

**A Guide to Powerful Programming
for Embedded Systems**

Assembly Language Essentials

Larry Cicchinelli

**CIRCUIT
CELLAR**

Assembly Language Essentials

A Guide to Powerful Programming for Embedded Systems

Larry Cicchinelli

Circuit Cellar, Inc.
www.circuitcellar.com

Preview Only

Copyright Circuit Cellar, 2011

Purchase book at www.cc-webshop.com

Assembly Language Essentials: A Guide to Powerful Programming for Embedded Systems

Published by
Circuit Cellar Inc.
4 Park Street
Vernon, CT 06066 USA

Copyright © 2011 Circuit Cellar Inc. All rights reserved

This book and the individual contributions contained in it are protected under copyright by the Publisher. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the Publisher. Requests to the Publisher for permission should be addressed to: Publisher, Circuit Cellar, 4 Park Street, Vernon, CT 06066, USA, or sent to circuitcellar@circuitcellar.com. Circuit Cellar is an Elektor International Media company.

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or technologies may become necessary.

Practitioners and researchers must always rely in their own experience and knowledge in evaluating and using any information, methods, parts, electronics, designs, code, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any reliability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Control Number: 2011923558

ISBN 978-0-9630133-2-3

Prepress production: Eric A.J. Bogers
Cover Design: Helfrich Ontwerpbureau
Printed by Wilco

109034-UK

Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

Table of Contents

1	Terms and Definitions	9
	Introduction	9
	Abbreviations	10
	CPU Architecture	10
2	What is Assembly Language?	25
	High-Level Language Calling Conventions	31
3	Address Modes	33
	Instruction Execution Time	36
	Instruction Length	37
	Address Modes	37
4	This Processor	51
	Basic Processor Features	51
	Interrupts	53
	Memory Layout	55
	Processor Status Register	56
	Condition Codes	56
	Current Register Set	59
	Interrupt Priority Level	60
	Programmer's Model	60
	Address Modes	61

Preview Only
Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

5	I/O Devices	63
	Parallel Ports	65
	Parallel Port Registers	66
	Serial Ports	69
	Asynchronous Mode	70
	Synchronous Mode	71
	USART Registers	72
	Transmit/Receive Summary	77
	Timers and Counters	78
	Timer/Counter Registers	78
	Pulse Modulators	83
	Pulse Modulator Registers	84
	Sample Port Assignments	86
	Other Features	87
	External I/O	87
6	Assembler	89
	Special Address Symbols	92
	Directives	94
	General Directives	94
	Setting Memory Addresses	98
	Declaring Storage for Variables and Constants	100
	Sample Program	102
	Conditional Assembly Directives	104
	Sample Program	106
	Reading the Assembler List File	107
	Assembler Screen Shot	112
7	Introduction to Instruction Sets	115
	Instruction Categories	116
	Double Operand	116
	Single Operand	117
	Program Counter	119
	Miscellaneous	121

Preview Only
Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

8	Instruction Set	123
	Condition Codes	125
	Address Mode Timing	125
	Detailed Examples of Timing Calculation	126
	Instruction Length	128
	Detailed Examples of Length Calculation	128
	Instruction Format	129
	Double Operand Integer	130
	Single Operand Integer	139
	Rotate	147
	Shift	151
	Floating Point	154
	Floating Point Summary	154
	Floating Point in This Processor	156
	Single Precision Floating Point	157
	Double Precision Floating Point	159
	Register Set	162
	Program Counter	165
	Jump	165
	Call	167
	Branch	169
	Short I/O	173
	Block Copy	177
	Status Register and Miscellaneous	179
9	A Few Algorithms	181
	Duplicate Instructions?	181
	Multiply and Divide	181
	Multiplication Algorithms	182
	Division Algorithms	185
10	Simple Exercises	191
	Exercise 1	191
	Exercise 2	196
	Exercise 3	197
	Exercise 4	199

11	Interrupt Service Routines	203
	Serial Port Receive	204
	Serial Port Transmit	210
12	Exercises	213
	Exercise 5	213
	Exercise 6	217
	Exercise 7	222
13	An Alternate Processor Architecture and Instruction Set	233
Appendix I		
	Instruction Set Summary	243
Appendix II		
	I/O Register Summary	247
Appendix III		
	Assembler Error and Warning Messages	251
Appendix IV		
	Opcodes for Instruction Set 1	253
	Index	259

Preview Only
Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

1

Terms and Definitions

Introduction

There seems to be a lack of general Assembly Language books available today. You can find information on the Internet for specific processors, but not much for learning how to program in Assembly Language in general terms. It is my hope that this book will at least partially fill this vacuum. The goal of this book is to get the reader to the point where he/she is reasonably familiar with the concepts of Assembly Language programming – not to make him/her an expert. Expertise only comes with practice, and becoming an expert in Assembly Language takes quite a bit of work.

