If they try to write code for you or tell you exactly what to write, don't allow it. Insist on getting only hints, so you can do the work yourself and gain a *real* understanding of robotics programming.

Watching someone else write code won't make you a programmer any more than watching Dan Jansen will make you a speed skater. It can point you down the right path, but in programming just as in sports, you have to put in the long hours of practice to hone your strength and skills. Letting someone else do it for you is faster and easier of course, but it robs you of the opportunity to develop your own skill and independence.

12.2 Maximizing Available Resources

Humans are better at some tasks, and machines are better at others. For example, the human brain (and the cockroach brain, for that matter) is a much better image processor than any machine available today. Hence, for tasks involving complex visual input, a human driver can generally perform much better than an autonomous robot. For more mundane tasks, the speed and precision of the computer will usually win out, so these should generally be automated.

The real power of a robot can be realized by utilizing both the human brain and senses, together with the speed and precision of the computer. For example, a human operator might point a robot at a desired target and bring it into the vicinity. At this point, the robot could be given control, and begin an automated sequence utilizing a sonar sensor to position itself a precise distance from the object in order to pick it up. A semi-autonomous robot that has a human operator, but performs simple subtasks autonomously can maximize throughput (work accomplished per unit time). Robotics competitions are generally well suited to semi-autonomous robots.

In addition, writing short automation sequences is usually easy, while fully automating a robot for a complex task can be extremely difficult. Not only are certain human senses superior to the machine's, but it is also difficult for the programmer to anticipate every scenario, so having a brain in the equation that can cope with new and unexpected situations makes the engineering task much more tractable. As you attempt to increase the sophistication of your automation, you will eventually encounter a point of diminishing returns.

These exercises are designed to help you gradually learn how to deal with the challenges of automation. *Do not skip the early exercises on the assumption that they're too simple.* That assumption is wrong. Unless you've been programming robots for a while, you're probably unaware of the challenges presented by even the simplest tasks. What you learn from the simple exercises will make it possible for you to develop more complex automations.

These exercises should be done individually. Teams invariably have dominant members who end up doing most of the work. Watching them program is not much of a learning experience: you have to do it yourself before you really understand.

For the first few exercises, you will need a basic robot such as the squarebot described in the Vex Inventor's Guide. The exact design of the robot is not important, as long as it has the ability to drive forward, backward, and make relatively sharp turns. Differential steering is generally the most versatile. In this type of steering, a separate motor drives each side of the robot, and each motor can be individually controlled. Sensors will need to be added for later exercises, so leave room for them in the design.

All of the programs should wait for a specific button on the remote control unit to be pressed before beginning autonomous routines

Many exercises are based on previous exercises, so that they will require only minor modifications to previous programs. For each exercise, copy the beginner code folder or the previous exercise folder to a new folder. For example:

```
shell: cd OpenVex-0.4.4
shell: cp -R Beginner/ Exercise1
shell: cd Exercise1
shell: ape firmware.c

(edit and test your program)

shell: cd ..
shell: cp -R Beginner/ Exercise2
shell: cd Exercise2
shell: ape firmware.c

and so on...
```

Note: The trailing / after Beginner in the example above is important if the destination directory already exists. This will tell the cp command to copy the contents of Beginner to Exercise1 rather than create the directory Exercise1/Beginner.

12.3 Basic Motor Control

To complete these exercises, you will need to set up a trigger such as a button sensor, limit switch, or RC button press to trigger the autonomous routine.

First, get your robot to the point where you can drive it around under remote control. If you're using a simple robot with differential steering, this should simply be a matter of plugging the drive motors into the correct ports for the Beginner code, and maybe adapting the code slightly.

You can then edit the autonomous routine code to make the robot perform the task(s) you want.

1. Write a program that drives the robot forward exactly 1 foot, and then backs it up to the exact position where it started. When you your robot can do this 10 times in a row, you're ready to move on to the next exercise. Try this first using a timer, and note the challenges with this approach. Later, you can try it again using shaft encoders.
2. Write a program that drives the robot in a circle with a radius of exactly 1 foot. The robot should stop in the exact position that it started. When your robot can do this 10 times in a row, you're ready to move on to the next exercise.
3. Make a copy of the program from the previous exercise, and alter it to drive a 1.5 foot radius circle. When you your robot can do this 10 times in a row, you're ready to move on to the next exercise.
4. Using data collected from exercises 1 and 2, try to calculate the parameters necessary to drive a 2 foot diameter circle. Alter the program again to test your calculations. How close did you get? How could you have gotten closer?
5. Write a program that drives the robot in a square pattern with sides of 1 foot. The robot should stop in the exact same position where it started. When you your robot can do this 10 times in a row, you're ready to move on to the next exercise.

12.4 Basic Sensor Input

1. Add two bumper switches to the front of your robot. Drive the robot forward until it bumps into an object. Then back it up 6 inches, turn it around, and drive it back to the exact spot where it started. Turn it around again, so it is in the exact

position and orientation that it started from. When you your robot can do this 10 times in a row, you're ready to move on to the next exercise.

2. Mount a sonar sensor on top of your robot. Note that sonar sensors are fragile, so be sure to mount it in a safe place so it doesn't hit any objects in the area while your robot is moving. Program the robot to drive until it comes to within 20cm of a wall or other object. It should then turn around, and return to its starting position. When you your robot can do this 10 times in a row, you're ready to move on to the next exercise.
3. Classic line follower: Mount two line following sensors on opposite sides of the robot, facing down, and fairly close to the ground. (see the line following sensor documentation in the Inventor's Guide for details.) Write a program that follows the dark line on the Vex mat, keeping the line between the two sensors.
4. Repeat the first few exercises utilizing shafts encoders to count rotations on each drive wheel.