PICMicro® Microcontroller Pocket Reference

- Key charts, protocols, and data tables
- Small trim, easy-to-scan layout
- Get all the facts and reference data

MYKE PREDKO
PICmicro®
Microcontroller
Pocket
Reference
This page intentionally left blank.
This page intentionally left blank.
Contents

Chapter 1. Conventions Used in This Book 1

Chapter 2. PICmicro® MCU Part Number Feature Comparison 3
   Feature to Part Number Table 3

Chapter 3. Device Pinouts 7
   Low-End 8
   Mid-Range 10
   PIC17Cxx 14
   PIC18Cxx 17

Chapter 4. PICmicro® MCU Instruction Sets 21
   Parameters 21
   Low-End Instruction Set 22
   Mid-Range Instruction Set 31
   PIC17Cxx Instruction Set 40
   PIC18Cxx Instruction Set 63
   Microchip Special Instruction Mnemonics 104
   Parallax PICmicro® MCU Instruction Set 111
vi  Contents

Chapter 5. PICmicro® MCU Processor Architectures 123
The PICmicro® MCU's Arithmetic Logic Unit 123
Low-End PICmicro® MCUs 125
  Register access 126
  STATUS register 128
  Program counter 128
Mid-Range PICmicro® MCUs 130
  Register access 130
  STATUS register 132
  Program counter 133
  Interrupt operation 134
  Interrupt handler skeleton 135
PIC17Cxx 136
  Register access 137
  STATUS register 138
  Program counter 139
  Interrupt operation 140
  Interrupt handler skeleton 142
PIC18Cxx 142
  Register access 144
  STATUS register 146
  Program counter 147
  Interrupt operation 150
  Interrupt handler skeleton 150

Chapter 6. PICmicro® MCU Register Mappings 151
Low-End PICmicro® MCUs 151
Mid-Range PICmicro® MCUs 154
PIC17Cxx 169
PIC18Cxx 186
Contents

Parallel Bus Device Interfacing 282
Button Interfacing 284
Switch Matrix Keypad/Keyboard Interfacing 287
Combining Input and Output 291
Simulated “Open Collector”/“Open Drain” I/O 292
LEDs 293
  Multisegment LED displays 293
LCD Interfaces 296
I2C Bit Banging “Master” Interface 334
RS-232 Interfaces 338
RS-485/RS-422 345
Asynchronous Serial I/O Software Routines 347
Dallas Semiconductor One-Wire Interface 357
Reading Potentiometer Using Parallel I/O Pins 363
Motor Drivers 365
R/C Servo Control 370
Audio Output 371
AC Power Control 373
Hall-Effect Sensors 374
Sony Infrared TV Remote Control 374

Chapter 9. PICmicro® MCU Programming 379
  “Hex” File Format 379
  Low-End PICmicro® MCU Programming 381
  Mid-Range Serial Programming 385
  PIC17Cxx Programming 394
  PIC17Cxx ICSP Programming 398
  PIC18Cxx Programming 402
  Microchip ICSP Programming Connector 405
  Third Party/Downloadable Programmers 406
## Contents

**Chapter 10. PC Interfaces** 407  
Memory Map 407  
I/O Space Map 407  
Interrupt Function by Number 422  
ISA Bus 425  
ISA pinouts 425  
Interrupts 429  
Keyboard and Mouse Ports 430  
Connector specification 430  
Keyboard operation with timing diagrams 431  
Keyboard scan codes 432  
Keyboard controller commands 436  
BIOS interfaces 436  
Keyboard commands 441  
Serial Port 441  
Connector pinouts 442  
8250 block diagram 443  
Serial port base addresses 443  
8250 registers 444  
Interrupts 449  
Interrupt 14h–RS-232 communications APIs 451  
Parallel Port 455  
Block diagram/connector 455  
Base registers 457  
Registers 457  
Data output waveform 458  
BIOS interfaces 459

**Chapter 11. Useful Code “Snippets”** 461  
Jumping Outside the Current Page 461  
Tables 462  
Conditional Branching 465
Contents

Time Delays 466
Negating the Contents of a Register 467
Incrementing/Decrementing “w” 468
Rotating a Byte in Place 469
Copy Bits from One Register to Another 470
Converting a Nybble to ASCII 470
Converting an ASCII Byte to a Hex Nybble 471
Using TOCKI as an Interrupt Source Pin 471
Dividing by Three 472
Sixteen-Bit Pulse Measurement with 5-Cycle Delay 473
Detect a Change in a Register 474
Test a Byte within a Range 474
Convert ASCII to Upper Case 475
Swap the Contents of “w” with a Register 475
Swap the Contents of Two Registers 475
Compare and Swap if Y < X 476
Counting the Number of “1”s in a Byte 476
Generating Parity for a Byte 477
Keeping a Variable within a Range 477
Swapping Bit Pairs 478
Bitwise Operations 478
Constant Multiplication 479
Constant Division 480

Chapter 12. 16-Bit Numbers 483

Defining 16 Bit Numbers 483
Increments and Decrements 484
Addition/Subtraction 485
Bitwise Operations on Constants and Variables 488
Contents

Comparisons with 16-Bit Variables 489
Multiplication 492
Division 495

Chapter 13. PICmicro® MCU
Operations Tables 499
I/O Pin Current Capabilities 500
RC Oscillator Component Values 500
LP Oscillator Operating Characteristics 501
XT Oscillator Operating Characteristics 502
HS Oscillator Operating Characteristics 504

Chapter 14. PICmicro® MCU Application
Debugging Checklist 507

Chapter 15. PICmicro® MCU Application
Software Development Tools 517
Microsoft Compatible Editor “Ctrl” Key
Combinations 517
MPSIM.INI 518
MPLAB 519
Stimulus (.STI) files 525
MPLAB assembler directives 526
Standard Declaration and Include (“.inc”) Files 536
Linking–Linked Applications 536
Application Code Template 539
The BASIC Language 540
Microsoft BASIC Enhancements 542
PicBasic 549
xi Contents

Visual Basic 578
  MSComm control 581
The “C” Language 588
  Declarations 588
  Statements 590
  Operators 593
  Directives 595
  “Backslash” characters 598
  Common C functions 598
  PICmicro® MCU enhancement functions 603

Chapter 16. Constants and Data Tables 605
  Mathematical and Physical Constants 605
  ASCII 606
    ASCII control characters 607
    ANSI display control sequences 609
    IBM PC extended ASCII characters 612
    Windows ASCII characters 612
  EBCDIC 612
  Audio Notes 615
  “Touch-Tone” Telephone Frequencies 616
  Modem “AT” Commands 616
  Modem registers 622
  Morse Code 625
  Phonetic Alphabets 626
  “Ten” Radio Codes 626

Chapter 17. Miscellaneous Electronics 629
  Resistor Color Coding 629
  Electromagnetic Spectrum 630
    Radar bands 632
This page intentionally left blank.
Chapter 1

Conventions Used in This Book

Hz Hertz (Cycles per Second)
kHz Kilohertz (Thousands of Cycles per Second)
MHz Megahertz (Millions of Cycles per Second)
GHz Gigahertz (Billions of Cycles per Second)
bps Bits per Second
kbps Thousands of Bits per Second
mbps Millions of Bits per Second
KBytes 1,024 Bytes
MBytes 1,048,576 Bytes
GBytes 1,073,741,824 Bytes
## Chapter 1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1,000 ohms</td>
</tr>
<tr>
<td>uF</td>
<td>microfarads</td>
</tr>
<tr>
<td>ms/msecs</td>
<td>milliseconds</td>
</tr>
<tr>
<td>us/usecs</td>
<td>microseconds</td>
</tr>
<tr>
<td>0x0nn, $nn, 0nnh, and H'nn'</td>
<td>Hex Numbers</td>
</tr>
<tr>
<td>0b0nnn, %nnn, 0numb, and B'nnn'</td>
<td>Binary Number</td>
</tr>
<tr>
<td>nn, 0nnn, and .nnn</td>
<td>Decimal Number</td>
</tr>
<tr>
<td>AND and &amp;</td>
<td>Bitwise “AND”</td>
</tr>
<tr>
<td>OR and</td>
<td>Bitwise “OR”</td>
</tr>
<tr>
<td>XOR and ^</td>
<td>Bitwise “XOR”</td>
</tr>
</tbody>
</table>

* _Label_ Negative Active Pin. In some manufacturer’s data sheets this is represented with a leading “!” character or with a bar over the entire label.

* [parameter] The parameter is optional.