Although my formal schooling was in Electrical Engineering, I have been programming in Assembly Language for most of my professional career, starting with the Hewlett Packard 2100 series, the 8008, 8080, 6800, 6805, 6811, PDP8™, PDP11™, VAX™, PIC™ and most recently the Rabbit™ series. Each has taught me more about Assembly Language programming, and thus I've become a better programmer. Each processor has advantages and disadvantages.. The instruction sets have varied from fairly simple to quite complex.

The selection of a programming language should be based on the application's requirements. Typically, engineers choose Assembly language for a project when speed is a primary concern. Another valid reason is program size. Even though compilers are getting better at code generation, there are still applications for which hand coding yields a more efficient program. In today's embedded systems there is usually a combination of both Assembly Language and a higher-level language.

Development time should also be a consideration. Any given task will probably take longer to develop in Assembly Language than in a higher-level language. If your project is time-constrained, you should consider using a higher language for the bulk of the program and Assembly Language only where absolutely necessary. If you are more constrained by program size, then Assembly Language may be a necessity.

Preview Only

Copyright Circuit Cellar, 2011

Purchase book at www.cc-webshop.com

The primary goal of this book is to teach Assembly Language and not to focus on a specific processor. With that goal in mind, I present a fictional processor with its own hardware and instruction set. Subsequent chapters cover various features and details of the processor used in this book. A number of references in this chapter and the next refer to features of “this processor.”

Just another thought before you proceed – Assembly Language is probably not a good choice as a first programming language to learn. It is as close to the hardware of the processor as you can get. A good Assembly Language programmer will likely have a reasonably good understanding of hardware. If you have no previous programming experience, I highly recommend that you review a language such as C or BASIC in order to get an understanding of programming fundamentals.

Abbreviations

A number of abbreviations are typically used to describe features of processors and programming. The following abbreviations are used throughout this book. Learn them now if you don't already know them:

▶▶▶	ALU	Arithmetic Logic Unit
▶▶▶	CPU	Central Processing Unit
▶▶▶	IPL	Interrupt Priority Level
▶▶▶	ISR	Interrupt Service Routine
▶▶▶	LSB	Least Significant Bit
▶▶▶	MMU	Memory Management Unit
▶▶▶	MSB	Most Significant Bit
▶▶▶	PC	Program Counter
▶▶▶	PSR	Processor Status Register
▶▶▶	RAM	Random Access Memory
▶▶▶	ROM	Read Only Memory
▶▶▶	SP	Stack Pointer

CPU Architecture

There are many topics that can be addressed when discussing processors and programming them. In this section I describe some major CPU architecture features.

The starting point for learning how to program any processor in Assembly Language is its CPU architecture. The term CPU is often used to refer to the entire processor, but here I will use a more narrow definition which refers to the processor “core,” where all of the high-speed logic is contained such as: the logic for the PC, instruction decode and execution, the ALU, etc. The ALU contains the logic for all of the arithmetic and logical operations.

Preview Only

Copyright Circuit Cellar, 2011

Purchase book at www.cc-webshop.com

2

What is Assembly Language?

Assembly language is generally considered as the lowest level of programming languages¹. Sometimes it is referred to as machine code; however, machine code is really the 1's and 0's that make up the binary values which the processor actually executes. For any given processor there is a 1:1 relationship² between assembly language and machine code – every assembly language statement has exactly one machine code equivalent and every machine code value has exactly one assembly language equivalent.

Assuming there is no operating system, when you program in Assembly Language you are in complete control of the processor and what it is doing. Your program has to do everything. You cannot simply add two numbers and print the result. You have to write the step-by-step instructions to get the numbers, add them and then display the sum. The following example gives you an idea of the details involved.