* [parameter]: One or another parameter can be used.
Chapter 2

PICmicro® MCU
Part Number
Feature Comparison

Feature to Part Number Table
The following table lists the different PICmicro® MCU families with the features that are specific to them.
<table>
<thead>
<tr>
<th>Part Number</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C5xx</td>
<td>8-Pin PICmicro® MCU. 12-Bit (Low-End) Processor. Internal Reset &amp; Oscillator.</td>
</tr>
<tr>
<td>PIC12C6xx</td>
<td>8-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. 8-Bit ADC/Internal Reset &amp; Oscillator/Optional EEPROM Data Memory.</td>
</tr>
<tr>
<td>PIC16C5x</td>
<td>18- to 28-Pin PICmicro® MCU. 12-Bit (Low-End) Processor.</td>
</tr>
<tr>
<td>PIC16C505</td>
<td>14-Pin PICmicro® MCU. 12-Bit (Low-End) Processor. Internal Reset &amp; Oscillator.</td>
</tr>
<tr>
<td>PIC16HV540</td>
<td>18-Pin PICmicro® MCU. 12-Bit (Low-End) Processor. Extended Vdd Capabilities with Built-In Regulator.</td>
</tr>
<tr>
<td>PIC16C6x</td>
<td>18- to 40-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. Optional TMR1 &amp; TMR2/Optional SPI/Optional USART/Optional PSP.</td>
</tr>
<tr>
<td>PIC16C62x</td>
<td>18-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. Voltage Comparators Built-In with Voltage Reference/Optional EEPROM Data Memory.</td>
</tr>
<tr>
<td>PIC16F62x</td>
<td>18-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. Flash Program Memory/Voltage Comparators Built-In with Voltage Reference/Internal Reset &amp; Oscillator.</td>
</tr>
<tr>
<td>PIC16C642</td>
<td>28-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. Voltage Comparators Built-In with Voltage Reference.</td>
</tr>
<tr>
<td>PIC16C662</td>
<td>40-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. Voltage Comparators Built-In with Voltage Reference.</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PIC16C71x</td>
<td>18-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. 8-Bit ADC.</td>
</tr>
<tr>
<td>PIC16C7x</td>
<td>18- to 40-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. 8-Bit ADC/Optional TMR1 &amp; TMR2/Optional SPI/Optional USART/Optional PSP.</td>
</tr>
<tr>
<td>PIC16C77x</td>
<td>28- to 40-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. 12-Bit ADC/TMR1 &amp; TMR2/USART/I2C/SPI/Optional PSP.</td>
</tr>
<tr>
<td>PIC16F8x</td>
<td>18-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. Flash Data and Program Memory.</td>
</tr>
<tr>
<td>PIC16F87x</td>
<td>28- to 40-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. 10-Bit ADC/TMR1 &amp; TMR2/USART/I2C/SPI/Optional PSP.</td>
</tr>
<tr>
<td>PIC16C92x</td>
<td>64-Pin PICmicro® MCU. 14-Bit (Mid-Range) Processor. Optional 8-Bit ADC/TMR1 &amp; TMR2/LCD Controller.</td>
</tr>
<tr>
<td>PIC17C4x</td>
<td>40-Pin PICmicro® MCU. 16-Bit (High-End) Processor. USART/Multiply.</td>
</tr>
<tr>
<td>PIC17C5x</td>
<td>68-Pin PICmicro® MCU. 16-Bit (High-End) Processor. USART/I2C/Multiply/10-Bit ADC.</td>
</tr>
<tr>
<td>PIC17C6x</td>
<td>84-Pin PICmicro® MCU. 16-Bit (High-End) Processor. USART/I2C/SP/12-Bit ADC.</td>
</tr>
<tr>
<td>PIC18Cxxx</td>
<td>28- to 40-Pin PICmicro® MCU. 16-Bit Advanced (18Cxx) Processor. USART/I2C/SP/10-Bit ADC.</td>
</tr>
</tbody>
</table>
Mid-range PICmicro® MCU Part Number to Feature Breakout

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>16Cx1</td>
<td>18-Pin PICmicro® MCU. 1K Program Memory. No USART/SPI/I2C. PIC16C61 and PIC16C71 are currently “obsoleted.”</td>
</tr>
<tr>
<td>16Cx2</td>
<td>28-Pin PICmicro® MCU. 2K Program Memory. SPI/TMR1 &amp; TMR2.</td>
</tr>
<tr>
<td>16Cx3</td>
<td>28-Pin PICmicro® MCU. 4K Program Memory. USART/SPI/TMR1 &amp; TMR2.</td>
</tr>
<tr>
<td>16Cx4</td>
<td>40-Pin PICmicro® MCU. 4K Program Memory. USART/SPI/PSP/TMR1 &amp; TMR2.</td>
</tr>
<tr>
<td>16Cx5</td>
<td>40-Pin PICmicro® MCU. 4K Program Memory. USART/SPI/PSP/TMR1 &amp; TMR2.</td>
</tr>
<tr>
<td>16Cx6</td>
<td>28-Pin PICmicro® MCU. 8K Program Memory. USART/SPI/I2C/TMR1 &amp; TMR2.</td>
</tr>
<tr>
<td>16Cx7</td>
<td>40-Pin PICmicro® MCU. 8K Program Memory. USART/SPI/I2C/PSP/TMR1 &amp; TMR2.</td>
</tr>
</tbody>
</table>
Chapter

3

Device Pinouts

As a rule of thumb, Pin-Through-Hole ("PTH") parts ("P" and "JW") are standard 0.300" and 0.600" widths with pins 0.100" apart in dual in-line packages. The height of the device is dependent on the package used. Surface Mount Technology ("SMT") parts are either in dual in-line packages ("SO") or in quad plastic chip carriers ("PT", "PQ", and "L").

For actual device dimensions, check the datasheets (on the CD-ROM or from the Microchip Web site) for the PICmicro® MCU that you are planning on using. Different packages for different PICmicro® MCUs have different via pad and clearance specifications.
Low-End

There are no PLCC or QFP packages used for the low-end devices and the pinouts remain the same whether or not the PICmicro® MCU is in a surface mount technology or pin-through-hole package (Figs. 3.1–3.4).

8 Chapter 3

Figure 3.1 “PIC12C508”/“PIC12C509” Pinout

Figure 3.2 “PIC16C505” Pinout
Device Pinouts

Figure 3.3 "PIC16C54/PIC16C56" Pinout

Figure 3.4 "PIC16C55/"PIC16C57" Pinout
Chapter 3

Mid-Range

The mid-range devices have the widest range of pinouts of any of the PICmicro® MCU families (Figs. 3.5–3.10). For many of the devices, the pinout is similar, but the pin functions may be different. In these cases, the pins marked with “*” show that these pins have other, optional purposes. Actual part number functions can be confirmed with Microchip Datasheets.

The PIC14000, which is designed for “Mixed Signals” uses the 28-pin packaging of the standard devices, but the pinouts are different as shown in Fig. 3.11.

The PIC16C92x LCD Driver microcontrollers are fairly high pin count devices. Figure 3.12 shows the 64-pin “DIP” ("Dual In-line Package") part. There is also a “PLCC” and “TQFP” package for the parts as well.

[Diagram of PIC12C67x Pinout]

*** - Indicates Analog I/O Pin
"JW"/"P"/"SO" Packages
0.300" PTH Package Width

Figure 3.5 “PIC12C67x” Pinout
Device Pinouts

**Figure 3.6** Mid-Range 18-Pin PICmicro® MCU Pinout

- Indicates Analog I/O Pin for 16C62x and 16C71(x)
- 0.300" PTH Package Width

**Figure 3.7** Mid-Range PICmicro® MCU 28-Pin Device Pinout

- Indicates Analog I/O Pin
- "&" - Indicates PIC16F87x Program/Emulator
- "JW"/"PT"/"SO" Packages
- 0.300" PTH Package Width
**Figure 3.8** Mid-Range PICmicro® MCU 40-Pin Device Pinout

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB1</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB2</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB3</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB4</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB5</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB6</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB7</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB8</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB9</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB10</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB11</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB12</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB13</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB14</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB15</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB16</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB17</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB18</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB19</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB20</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB21</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB22</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB23</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB24</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB25</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB26</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB27</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB28</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB29</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB30</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB31</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB32</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB33</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB34</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB35</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB36</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB37</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB38</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB39</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB40</td>
<td>Analog I/O Pin</td>
</tr>
</tbody>
</table>

**Figure 3.9** Mid-Range PICmicro® MCU 44-Pin "PLCC" Pinout

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB1</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB2</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB3</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB4</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB5</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB6</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB7</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB8</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB9</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB10</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB11</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB12</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB13</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB14</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB15</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB16</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB17</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB18</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB19</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB20</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB21</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB22</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB23</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB24</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB25</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB26</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB27</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB28</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB29</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB30</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB31</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB32</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB33</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB34</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB35</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB36</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB37</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB38</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB39</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB40</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB41</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB42</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB43</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB44</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB45</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB46</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB47</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB48</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB49</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB50</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB51</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB52</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB53</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB54</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB55</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB56</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB57</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB58</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB59</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB60</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB61</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB62</td>
<td>Analog I/O Pin</td>
</tr>
<tr>
<td>RB63</td>
<td>Analog I/O Pin</td>
</tr>
</tbody>
</table>

---

*Indicates Analog I/O Pin

0.600" Package Width

"8" - Indicates PIC16F87x Program/Emulator
Device Pinouts

Figure 3.10 Mid-Range PICmicro® MCU 44-Pin "QFP" Pinout

Figure 3.11 PIC14000 28-Pin Device Pinout
PIC17Cxx

The PIC17Cxx PICmicro® MCUs are available in 40- or 64-pin DIP packages as shown in Figs. 3.13 and 3.14. "PLCC" and "TQFP" surface mount packages as well for the 40-pin parts are displayed in the following graphics (Figs. 3.13–3.16).
Figure 3.13 PIC17C4x 40-Pin Device Pinout

Figure 3.14 PIC17C75x 64-Pin Device Pinout
Figure 3.15  PIC17C4x 44-Pin “PLCC” Pinout

Figure 3.16  PIC17C4x 44-Pin “QFP” Pinout
PIC18Cxx

There is a lot of similarity between the mid-range PICmicro® MCU's pinouts and the PIC18Cxx parts, as will be seen in the following pinouts (Figs. 3.17–3.20). Note that several pins that are optional in one PICmicro® MCU family are not optional in others.