As an introduction to Assembly Language here is a very simple C program followed by an explanation of the assembly language statements the compiler generated:

```
main ()
{
    int      int_a;
    long     long_a;
    float    float_a;

    int_a = 1;
    long_a = 1;
    float_a = 1;           // the compiler automatically
```

1 The only possible exception would be that of microprogramming. Very simply put, microprogramming is a technique for implementing assembly language instructions using even lower level instructions. Each assembly language instruction consists of one or more micro-instructions. However, I am not aware of any current processors that use microprogramming. You can find several discussions on the Internet about microprogramming, but all the ones I found pre-date 1995.

2 In some very rare cases, one assembly language statement can have two machine code equivalents.

```

        // "promotes" the integer 1
        // to a floating point value

float_a = 1 + int_a; // the compiler automatically...
}

```

The compiler used for this example is Dynamic C™, version 10.56, for a Rabbit™ 5000 processor. The compiler will first generate Assembly Language statements and then pass them to an Assembler. Below are the assembly language statements which were generated. Comments have been inserted that describe each statement as well as some explanatory notes. The compiler uses lower case letters for the registers – they have been changed to upper case to make them easier to read. First, a few comments about the statements that follow:

- ▣ The syntax of each statements is: instruction argument one, argument two.
 - Some instructions do not have argument two.
- ▣ The C source statements are on lines by themselves and are shown as comments.
 - Comments are indicated using a ‘;’
- ▣ The first text on each of the generated lines is the assembly language instruction
 - add add the value of argument two to the value of argument one leaving the result in the location specified by argument one.
 - ld copy the value of argument two to the location specified in argument one
 - lcall call the function (or subroutine) indicated by the argument
 - clr clear (put a value of 0 into) the contents of the argument
- ▣ The 16 bit registers are
 - SP Stack Pointer
 - HL The combined H and L registers (8 bits each), H is the MSByte
 - BC The combined B and C registers (8 bits each), B is the MSByte
 - DE The combined D and E registers (8 bits each), D is the MSByte

Preview Only
 Copyright Circuit Cellar, 2011
 Purchase book at www.cc-webshop.com

- ⇒ The only 32 bit register is BCDE – the combined B, C, D and E registers (8 bits each), B is the MSByte
- ⇒ “0x” is a prefix for hexadecimal values

```

add    SP, -10           ; make room on the stack for the auto
                        ; variables

```

Since these are all auto variables, they will be placed on the stack. The compiler is inserting code which allocates 10 bytes of stack space – int_a requires 2 bytes, long_a and float_a both require 4 bytes. The Rabbit instruction set does not have a subtract instruction that uses SP.

```

; int_a = 1;
ld     HL, 0x0001       ; load an integer value of 1 into the
                        ; HL register (16 bits)
ld     (SP + 0x08), HL  ; store the contents of HL into
                        ; the auto variable location
                        ; for int_a

```

This processor does not have an instruction that loads a constant directly to a memory location.

```

; long_a = 1;
ld     BCDE, 1          ; load an integer value of 1 into the
                        ; BCDE register (32 bits)
ld     (SP + 0x04), BCDE ; store the contents of BCDE
                        ; into the auto variable
                        ; location for long_a

; float_a = 1;
ld     BCDE, 0          ; load a value of 0 into the BCDE
                        ; register (32 bits)
ld     BC, 0x3F80       ; load the most significant 16 bits of
                        ; a floating point 1.0 into register
                        ; BC
ld     (SP + 0x00), BCDE ; store the contents of BCDE
                        ; into the auto variable
                        ; location for float_a

```

The lower 2 bytes of a floating point value of 1.0 are 0. The compiler “knows” how to construct the needed binary pattern.

Preview Only

Copyright Circuit Cellar, 2011

Purchase book at www.cc-webshop.com

```

; float_a = 1 + int_a;
ld    HL, (SP + 0x08)    ; load the contents of the auto
                        ; variable int_a into register HL
ld    DE, 0x0001        ; load an integer value of 1 into
                        ; the DE register (16 bits)
add   HL, DE            ; add the contents of DE to the
                        ; contents of HL

```

The compiler recognizes the two values being added as integers. It also recognizes that the constant value (1) can be represented with 16 bits and is the same size as the variable so they can be added without having to convert either of them to a compatible data type.

```

lcall Hi_Bf            ; convert the integer value in HL to
                        ; floating point and store in BCDE

```

Since the values just added result in an integer value the compiler recognizes that the result must be converted to floating point. The compiler will include the function `Hi_Bf` in the code to do the conversion. From examining the statements we can conclude that the function wants the integer value in HL and returns the result in BCDE.

```

clr   HL              ; offset on the stack for the auto
                        ; variable float_a
add   HL, SP          ; add the contents of the stack
                        ; pointer to HL to get the address of
                        ; float_a

```

Obviously, simply copying `SP` to `HL` will accomplish the same thing or even better – loading BCDE directly to `(SP)`. However, the code generator must handle the general case. I am not a compiler writer but I imagine that optimizing the code for this case might prove to be somewhat difficult.

```

ld    (HL), BCDE     ; store the contents of BCDE into the
                        ; auto variable location for float_a

```

As you can see from the generated code, programming in Assembly Language is quite different from higher level languages. You have to do everything yourself. Even the code shown above does not show all the code generated by the compiler (i.e., the code for the `Hi_Bf` function). Also, since the code above was generated for a C environment there is lots of other code that does not have anything to do directly with the C statements but that are used to set up the target processor, debugging, etc.

When programming in assembly language, the programmer must think of every detail involved in the process for the task at hand. The following outline shows exactly what this means. The task is to add two numbers and display the result.

- ▣ Get the first number
- ▣ Get the second number
- ▣ Add them
- ▣ Display them

The outline seems simple enough, but exactly how do you program each of the steps? It depends entirely on the processor: the instructions and hardware it supports. There are questions which need to be answered:

- ▣ What kind of numbers are they?
 - Integer
 - 16 bits
 - 32 bits
 - Floating point
 - Single precision
 - Double precision
- ▣ Are they both the same kind?
- ▣ Does the program have to support whatever numbers are entered?
- ▣ How does the program get the numbers?
 - Constants embedded in the program
 - Operator entry
- ▣ What protocol is used to communicate with the display

These questions need to be asked whatever the programming language, but in assembly language the implementation is quite a bit more difficult.

That implementation depends on a number of factors based on the processor:

- ▣ Does the processor have floating point hardware
- ▣ What size are the registers
- ▣ How many registers does it have
- ▣ Is it an orthogonal instruction set
- ▣ What kind of I/O devices does it have

Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

A higher level language allows the programmer to ignore most, if not all, of these questions. For instance – if the processor does not support floating point hardware, there will be software functions available to handle the floating point operations (as in the above sample). If the processor has only eight and/or 16 bit registers, the compiler will have functions to handle the arithmetic operations that require more than eight or 16 bits.

Here is a more detailed outline that will probably suffice for a pure assembly language implementation. The assumption is that both values will be entered via a serial interfaced terminal and that the result will be displayed on that same terminal.

- ▣ Define whatever memory configuration values are necessary
- ▣ Allocate RAM for all variables – including terminal input
- ▣ Allocate ROM for and define all messages to be sent to the terminal
- ▣ Ask for the first value
 - This probably requires using a UART and the software driver necessary for accessing it
 - The driver needs to be written
- ▣ Get the value
 - Again – this requires a UART and the driver which will have to be written
- ▣ Determine the data type, convert and store the binary value accordingly
 - This is not as easy as it may sound:
 - Check for a leading ‘-’ to see if it is negative
 - Check for a leading ‘+’
 - Is there a decimal point – making it floating point?
 - Does the program have to support ‘E’ format?
- ▣ Ask for the second value
- ▣ Get the value
- ▣ Determine the data type, convert and store the binary value accordingly
- ▣ Add the two values based on their data type

Preview Only
Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

- ➡ Convert the sum to ASCII
- ➡ Display a message with the result

As you can see there is quite a bit of detail involved in writing an Assembly Language program. However, in most cases you will probably not be writing an entire program – you will be writing one or more functions in Assembly Language. As stated elsewhere in this book, one of the main reasons for writing in Assembly Language is execution speed. If your application is written in C, or any other high level language, and it requires any high speed processing you may need to consider writing some function(s) in Assembly Language. This is especially true for Interrupt Service Routines which should be executed as quickly as possible.

As is the case with most things, the only way to become proficient with Assembly Language is to use it. There are several exercises later in this book which can be used to get you started. Although this book is not about any specific processor, the skills acquired learning the Assembly Language for any processor can be used immediately with any other.