Figure 3.17  PIC18C2X2 28-Pin Device Pinout
Chapter 3

Figure 3.18  PIC18C4X2 40-Pin Device Pinout

Figure 3.19  PIC18C4X2 44-Pin "PLCC" Pinout
Device Pinouts

Figure 3.20 PIC18C4X2 44-Pin "QFP" Pinout
This page intentionally left blank.
Chapter

4

PICmicro® MCU Instruction Sets

Unless otherwise noted, all instructions execute in one instruction cycle.

Parameters
There are a number of parameters that are used with the instructions. The parameters are defined as:
### Low-End Instruction Set

Register Banks are 32 bytes in size in the low-end devices. This makes “Reg” in the range of 0x00 to 0x01F.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Op code letter</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don’t Care</td>
<td>N/A</td>
<td>x</td>
<td>0 to 0xFF</td>
</tr>
<tr>
<td>Byte Constant</td>
<td>k</td>
<td>k</td>
<td>0 to 0xFF</td>
</tr>
<tr>
<td>Register Address</td>
<td>Reg</td>
<td>f</td>
<td>PICmicro® MCU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Architecture Specific</td>
<td></td>
</tr>
<tr>
<td>Destination</td>
<td>d</td>
<td>d</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Selection Bit</td>
<td>Bit</td>
<td>b</td>
<td>0 to 7</td>
</tr>
<tr>
<td>Destination Address</td>
<td>a</td>
<td>a</td>
<td>0 to 0x07FF</td>
</tr>
<tr>
<td>Destination Port</td>
<td>Port</td>
<td>p</td>
<td>PORTA (5) to PORTC (7)</td>
</tr>
<tr>
<td>Description</td>
<td>Instruction</td>
<td>Operation</td>
<td>Op code</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| Add Register Contents to "w" and optionally store result in "w" | addwf Reg, d | #if (d == 1)  
Reg = Reg + w  
else  
w = Reg + w  
endif  
C = (Reg + w)  
> 0x0FF  
Z = ((Reg + w) &  
0x0FF) == 0  
DC = ((Reg & 0x0F)  
+ (w & 0x0F))  
> 0x0F | 0001 11df ffff |
| AND Immediate with "w"                          | andlw k     | w = w & k  
Z = (w & k) == 0                               | 1110 kkkk kkkk |
| AND Register Contents with "w" and Optionally store result in "w" | andwf Reg, d | #if (d == 1)  
Reg = Reg & w  
else  
w = Reg & w  
endif  
Z = (Reg & w) == 0 | 0001 01df ffff |
| Clear the Specified Bit in the Register         | bcf Reg, bit| Reg = Reg &  
(0x0F ^  
(1 << Bit)) | 0100 bbbf ffff |
<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Operation</th>
<th>Op code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set the Specified Bit in the Register</td>
<td>bcf Reg, bit</td>
<td>Reg = Reg</td>
<td>(1 «&lt; Bit)</td>
</tr>
<tr>
<td>Skip if the Specified Bit In the Register is Clear. One Instruction Cycle if Skip not Executed, two if it is</td>
<td>btfsr0 Reg, bit</td>
<td>if ((Reg &amp; (1 «&lt; Bit)) == 0) PC = PC + 1 endif</td>
<td>0110 bbbf ffff</td>
</tr>
<tr>
<td>Skip if the Specified Bit In the Register is Set. One Instruction Cycle if Skip not Executed, two if it is</td>
<td>btfsr0 Reg, bit</td>
<td>if ((Reg &amp; (1 «&lt; Bit)) != 0) PC = PC + 1 endif</td>
<td>0111 bbbf ffff</td>
</tr>
<tr>
<td>Save the Stack Pointer and jump to the Specified Address (two Instruction cycles)</td>
<td>call Address</td>
<td>[SP] = PC SP = SP + 1 PC = ((STATUS &amp; 0x0E0) «&lt; 4) + Address</td>
<td>1001 aaaa aaaa</td>
</tr>
<tr>
<td>Clear the Specified Register</td>
<td>clrf Reg</td>
<td>Reg = 0 Z = 1</td>
<td>0000 011f ffff</td>
</tr>
<tr>
<td>Clear the “w” Register</td>
<td>clrw</td>
<td>w = 0 Z = 1</td>
<td>0000 0100 0000</td>
</tr>
<tr>
<td>Clear the Watchdog Timer’s Counter</td>
<td>clrwdt</td>
<td>WDT = 0 TO = 1 PD = 1</td>
<td>0000 0000 0100</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Code</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>-------</td>
</tr>
</tbody>
</table>
| **Complement the Contents of the Specified Register and optionally store the results in “w”** | `comf Reg, d` | 0010 01df ffff | if `(d == 1)`  
 Reg = Reg ^ 0x0FF  
 else  
 w = Reg ^ 0x0FF  
endif  
Z = (Reg ^ 0x0FF)  
== 0 |
| **Decrement the Contents of the Register and Optionally store the results in “w”** | `decf Reg, d` | 0011 11df ffff | if `(d == 1)`  
 Reg = Reg - 1  
 else  
 w = Reg - 1  
endif  
Z = (Reg - 1) == 0 |
| **Decrement the Contents of the Register and Optionally store the results in “w”** and Skip the next instruction if the results are equal to Zero. Two Instruction Cycles taken if Skip Executed | `decfsz Reg, d` | 0010 11df ffff | if `(d == 1)`  
 Reg = Reg - 1  
 else  
 w = Reg - 1  
endif  
 PC = PC + 1  
 if ((Reg - 1) == 0)  
 PC = PC + 1  
endif |
| **Jump to the Specified Address (two Instruction cycles)** | `goto Address` | 101a aaaa aaaa | if 
PC = `(STATUS & 0x0E0) << 4) + Address` |
<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Operation</th>
<th>Op code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment the Contents of the Register and Optionally store the results in “w”</td>
<td>incf Reg, d</td>
<td>if (d == 1) Reg = Reg + 1 else w = Reg + 1 endif Z = (Reg + 1) == 0</td>
<td>0010 10df ffff</td>
</tr>
<tr>
<td>Increment the Contents of the Register and Optionally store the results in “w” and Skip the next instruction if the results are equal to Zero. Two Instruction Cycles taken if Skip Executed</td>
<td>incfsz Reg, d</td>
<td>if (d == 1) Reg = Reg + 1 else w = Reg + 1 endif PC = PC + 1 endif if ((Reg + 1) == 0) PC = PC + 1 endif</td>
<td>0011 11df ffff</td>
</tr>
<tr>
<td>OR Immediate with “w”</td>
<td>iorlw k</td>
<td>w = w</td>
<td>k Z = (w</td>
</tr>
<tr>
<td>OR Register Contents with “w” and Optionally store result in “w”</td>
<td>iorwf Reg, d</td>
<td>if (d == 1) Reg = Reg</td>
<td>w else w = Reg</td>
</tr>
<tr>
<td>Check Register Contents equal to zero and Optionally store result in &quot;w&quot;</td>
<td>movf Reg, d</td>
<td>if (d == 0) w = Reg endif</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = Reg = 0</td>
<td></td>
</tr>
<tr>
<td>Load &quot;w&quot; with an Immediate value</td>
<td>movlw k</td>
<td>w = k</td>
<td></td>
</tr>
<tr>
<td>Store the value in &quot;w&quot;</td>
<td>movwf Reg</td>
<td>Reg = w</td>
<td></td>
</tr>
<tr>
<td>Waste one Instruction</td>
<td>nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move the contents of &quot;w&quot; into the OPTION Register</td>
<td>option TRIS(Port) = w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resume Execution after Subroutine and Place a constant value in &quot;w&quot; (Two Cycles used)</td>
<td>retlw k</td>
<td>w = k SP = SP - 1 PC = [SP]</td>
<td></td>
</tr>
<tr>
<td>Resume Execution after Subroutine and Place Zero in &quot;w&quot; (Two Cycles used). This is actually a &quot;retlw 0&quot; instruction that MPLAB provides</td>
<td>return</td>
<td>w = 0 SP = SP - 1 PC = [SP]</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Instruction</td>
<td>Operation</td>
<td>Op code</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Rotate the Register left through carry</td>
<td>rlf Reg, d</td>
<td>Temp = C C = (Reg &gt;&gt; 7) &amp; 1 if (d = = 1) Reg = (Reg &lt;&lt; 1) + Temp else w = (Reg &lt;&lt; 1) + Temp endif</td>
<td>0011 01df ffff</td>
</tr>
<tr>
<td>and optionally Save the Result in “w”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate the Register right through carry</td>
<td>rrf Reg, d</td>
<td>Temp = C C = Reg &amp; 1 if (d = = 1) Reg = (Reg &gt;&gt; 1) + (Temp &lt;&lt; 7) else w = (Reg &gt;&gt; 1) + (Temp &lt;&lt; 7) endif</td>
<td>0011 00df ffff</td>
</tr>
<tr>
<td>and optionally Save the Result in “w”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Go into “Standby” Mode (Indeterminate number of cycles used)</td>
<td>sleep</td>
<td>_TO = 1 _PD = 0</td>
<td>0000 0000 0011</td>
</tr>
<tr>
<td>Operation Description</td>
<td>Assembly Code</td>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>---------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>Subtract “w” Register Contents from</td>
<td><code>subwf Reg, d</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register and Optionally store</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result in “w”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (d == 1)</td>
<td></td>
<td>0000 10df ffff</td>
<td></td>
</tr>
<tr>
<td>Reg = Reg +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(w ^ 0x0FF) + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w = Reg +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(w ^ 0x0FF) + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C = (Reg +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(w ^ 0x0FF) + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z = (Reg +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(w ^ 0x0FF) + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp; 0x0FF) == 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC = (Reg &amp; 0x0F)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ ((w ^ 0x0FF) &amp; 0x0F) + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; 0x0F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap the Upper and lower nybbles</td>
<td><code>swapf Reg, d</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>of a Register and Optionally store</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result in “w”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (d == 1)</td>
<td></td>
<td>0011 10df ffff</td>
<td></td>
</tr>
<tr>
<td>Reg = ((Reg &amp; 0x0F0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; 4) +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((Reg &amp; 0x00F)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; 4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w = ((Reg &amp; 0x0F0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; 4) +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((Reg &amp; 0x00F)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; 4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Instruction</td>
<td>Operation</td>
<td>Op code</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------</td>
<td>-------------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Move the contents of &quot;w&quot; into the tri-state control Register of the Port</td>
<td>tris Port</td>
<td>TRIS(Port) = w</td>
<td>0000 0000 0ppp</td>
</tr>
<tr>
<td>XOR Immediate with &quot;w&quot;</td>
<td>xorlw k</td>
<td>w = w ^ k</td>
<td>1111 kkkk kkkk</td>
</tr>
<tr>
<td>XOR Register Contents with &quot;w&quot; and Optionally store result in &quot;w&quot;</td>
<td>xorwf Reg, d</td>
<td>if (d == 1) Reg = Reg ^ w</td>
<td>0001 1odf ffff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else w = Reg ^ w</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = (Reg ^ w) == 0</td>
<td></td>
</tr>
</tbody>
</table>
Mid-Range Instruction Set

Register Banks are 128-bytes in size in the low-end devices. This makes “Reg” in the range of 0 to 0x07F.

<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Operation</th>
<th>Op code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Immediate to “w”</td>
<td>addlw k</td>
<td>w = w + k</td>
<td>11 111x kkkk kkkk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = (w + k) &gt; 0xFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = ((w + k) &amp; 0xFF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = ((w &amp; 0xFF) +</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(k &amp; 0xFF)) &gt; 0xFF</td>
<td></td>
</tr>
<tr>
<td>Add Register Contents to “w”</td>
<td>addwf Reg, d</td>
<td>if (d == 1)</td>
<td>00 0111 dfff ffff</td>
</tr>
<tr>
<td>and optionally store result in “w”</td>
<td></td>
<td>Reg = Reg + w</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>w = Reg + w</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = (Reg + w)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 0xFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = ((Reg + w) &amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xFF) == 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = ((Reg &amp; 0xFF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ (w &amp; 0xFF)) &gt; 0xF</td>
<td></td>
</tr>
<tr>
<td>AND Immediate with “w”</td>
<td>andlw k</td>
<td>w = w &amp; k</td>
<td>11 1001 kkkk kkkk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = (w &amp; k) == 0</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Instruction</td>
<td>Operation</td>
<td>Op code</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| AND Register                                    | \texttt{andwf Reg, d} | if (d == 1) \begin{align*} \text{Reg} &= \text{Reg} \& w \\
| Contents with \texttt{"w"} and Optionally store result in \texttt{"w"} |             | else \begin{align*} w &= \text{Reg} \& w \\
|                                                 |             | Z &= (\text{Reg} \& w) == 0 \end{align*}                                   | 00 0101 dfff ffff |
| Clear the Specified Bit in the Register         | \texttt{bcf Reg, bit} | \text{Reg} = \text{Reg} \& \lnot (0x0FF \ll\, \text{Bit}) \ll\, \text{Bit} | 01 00bb bfff ffff |
| Set the Specified Bit in the Register           | \texttt{bcf Reg, bit} | \text{Reg} = \text{Reg} \ll\, \text{Bit}                                     | 01 01bb bfff ffff |
| Skip if the Specified Bit in the Register is Clear. One Instruction Cycle if Skip not Executed, two if Skip Executed | \texttt{btfsc Reg, bit} | if ((\text{Reg} \ll\, \text{Bit}) == 0) \begin{align*} PC &= PC + 1 \\
<p>|                                                 |             | \text{endif}                                                                 | 01 10bb bfff ffff |</p>
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>btfsc Reg, bit</code></td>
<td>Skip if the Specified Bit in the Register is Set. One Instruction Cycle if Skip not Executed, two if it is.</td>
</tr>
<tr>
<td><code>call Address</code></td>
<td>Save the Stack Pointer and jump to the Specified Address (two Instruction cycles)</td>
</tr>
<tr>
<td><code>clrf Reg</code></td>
<td>Clear the Specified Register</td>
</tr>
<tr>
<td><code>clrw</code></td>
<td>Clear the “w” Register</td>
</tr>
<tr>
<td><code>clrwdt</code></td>
<td>Clear the Watchdog Timer's Counter</td>
</tr>
<tr>
<td><code>comf Reg, d</code></td>
<td>Complement the Contents of the Specified Register and Optionally store the results in “w”</td>
</tr>
</tbody>
</table>

**Assembly Code**

```assembly
btfsc Reg, bit
    if ((Reg & (1 << Bit)) != 0)
        PC = PC + 1
    endif

call Address
    [SP] = PC
    SP = SP - 1
    PC = ((PCLATH << 8) & 0x01800) + Address

clrf Reg
    Reg = 0
    Z = 1

clrw
    w = 0
    Z = 1

clrwdt
    WDT = 0
    _TO = 1
    _PD = 1

comf Reg, d
    if (d == 1)
        Reg = Reg ^ 0xFF
    else
        w = Reg ^ 0xFF
    endif
    Z = (Reg ^ 0xFF) == 0
```

**Hexadecimal Code**

```
01 11bb bfff ffff
10 0aaa aaaa aaaa
00 0001 1fff ffff
00 0001 0xxx xxxx
00 0000 0110 0100
00 1001 dfff ffff
```
<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Operation</th>
<th>Op code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement the Contents of the Register and</td>
<td>decf, d</td>
<td>if (d == 1) Reg = Reg - 1; else w = Reg - 1; endif; Z = (Reg - 1) == 0;</td>
<td>00 0011 dfff ffff</td>
</tr>
<tr>
<td>Optionally store the results in &quot;w&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decrement the Contents of the Register and</td>
<td>decfss, d</td>
<td>if (d == 1) Reg = Reg - 1; else w = Reg - 1; endif; if ((Reg - 1) == 0)</td>
<td>00 1011 dfff ffff</td>
</tr>
<tr>
<td>Optionally store the results in &quot;w&quot; and Skip</td>
<td></td>
<td>PC = PC + 1; endif</td>
<td></td>
</tr>
<tr>
<td>the next instruction if the results are equal to</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero. Two Instruction Cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>taken if skip executed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump to the Specified Address (two Instruction</td>
<td>goto Address</td>
<td>PC = (PCLATH &lt;&lt; 8) &amp; 0x01800 + Address</td>
<td>10 1aaa aaaa aaaa</td>
</tr>
<tr>
<td>cycles)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increment the Contents of the Register and</td>
<td>incf, d</td>
<td>if (d == 1) Reg = Reg + 1; else w = Reg + 1; endif; Z = (Reg + 1) == 0;</td>
<td>00 1010 dfff ffff</td>
</tr>
<tr>
<td>Optionally store the results in &quot;w&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Increment the Contents of the Register and Optionally store the results in "w" and Skip the next instruction if the results are equal to Zero. Two Instruction Cycles taken if Skip Executed.

```
increment Register and store results in 'w'
if (d = 1)
    Reg = Reg + 1
else
    w = Reg + 1
endif
```

```
if ((Reg + 1) = 0)
    PC = PC + 1
endif
```

OR Immediate with "w"
```
OR with Immediate
iorlw k
w = w | k
Z = (w | k) = 0
```

```
OR with Register
iorwf Reg, d
if (d = 1)
    Reg = Reg | w
else
    w = Reg | w
endif
Z = (Reg | w) = 0
```