Basically you will need to learn the syntax of the new instructions, the register set and processor architecture. This is not as difficult as it may at first seem. There are certainly differences but usually once you have learned the Assembly Language for one processor, learning the details of another processor is relatively easy.

High Level Language Calling Conventions

In many cases your Assembly Language code will have to interface to a higher level language, sometimes abbreviated HLL. These interfaces will usually fall into one of two categories:

- 1 Stand-alone such as is found in an Interrupt Service Routine
- 2 Called from the HLL

There are a few examples of ISRs in Chapter 11 of this book. The main criteria you need to consider in an ISR, as far as the HLL is concerned, is the preservation of registers. When the interrupt occurs, your ISR code has no idea of how the HLL is using the processors registers so it is up to the ISR to both save and restore any registers it uses which may also be in use by the calling function. All of this is discussed in more detail in Chapter 11.

When an Assembly language function is called from an HLL, it is up to the compiler, or interpreter, to define the calling convention. In general, the HLL will allow the called function to use all available registers. However, there are strict rules for passing parameters such as:

Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

- ▣ The first argument may be put into a specific register. The register that is used may be dependent on the size of the argument and whether or not the argument is a pointer.
 - ▣ All arguments, including the first, are usually pushed onto the stack. In most cases the last argument is the first one pushed on the stack. This makes the first argument the last one pushed on the stack.
 - ▣ Some systems may put the number of arguments as the last value pushed. This will only be required if the number of arguments is variable.
 - ▣ It is up to the called function to “know” the data type, size and stack offset for each argument.
 - ▣ Global values can usually be accessed directly by referencing their names.
 - ▣ Local values, usually stored on the stack, will need to be accessed using a compiler/interpreter defined method.
- See chapter 6 for definitions of Global and Local.*
- ▣ Keep in mind that the last item pushed onto the stack is the return address.
 - ▣ A return value will usually be put into a specific register.

You need to read the compiler/interpreter documentation and thoroughly understand the conventions it uses for calling Assembly Language functions. Debugging Assembly Language can be difficult but it will be considerably more so if these rules are not strictly followed. Here are two possibilities for errors:

- 1 Using a register which must be preserved but you do not. Your function may execute just fine but the program will eventually crash.
- 2 Using parameters off the stack with the incorrect size or retrieving them with an incorrect stack offset will cause your function to execute incorrectly.

Preview Only
Copyright Circuit Cellar, 2011
Purchase book at www.cc-webshop.com

Index

I		C	
1's-Complement	16	C-bit	57
2's-Complement	16	Central Processing Unit	10
A		CISC	21
Address Mode Timing	125	Condition Codes	11, 56, 125
Address Modes	11, 61	Counters	78
Address Absolute	48, 61	CPU	10
Address Relative	46	CPU_F	87
Auto-Increment/Decrement	44	CPU_FREQUENCY	87
Auto-Increment/Decrement Indirect	46	Current Register Set	59
Immediate	39, 61	D	
Indirect	40	Destination	124
Offset	42, 61	Direct Memory Access	12
Offset Deferred	43	Directives	89
Offset Indirect	43	#address	98
Register	38, 61	#align	98
Register Deferred	40	#def_data_size	101
Register Indirect	40, 61	#define	94
Address Pointer	93, 98	#error	96
ALU	10	#global	95
Argument	123	#if !defined	105
Arithmetic Logic Unit	10	#if defined	105
Assembler	89	#ifdef	105
Atomic	11	#ifndef	105
B		#include	96
BCD	17	#nprint	96
Big Endian	15	#print	96
Binary Coded Decimal	17	#radix	96
Bottom of the Stack	22	#warning	96
		#width	98
		Conditional	
		#else	105
		#endif	105
		#if	104

Preview Only
 Copyright Circuit Cellar, 2011
 Purchase book at www.cc-webshop.com

Expression	95	SUB	138
Memory allocation		XOR	138
#alloc	100	Floating Point	
#allocb	101	FPDADD	160
#vector	99	FPDCOMP	160
Directives	94	FPDDIV	161
DMA	12	FPDI	161
E		FPDMUL	161
Execution time	36	FPDSP	162
Exponent	154	FPDSUB	162
Expression	91, 94, 104	FPID	159
External I/O	12	FPIS	157
F		FPSADD	157
Flash Memory	12	FPSCOMP	157
Floating Point	17, 154, 155, 157, 159, 161	FPSDIV	158
G		FPSDP	158
Global	32, 91	FPSI	159
H		FPSMUL	158
Harvard	13	FPSSUB	159
I		Program Counter	
I/O	13	BCC	171
IEEE 754	155	BCS	171
Initialization file	89	BEQ	171
Instruction Format	129	BGE	171
Instruction Length	128	BGT	171
Instruction Set		BGTU	172
Block Copy	177	BLE	172
BC	177	BLEU	172
BCID	178	BLT	172
BCIS	178	BMI	172
Double Operand Integer		BNC	173
ADD	131	BNE	173
AND	132	BNS	172
ASHIFT	132	BPL	173
BITEST	133	BR	171
COMP	134	BVC	173
COPY	135	BVS	173
DIV	135	BZC	173
DIVU	136	BZS	171
MUL	136	CALL	167
MULU	137	CALLR	167
OR	137	CALLS	168
		DBGE	169
		DBNE	170
		JMP	165
		JMPR	166
		JMPS	166
		RETI	168
		RETS	168
		PSR and Misc	
		CCC	179
		CCS	179

Preview Only
 Copyright Circuit Cellar, 2011
 Purchase book at www.cc-webshop.com

NOP	179	L	
PSRCOPY	179	Label	90
SETIPL	179	Latency	54
WFI	180	Least Significant Bit	10, 14
Register Set		Least Significant Byte	14
POPR	164	Linker	91
PUSHR	164	Little Endian	15
RCOPY	163	Load and Store	15, 116, 234
REXCH	163	Local	32, 91
RSCOPY	163	Logical Memory	19, 20
RSDEF	163	LSB	10, 14
RSEXCH	163	LSByte	14
Shift and Rotate		M	
RL	148	Machine code	25
RLB	150	Mantissa	154
RLC	148	Memory Management	20
RR	149	Memory Management Unit	10, 15
RRB	150	Memory Mapped I/O	15
RRC	149	Microprogramming	25
SLA	151	MMU	10, 15
SLL	152	Most Significant Bit	10, 16
SLLC	152	Most Significant Byte	16
SRL	153	MSB	10, 16
Short I/O	173	MSByte	16
IOAND	173	N	
IOCOPY	174	N-bit	57
IOOR	175	NEXT_PC	93
IOTST	176	NEXT_RAM	92
Single Operand Integer		NEXT_ROM	93
ADDC	139	Number Format	16
CLR	140	Number of Bits	18
COM	140	O	
DEC	141	Opcode	37
DECP	141	Operand	123
INC	142	Operating Modes	18
INCP	142	Operators	95
INV	143	Orthogonal	18
POP	143	P	
PUSH	144	Parallel Ports	65
SUBC	145	Parameter	123
SWAP	145	PC	10
SXT	146	Physical Memory	19, 20
TEST	147	Pop	22
Status Register	179		
Interrupt	13		
Interrupt latency	54		
Interrupt Priority Level	10, 60		
Interrupt Service Routine	10, 14		
Interrupt vector	14, 54, 99		
Interrupts	53		
IPL	10		
ISR	10, 14, 53		

Position Independent	19	Signed	21
PPM	83	Sign-magnitude	17, 155
Processor Status Register	10, 19, 56	Source	123
Processor_Definitions.h	63	SP	10
Program Counter	10, 20	SPI	71
PSR	10, 19	SPI Timing	74
Pull	22	Stack	22
Pulse Position Modulators	83	Stack overflow	22
Pulse Width Modulators	83	Stack Pointer	10, 22
Push	22	Stack underflow	22
PWM	83	Symbol	90
R		T	
RAM	10, 20	Tables of constants	90
Random Access Memory	10	Timers	78
Read Only Memory	10	Top of the Stack	22
Read-modify-write	37, 124	U	
Real Time Clock	87	Unsigned	21
Registers	21	USART	69
Relational operators	104	V	
RISC	21	Value	89
ROM	10, 20	V-bit	59
RS232	69	Vector	14
RS485	69	Vector table	53
RTC	87	Von Neumann	13
S		Z	
Serial Ports	69	Z-bit	57
Shift and add	182		
Shift and subtract	185		
Sign bit	154		