Check Register Contents equal to zero and Optionally store Register contents in "w"
```
movf Reg, d
if (d = 0)
    w = Reg
endif
Z = Reg = 0
```
<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Operation</th>
<th>Op code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load “w” with an immediate value</td>
<td><code>movlw k</code></td>
<td>( w = k )</td>
<td>( 11 \ 00xx \ kkkk \ kkkk )</td>
</tr>
<tr>
<td>Store the value in “w”</td>
<td><code>movwf Reg</code></td>
<td>( \text{Reg} = w )</td>
<td>( 00 \ 0000 \ 1fff \ fffe )</td>
</tr>
<tr>
<td>Waste one Instruction</td>
<td><code>nop</code></td>
<td></td>
<td>( 00 \ 0000 \ 0xx0 \ 0000 )</td>
</tr>
<tr>
<td>Move the contents of “w” into the OPTION Register. Use of this instruction is not recommended</td>
<td><code>option</code></td>
<td>( \text{TRIS(Port)} = w )</td>
<td>( 00 \ 0000 \ 0110 \ 0010 )</td>
</tr>
<tr>
<td>Resume Execution after Interrupt (Two Cycles used)</td>
<td><code>retfie</code></td>
<td>( \text{GIE} = 1 )</td>
<td>( 00 \ 0000 \ 0000 \ 1001 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{SP} = \text{SP} - 1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{PC} = \text{[SP]} )</td>
<td></td>
</tr>
<tr>
<td>Resume Execution after Subroutine and Place a constant Value in “w” (Two Cycles used)</td>
<td><code>retlw k</code></td>
<td>( w = k )</td>
<td>( 11 \ 01xx \ kkkk \ kkkk )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{SP} = \text{SP} - 1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{PC} = \text{[SP]} )</td>
<td></td>
</tr>
</tbody>
</table>
Resume Execute after Subroutine (Two Cycles used)

```assembly
return
SP = SP - 1
PC = [SP]
```

Rotate the Register left through carry and Optionally Save the Result in "w"

```assembly
rlf Reg, d
Temp = C
C = (Reg >> 7) & 1
if (d == 1)
  Reg = (Reg << 1) + Temp
else
  w = (Reg << 1) + Temp
endif
```

Rotate the Register right through carry and Optionally Save the Result in "w"

```assembly
rrf Reg, d
Temp = C
C = Reg & 1
if (d == 1)
  Reg = (Reg >> 1) + (Temp << 7)
else
  w = (Reg >> 1) + (Temp << 7)
endif
```

Go into “Standby” Mode (Indeterminate number of cycles used)

```assembly
sleep
.TO = 1
._PD = 0
```

```
00 0000 0110 0011
```

```
00 1100 dfff ffff
```

```
00 1100 dfff ffff
```

```
00 0000 0010 0011
```
<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Operation</th>
<th>Op code</th>
</tr>
</thead>
</table>
| Subtract “w” Contents from Immediate and Store the Result In “w” | sublw k             | $w = k + (w \wedge 0x0FF) + 1$  
$C = (k + (w \wedge 0x0FF) + 1) > 0x0FF$  
$Z = ((k + (w \wedge 0x0FF) + 1) \& 0x0FF) = 0$  
$DC = ((k \& 0x0F) + (w \wedge 0x0FF) \& 0x0F) + 1) > 0x0F$ | 11 110x kkkk kkkk |
| Subtract “w” Register Contents from Register and Optionally store Result in “w” | subwf Reg, d        | if (d == 1)  
Reg = Reg + (w \wedge 0x0FF) + 1  
else  
w = Reg + (w \wedge 0x0FF) + 1  
endif  
$C = (Reg + (w \wedge 0x0FF) + 1) > 0x0FF$  
$Z = ((Reg + (w \wedge 0x0FF) + 1) \& 0x0FF) = 0$  
$DC = ((Reg \& 0x0F) + (w \wedge 0x0FF) \& 0x0F) + 1) > 0x0F$ | 00 0010 dff dfff |
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| **Swap the Upper and lower Nybbles of a Register and Optionally store result in “w”** | `swapf Reg, d` | if $(d = 1)$
                  Reg =
                        $((Reg \& 0x0F0) >> 4)$
                        $+ ((Reg \& 0x00F) << 4)$
                  else
                        $w = ((Reg \& 0x0F0) >> 4) + ((Reg \& 0x00F) << 4)$
                  endif
| **Move the contents of “w” into the Tri-state control Register of the Port. Use of this Instruction is not recommended** | `tris Port` | `TRIS(Reg) = w` | 00 0000 0110 0ppp |
| **XOR Immediate with “w”** | `xorlw k` | $w = w \oplus k$
                  $Z = (w \oplus k) = 0$
                  11 1010 kkkk kkkk |
| **XOR Register Contents with “w” and Optionally store result in “w”** | `xorwf Reg, d` | if $(d = 1)$
                  Reg =
                        $Reg \oplus w$
                  else
                        $w = Reg \oplus w$
                  endif
                  $Z = (Reg \oplus w) = 0$
|
PIC17Cxx Instruction Set

The PIC17Cxx's instruction set is very similar to both the low-end and the mid-range instruction sets except for the basic "move" instructions. These instructions are quite a bit different because of the operation of the PIC17Cxx's "primary" register subset of the total 256 possible addresses.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a Constant to the &quot;wreg&quot; and store the Result in &quot;wreg&quot;</td>
<td>addlw Constant</td>
<td>wreg = wreg + Constant if ((wreg &gt; 0) &amp; (Constant &gt; 0)) &amp; (wreg + Constant &gt; 0x07F)) OV = 1 else OV = 0 if ((wreg + Constant) &gt; 0x0FF) C = 1 else C = 0 if (((wreg &amp; 0x0F) + (Constant &amp; 0x0F)) &gt; 0x0F) DC = 1 else DC = 0 if (((wreg + Constant) &amp; 0x0FF) == 0x000) Z = 1 else Z = 0</td>
<td>1011 0001 kkkk kkkk</td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Operation</td>
<td>Bit pattern</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Add &quot;wreg&quot; to the Contents of &quot;Reg&quot; and store the Result According to &quot;d&quot;</td>
<td>addwf Reg, d</td>
<td>if (&quot;d&quot; = -1) &lt;br&gt; wreg = wreg + Reg &lt;br&gt; else &lt;br&gt; Reg = wreg &lt;br&gt; - Reg &lt;br&gt; if (((wreg &gt; 0) &amp; (Reg &gt; 0)) &amp; (wreg + Reg) &gt; 0x07F)) &lt;br&gt; OV = 1 &lt;br&gt; else &lt;br&gt; OV = 0 &lt;br&gt; if ((wreg + Reg) &gt; 0xFF) &lt;br&gt; C = 1 &lt;br&gt; else &lt;br&gt; C = 0 &lt;br&gt; if (((wreg &amp; 0x0F) + (Reg &amp; 0x0F)) &gt; 0x0F) &lt;br&gt; DC = 1 &lt;br&gt; else &lt;br&gt; DC = 0</td>
<td>0000 l1ld ffff ffff</td>
</tr>
</tbody>
</table>
Add "wreg" to the Contents of "Reg" and "C", store the Result According to "d"
Result in "wreg"
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND a Constant to the</td>
<td>andlw</td>
<td>wreg = wreg &amp; Constant</td>
<td>1011 0101 kkkk kkkk</td>
</tr>
<tr>
<td>“wreg” and store the Result in</td>
<td></td>
<td>&amp; Constant</td>
<td></td>
</tr>
<tr>
<td>“wreg”</td>
<td></td>
<td>if ((wreg &amp; Constant)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x000)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 0</td>
<td></td>
</tr>
<tr>
<td>AND “wreg” to the Contents of</td>
<td>andwf</td>
<td>wreg = wreg &amp; Reg &amp; Reg</td>
<td>0000 101d ffff ffff</td>
</tr>
<tr>
<td>“Reg” and store the Result</td>
<td></td>
<td>if (&quot;d&quot; == 1)</td>
<td></td>
</tr>
<tr>
<td>According to “d”</td>
<td></td>
<td>wreg = wreg &amp; Reg</td>
<td></td>
</tr>
<tr>
<td>Result in “wreg”</td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reg = wreg &amp; Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((wreg &amp; Reg)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x000)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 0</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Clear the Specified Bit in &quot;Reg&quot;</td>
<td><code>bcf Reg, Bit</code></td>
<td>Reg = Reg &amp; (0xFFFF ^ (1 &lt;&lt; Bit))</td>
<td></td>
</tr>
<tr>
<td>Set the Specified Bit in &quot;Reg&quot;</td>
<td><code>bsf Reg, Bit</code></td>
<td>Reg = Reg &amp; (1 &lt;&lt; Bit)</td>
<td></td>
</tr>
<tr>
<td>Test the Specified Bit in &quot;Reg&quot;</td>
<td><code>btfsc Reg, Bit</code></td>
<td>if ((Reg &amp; (1 &lt;&lt; Bit)) == 0) PC = PC + 1</td>
<td></td>
</tr>
<tr>
<td>Test the Specified Bit in &quot;Reg&quot;</td>
<td><code>btfss Reg, Bit</code></td>
<td>if ((Reg &amp; (1 &lt;&lt; Bit)) ^= 0)</td>
<td></td>
</tr>
<tr>
<td>Toggle the Specified Bit in &quot;Reg&quot;</td>
<td><code>btg Reg, Bit</code></td>
<td>Reg = Reg &amp; (1 &lt;&lt; Bit)</td>
<td></td>
</tr>
<tr>
<td>Call the &quot;Label&quot; Address</td>
<td><code>call Label</code></td>
<td>PUSH (PC) PCLATH = PC (15:13) + Label (12:8) PCL = Label (7:0)</td>
<td></td>
</tr>
<tr>
<td>Clear the Specified Register</td>
<td><code>clrf Reg, s</code></td>
<td>Reg = 0 if (s = 0) wreg = 0</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Operation</td>
<td>Bit pattern</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Clear the Watchdog</td>
<td>clrwdt</td>
<td>WDT = 0</td>
<td>0000 0000 0000 0100</td>
</tr>
<tr>
<td>Register and STATUS flags</td>
<td>clrwdt</td>
<td>WDT Postscaler = 0 _TO = 1 _PD = 1</td>
<td></td>
</tr>
<tr>
<td>Complement the Contents of the Specified Register</td>
<td>comf Reg, d</td>
<td>if (&quot;d&quot; == 0) wreg = Reg ^ 0xFF else Reg = Reg ^ wreg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((Reg ^ 0xFF) == 0x000) Z = 1 else Z = 0</td>
<td></td>
</tr>
<tr>
<td>Compare the Specified Register with wreg and skip if the Register = Wreg. If Skip Executed.</td>
<td>cphseq Reg</td>
<td>if ((Reg - wreg) == 0) PC = PC + 1</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>Compare the Specified Register with ( wreg ) and skip if the Register &gt; ( wreg ). If Skip Executed, two Instruction Cycles Executed Else one Cycle</td>
<td><code>cplgt Reg</code></td>
<td><code>0011 0010 ffff ffff</code></td>
<td></td>
</tr>
<tr>
<td>Compare the Specified Register with ( wreg ) and skip if the Register &lt; ( wreg ). If Skip Executed, two Instruction Cycles Executed Else one Cycle</td>
<td><code>cplslt Reg</code></td>
<td><code>0011 0000 ffff ffff</code></td>
<td></td>
</tr>
<tr>
<td>Do a Decimal Adjust after Addition of Two BCD Values</td>
<td><code>daw Reg, s</code></td>
<td><code>0010 111s ffff ffff</code></td>
<td></td>
</tr>
</tbody>
</table>

\( s = 0 \) if \((wreg \& 0x0F) > 9\) else \((wreg \& 0x0F) + 0x010\)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement the Contents of the Specified Register</td>
<td>decf Reg, d</td>
<td>if (*d = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>wreg = Reg − 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reg = Reg − 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((Reg = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b = 0x000)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((Reg &gt; 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; ((Reg = 1) &lt; 0x080))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OV = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OV = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (((Reg &amp; 0x00F) − 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; 0x080) = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((Reg = 1) &lt; 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = 1</td>
<td></td>
</tr>
</tbody>
</table>
Decrement the Contents of the Specified Register and skip the Next Instruction if Result $= 0$. If Skip Executed, two Instruction Cycles Executed Else one Cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembler Code</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>decfsz Reg, d</td>
<td>Decrement the Contents of the Specified Register and skip the Next Instruction if Result $= 0$. If Skip Executed, two Instruction Cycles Executed Else one Cycle</td>
<td>if (“d” $= 0$) ( \text{wreg} = \text{Reg} - 1 ) else ( \text{Reg} = \text{Reg} - 1 ) if ((Reg $= 0$) ( \text{PC} = \text{PC} + 1 )) ( \text{Result} \neq 0 ). If Skip Executed, two Instruction Cycles Executed Else one Cycle</td>
<td>0001 011d ffff ffff</td>
</tr>
</tbody>
</table>

Decrement the Contents of the Specified Register and skip the Next Instruction if Result $= 0$. If Skip Executed, two Instruction Cycles Executed Else one Cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembler Code</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcfsnz Reg, d</td>
<td>Decrement the Contents of the Specified Register and skip the Next Instruction if Result $= 0$. If Skip Executed, two Instruction Cycles Executed Else one Cycle</td>
<td>if (“d” $= 0$) ( \text{wreg} = \text{Reg} - 1 ) else ( \text{Reg} = \text{Reg} - 1 ) if ((Reg $= 0$) ( \text{PC} = \text{PC} + 1 )) ( \text{Result} \neq 0 ). If Skip Executed, two Instruction Cycles Executed Else one Cycle</td>
<td>0010 011d ffff ffff</td>
</tr>
</tbody>
</table>

Goto the “Label” Address. Two Instruction Cycles

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembler Code</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto Label</td>
<td>Goto the “Label” Address. Two Instruction Cycles</td>
<td>( \text{PCLATH} = \text{PC} (15:13) ) ( \text{Label} (12:8) ) ( \text{Label} (7:0) )</td>
<td>1101 kkkk kkkk kkkk</td>
</tr>
</tbody>
</table>

\( \text{PC} (15:13) \) \( \text{Label} (12:8) \) \( \text{Label} (7:0) \)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment the Contents of the Specified Register</td>
<td>incf Reg, d</td>
<td>if (&quot;d&quot; == 0) wreg = Reg + 1 else Reg = Reg + 1 if (Reg + 1) == 0x000 Z = 1 else Z = 0 if (((Reg &gt; 0) &amp; ((Reg + 1) &gt; 0x07F)) OV = 1 else OV = 0 if (((Reg &amp; 0x00F) + 1) &amp; 0x010) DC = 1 else DC = 0 if ((Reg + 1) == 0x0100) C = 1 else C = 0</td>
<td>0001 010d ffff ffff</td>
</tr>
</tbody>
</table>
Increment the Contents of the Specified Register and skip the Next Instruction if Result == 0. If Skip Executed, two Instruction Cycles Executed Else one Cycle

```
incfsz Reg, d
```

```
if ("d" == 0) wreg = Reg + 1
else Reg = Reg + 1
if ((Reg + 1) == 0x000) PC = PC + 1
```

```
infsnz Reg, d
```

```
if ("d" == 0) wreg = Reg + 1
else Reg = Reg + 1
if ((Reg + 1) != 0x000) PC = PC + 1
```

OR a Constant to the "wreg" and store the Result in "wreg"

```
iorlw Constant
```

```
wreg = wreg | Constant
if ((wreg | Constant) == 0x000) Z = 1
else Z = 0
```

```
0001 111d ffff ffff
```

```
0010 010d ffff ffff
```

```
1011 0011 kkkk kkkk
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR “wreg” to the Contents of “Reg” and store the Result According to “d” Result in “wreg”</td>
<td>lorwf Reg, d</td>
<td>if (&quot;d&quot; = = 1) wreg = wreg</td>
<td>0000 100d ffff ffff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l Reg else Reg = wreg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>l Reg if ((wreg</td>
<td>Reg) = 0x000) Z = 1 else Z = 0</td>
</tr>
<tr>
<td>Call the “Label” Address Using PCLATH and the Least Significant Eight bits of “Label”. Two Instruction Cycles</td>
<td>lcall Label</td>
<td>PUSH (PC)</td>
<td>1011 0111 kkkk kkkk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCL = Label (7:0)</td>
<td></td>
</tr>
<tr>
<td>Move data from 256 Address Register Data to Primary Register Set</td>
<td>movfp Reg, p</td>
<td>p = Reg</td>
<td>011p pppp ffff ffff</td>
</tr>
<tr>
<td>Move Constant into low Nybble of BSR</td>
<td>movlb Constant</td>
<td>BSR (3:0) = Constant</td>
<td>1011 1000 0000 kkkk</td>
</tr>
<tr>
<td>Move Constant into high Nybble of BSR</td>
<td>movir Constant</td>
<td>BSR (7:4) = Constant</td>
<td>1011 1010 kkkk 0000</td>
</tr>
</tbody>
</table>
Move Constant into wreg
movlw Constant

Move data from Primary
Register Data to 256
Address Register Set
movwf p, Reg
if (p == 0)
   Z = 1
else
   Z = 0

Move contents of "wreg" into "Reg"
movwf Reg

Multiply Constant by "wreg"
mullw Constant

Multiply Register by "wreg"
mulwf Reg
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negate the Contents of “wreg” and Optionally</td>
<td>negw Reg, a</td>
<td>if (s == 0)</td>
<td>0010 110s ffff ffff</td>
</tr>
<tr>
<td>store the Result</td>
<td></td>
<td>Reg = ¬wreg</td>
<td></td>
</tr>
<tr>
<td>in a Register</td>
<td></td>
<td>wreg = ¬wreg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (wreg &lt; 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; (¬wreg &lt; 0x080))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OV = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OV = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (¬wreg &gt; 0x0FF)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (¬wreg &amp; 0x0F)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 0x0F</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (¬wreg == 0x000)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 0</td>
<td></td>
</tr>
<tr>
<td>Do Nothing for one Instruction Cycle</td>
<td>nop</td>
<td></td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>Return from Interrupt Handler. Two Instruction</td>
<td>retfie</td>
<td>PC = POP ( )</td>
<td>0000 0000 0000 0101</td>
</tr>
<tr>
<td>Cycles</td>
<td></td>
<td>GLTIND = 0</td>
<td></td>
</tr>
<tr>
<td>Instruction Type</td>
<td>Description</td>
<td>Assembly Code</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Return from Subroutine</td>
<td>with new value in wreg.</td>
<td><code>retlw Constant wreg = Constant PC = POP()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two Instruction Cycles</td>
<td><code>1011 0110 kkkk kkkk</code></td>
<td></td>
</tr>
<tr>
<td>Return from Subroutine.</td>
<td>Two Instruction Cycles</td>
<td><code>return PC = POP()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>0000 0000 0000 0010</code></td>
<td></td>
</tr>
<tr>
<td>Rotate Left Through the Carry Flag</td>
<td></td>
<td><code>rlcf Reg, d</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (d == 0)</td>
<td><code>if (d == 0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wreg (7:1) = Reg (6:0)</td>
<td><code>wreg (7:1) = Reg (6:0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wreg (0) = C</td>
<td><code>wreg (0) = C</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C = Reg (7)</td>
<td><code>C = Reg (7)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td><code>else</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg (7:1) = Reg (6:0)</td>
<td><code>Reg (7:1) = Reg (6:0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg (0) = C</td>
<td><code>Reg (0) = C</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C = Reg (7)</td>
<td><code>C = Reg (7)</code></td>
<td></td>
</tr>
<tr>
<td>Rotate Left</td>
<td></td>
<td><code>rlcnf Reg, d</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (d == 0)</td>
<td><code>if (d == 0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wreg (7:1) = Reg (6:0)</td>
<td><code>wreg (7:1) = Reg (6:0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wreg (0) = Reg (7)</td>
<td><code>wreg (0) = Reg (7)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td><code>else</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg (7:1) = Reg (6:0)</td>
<td><code>Reg (7:1) = Reg (6:0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg (0) = Reg (7)</td>
<td><code>Reg (0) = Reg (7)</code></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Operation</td>
<td>Bit pattern</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Rotate Right Through the</td>
<td>rrcf Reg, d</td>
<td>if (d == 0) wreg (6:0) = Reg (7:1) wreg (7) = C C = Reg (0) else Reg (6:0) = Reg (7:1) Reg (7) = C C = Reg (0)</td>
<td>0001 100d ffff ffff</td>
</tr>
<tr>
<td>Carry Flag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate Right</td>
<td>rrcnf Reg, d</td>
<td>if (d == 0) wreg (6:0) = Reg (7:1) wreg (7) = Reg (0) else Reg (6:0) = Reg (7:1) Reg (7) = Reg (0)</td>
<td>0010 000d ffff ffff</td>
</tr>
<tr>
<td>Set the Specified Register</td>
<td>setf Reg, s</td>
<td>Reg = 0x0FF if (s == 0) wreg = 0x0FF</td>
<td>0010 101s ffff ffff</td>
</tr>
<tr>
<td>and Optionally “wreg”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>Sleep</td>
<td>Put the PICmicro® MCU in a “Power Down” State</td>
<td><code>WDT = 0</code>&lt;br&gt;<code>WDT Postscaler = 0</code>&lt;br&gt;<code>TO = 1</code>&lt;br&gt;<code>PD = 0</code>&lt;br&gt;Power Down</td>
<td></td>
</tr>
</tbody>
</table>
| Sublw  | Subtract “wreg” from a Constant and store the Result in “wreg” | `wreg = Constant`<br>`- wreg`<br>`if ((wreg < 0)`<br>`& (Constant < 0))`<br>`& ((Constant`<br>`- wreg)`<br>`< 0x00))`<br>`OV = 1`<br>`else`<br>`OV = 0`<br>`if ((Constant`<br>`- wreg) > 0x0FF)`<br>`C = 1`<br>`else`<br>`C = 0`<br>`else`<br>`if (((Constant & 0x0F)`<br>`- wreg & 0x0F))`<br>`> 0x0F)`<br>`DC = 1`<br>`else`<br>`DC = 0`<br>`else`<br>`if (((Constant`<br>`- wreg) & 0x0FF)`<br>`== 0x00)`)`<br>`else`<br>`else`<br>`else`<br>`else`| `0000 0000 0000 0011`<br>`1011 0010 kkkk kkkk`
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
</table>
| Subtract “wreg” from the
Contents of “Reg” and
store the Result According
to “d” Result in “wreg” | subwf Reg, d | if (“d” == 1)
  wreg = Reg –
  wreg
  else
  Reg = Reg
  – wreg
  if (((wreg < 0) & (Reg < 0))
  & ((Reg – wreg) < 0x080))
  OV = 1
  else
  OV = 0
  if ((Reg – wreg) > 0x0FF)
  C = 1
  else
  C = 0
  if (((Reg & 0x0F) –
  (wreg & 0x0F)) > 0x0F)
  DC = 1 |
| Z = 1
else
Z = 0 | 0000 010d ffff ffff |
else
  DC = 0
  if (((Reg -
      wreg) & 0x0FF)
    Z = 0
  else
    Z = 0

Subtract "wreg" from the
Contents of "Reg" and "C",
store the Result According
to "d" Result in "wreg"
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>(wreg &amp; 0x0F) - (Reg &amp; 0x0FF)</td>
<td></td>
<td>if (((Reg - wreg) &amp; 0x00F) == 0x000) else if (((Reg &amp; 0x0FF) &gt;&gt; 4) else</td>
<td>0001 110d ffff ffff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reg = ((Reg &amp; 0x00F) &lt;&lt; 4) + ((Reg &amp; 0x0F0) &gt;&gt; 4) else Reg = (Reg &amp; 0x0FF)</td>
<td></td>
</tr>
<tr>
<td>Swap the Contents of &quot;Reg&quot; and store the Result According to &quot;d&quot;</td>
<td>swapf Reg, d</td>
<td>if (&quot;d&quot; = 1)wreg = ((Reg &amp; 0x0FF) &lt;&lt; 4) + ((Reg &amp; 0x0F0) &gt;&gt; 4) else Reg = (Reg &amp; 0x0FF)</td>
<td></td>
</tr>
</tbody>
</table>

"Swap the Contents of "Reg" and store the Result According to "d" Result in "wreg""
### Read the Contents of the Table Pointer

- **tabird t, i, f**
  - if $t = 1$
    - $f = \text{TBLATH}$
    - $\text{TBLPTR} = \text{ProgMem}($TBLPTR$)$
    - if $i = 1$
      - $\text{TBLPTR} = \text{TBLPTR} + 1$
  - else
    - $f = \text{TBLATH}$
    - $\text{TBLAT} = \text{ProgMem}($TBLPTR$)$
    - if $i = 1$
      - $\text{TBLPTR} = \text{TBLPTR} + 1$

### Read Program Memory into the Table Pointer

Two or three Instruction Cycles

### Write new Contents of the Program Memory from the Table Pointer

If the Destination is Internal EPROM, the Instruction Does not End until an Interrupt.

- **tablwt t, i, f**
  - if $t = 0$
    - $\text{TBLATL} = f$
    - $\text{TBLAT} = f$
  - else
    - $\text{TBLATH} = f$
    - $\text{TBLAT} = \text{ProgMem}($TBLPTR$)$
    - $\text{TBLPTR} = \text{TBLPTR} + 1$
  - if $i = 1$
    - $\text{TBLPTR} = \text{TBLPTR} + 1$

### Read the Contents of the Table Pointer

- **tird t, f**
  - if $t = 1$
    - $f = \text{TBLATH}$
    - $\text{TBLATL} = f$
  - else
    - $f = \text{TBLATH}$

### Write the Contents of the Register into the Table Pointer

- **tlwt t, f**
  - if $t = 1$
    - $f = \text{TBLATH}$
    - $\text{TBLATL} = f$
  - else
    - $f = \text{TBLATL}$
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare the Specified Register zero and skip if the Register ( \neq 0 ). One Instruction Cycle if Skip not Executed, Two if it is</td>
<td>tstfssz  Reg</td>
<td>if (Reg ( = 0 )) ( \text{PC} = \text{PC} + 1 )</td>
<td>(0011 \ 0011 \ ffff \ ffff)</td>
</tr>
<tr>
<td>XOR a Constant to the “wreg” and store the Result in “wreg”</td>
<td>xorlw    Constant</td>
<td>wreg = wreg ^ Constant if ((wreg ^ Constant) ( = 0) (0000 )) (Z = 1) else (Z = 0)</td>
<td>(1011 \ 0100 \ kkkk \ kkkk)</td>
</tr>
<tr>
<td>XOR “wreg” to the Contents of “Reg” and store the Result According to “d” Result in “wreg”</td>
<td>xorwf    Reg, d</td>
<td>if (“d” ( = 1)) wreg = wreg ^ Reg else Reg = wreg ^ Reg if ((wreg ^ Reg) ( = 0) (0000 )) (Z = 1) else (Z = 0)</td>
<td>(0000 \ 110d \ ffff \ ffff)</td>
</tr>
</tbody>
</table>
# PIC18Cxx Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
</table>
| Add a Constant to the “wreg” and store the Result in “wreg” | `addlw Constant` | `wreg = wreg + Constant`<br>
if `(wreg > 0)`<br>
& `(Constant > 0)`<br>
& `(wreg + Constant) > 0x07F)`<br>
`OV = 1`<br>
else<br>
`OV = 0`<br>
if `(wreg + Constant) > 0xFF`<br>
`C = 1`<br>
else<br>
`C = 0`<br>
if `(wreg + Constant) & 0x80 != 0`<br>
`N = 1`<br>
else<br>
`N = 0`<br>
if `((wreg & 0xF) + (Constant & 0xF)) > 0x0F)`<br>
`DC = 1`<br>
else | 0000 1111 kkkk kkkk |
Add "wreg" to the Contents of "Reg" and store the Result. According to "d" Result in "wreg". If "a" is set then BSR used for Reg. else Access Bank is used.

Add "wreg" to the Contents of "Reg" and store the Result. According to "d" Result in "wreg". If "a" is set then BSR used for Reg. else Access Bank is used.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>addw Reg, d, a</td>
<td>addw Reg, d, a</td>
<td>if (&quot;d&quot; == 1) wreg = wreg + Reg; else Reg = wreg + Reg; if ((wreg &amp; Reg &gt; 0) &amp; (Reg &gt; 0)) &amp; ((wreg + Reg) &gt; 0x07F)) OV = 1; else OV = 0; if ((wreg + Reg) &amp; 0x080) != 0) N = 1; else N = 0</td>
<td>0010 01da ffff ffff</td>
</tr>
</tbody>
</table>
Add "wreg" to the Contents of "Reg", set the elseResult Reg according to the Result of "wreg", store the elseResult Reg in a set if "d" & ((wreg & (Reg & 0x0F)) > 0x0FF) Then Reg is set & ((wreg & 0x0FF) = 0x000)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>in the BSR Bank else, Reg is in the Access Bank</td>
<td></td>
<td>(OV = 1) (OV = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ((\text{wreg} + \text{Reg} + \text{C}) &amp; 0x080) != 0)</td>
<td>(N = 1) (N = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ((\text{wreg} + \text{Reg} + \text{C}) &gt; 0x0FF)</td>
<td>(C = 1) (C = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (((\text{wreg} &amp; 0x0F) + (\text{Reg} &amp; 0x0F) + \text{C}) &gt; 0x0F)</td>
<td>(DC = 1) (DC = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (((\text{wreg} + \text{Reg} + \text{C}) &amp; 0x0FF) == 0x000)</td>
<td>(Z = 1) (Z = 0)</td>
<td></td>
</tr>
</tbody>
</table>
AND a Constant to the "wreg" and store the Result in "wreg"

```
andlw Constant
```

```
wreg = wreg & Constant
if ((wreg & Constant) == 0x000)
   Z = 1
else
   Z = 0
if (((wreg & Constant) & 0x080) != 0)
   N = 1
else
   N = 0
```

```
andwf Reg, d, a
```

```
if ("d" == 1)
   wreg = wreg & Reg
else
   Reg = wreg & Reg
if ((wreg & Reg) == 0x000)
   Z = 1
else
   Z = 0
if (((wreg & Reg) & 0x080) != 0)
   N = 1
else
   N = 0
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch if</td>
<td>bc Label</td>
<td>if (C == 1)</td>
<td>1110 0010 kkkk kkkk</td>
</tr>
<tr>
<td>the Carry</td>
<td></td>
<td>PC = PC + 2 + Label</td>
<td></td>
</tr>
<tr>
<td>Label is Set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Label is a Two's Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset: One Instruction Cycle if Branch Not Executed, Two if Branch Executed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear the Specified Bit in &quot;Reg&quot;</td>
<td>bcf Reg, Bit, a</td>
<td>Reg = Reg &amp; (0x0FF ^ (1 &lt;&lt; Bit))</td>
<td>1001 bbba ffff ffff</td>
</tr>
<tr>
<td>If &quot;a&quot; is set then Reg is in the BSR Bank else, Reg is in the Access Bank</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch if bn Label
Negative Flag is Set.
Label is a Two's Complement Offset. One Instruction Cycle if Branch Not Executed,
Two if Branch Executed.

Branch if bnc Label
CARRY Flag is Reset.
Label is a Two's Complement Offset. One Instruction Cycle if Branch Not Executed,
Two if Branch Executed.

if (N = 1) 1110 0110 kkkk kkkk
PC = PC + 2 + Label

if (C = 0) 1110 0011 kkkk kkkk
PC = PC + 2 + Label
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch if bnn Label</td>
<td>1110</td>
<td>if (N == 0)</td>
<td>0111  kkkk kkkk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + 2 + Label</td>
<td></td>
</tr>
<tr>
<td>Negative Flag is Reset. Label is a Two's Complement Offset One Instruction Cycle if Branch Not Executed, Two if Branch Executed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch if bnov Label</td>
<td>1110</td>
<td>if (OV == 0)</td>
<td>0101  kkkk kkkk</td>
</tr>
<tr>
<td>Overflow Flag is Reset. Label is a Two's Complement Offset One Instruction Cycle if Branch Not Executed, Two if Branch Executed.</td>
<td></td>
<td>PC = PC + 2 + Label</td>
<td></td>
</tr>
</tbody>
</table>
Branch if
the Zero
Flag is Reset.
Label is a
Two’s
Complement
Offset. One
Instruction
Cycle if Branch
Not Executed,
Two if Branch
Executed.

Branch if
Overflow
Flag is Set.
Label is a
Two’s
Complement
Offset. One
Instruction
Cycle if Branch
Not Executed,
Two if Branch
Executed.

bnz Label if \( Z = 0 \)  
\[ PC = PC + 2 \]  
Label

1110 0001 kkkk kkkk

bov Label if \( OV = 1 \)  
\[ PC = PC + 2 \]  
Label

1110 0100 kkkk kkkk
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>bra Label</td>
<td>PC = PC + 2 + Label</td>
<td>1110 6kkk kkkk kkkk</td>
</tr>
<tr>
<td>Always. Label is a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two’s Complement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset. Two</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set the Specified Bit in “Reg”.</td>
<td>bsf Reg, Bit, a</td>
<td>Reg = Reg</td>
<td>1 &amp; (1 &lt;&lt; Bit)</td>
</tr>
<tr>
<td>If “a” is set</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>then Reg is in the BSR Bank, else Reg is in the Access Bank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test the Specified Bit in “Reg” and skip if Clear.</td>
<td>btfsc Reg, Bit, a</td>
<td>if ((Reg &amp; (1 &lt;&lt; Bit)) = 0)</td>
<td>PC = NextIns</td>
</tr>
<tr>
<td>If “a” is set then</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the BSR is used for Reg, else the Access Bank is Used.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One Instruction Cycle if Skip Not Executed, Two if Skip Executed.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Test the Specified Bit in "Reg" and skip if Set. If "a" is set then the BSR is used for Reg, else the Access Bank is used. One Instruction Cycle if Skip Not Executed, Two if Skip Executed.

*btfss Reg, Bit, a*

if ((Reg & (1 << Bit)) ! = 0)
PC = NextIns

*btg Reg, Bit, a*

Reg = Reg ^ (1 << Bit)

1010 bbba ffff ffff

0111 bbba ffff ffff
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch if the Zero Flag is Set.</td>
<td>bz Label</td>
<td>if (Z = = 1) PC = PC + 2 + Label</td>
<td>1110 0000 kkkk kkkk</td>
</tr>
<tr>
<td>Label is a Two's Complement Offet. One Instruction Cycle if Branch Not Executed, Two if Branch Executed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call the 20-Bit &quot;Label&quot; Address. If &quot;s&quot; is set, Save the Context Registers, Two Instruction Cycles</td>
<td>call Label,s</td>
<td>PUSH( PC ) if (s = = 1) PUSH (W, STATUS BSR) PC = Label</td>
<td>1110 110s kkkk kkkk</td>
</tr>
</tbody>
</table>

|                    |                  |                                  | 1111 kkkk kkkk kkkk  |
Clear the
Specified
Register.
If "a" is set
then the BSR
is used for
Reg, else the
Access Bank
is used

Clear the
Watchdog
Register and
STATUS flags

Complement
the Contents
of the
Specified
Register.
If "a" is set
then the BSR
is used for
Reg, else the
Access Bank
is used

clr Reg, a
Reg = 0
Z = 1
0110 101a ffff ffff

clrwdt
WDT = 0
WDT Postscaler = 0
_TO = 1
_PD = 1
0000 0000 0000 0100

comf Reg, d, a
if ("d" == 0)
wreg = Reg ^
0x0FF
else
Reg = Reg ^
if ((Reg ^ 0x0FF) == 0x000)
Z = 1
else
Z = 0
if ((Reg ^ 0x0FF) & 0x080) != 0)
N = 1
else
N = 0
0001 11da ffff ffff
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare the Specified Register with wreg and skip if the Register == Wreg. If &quot;a&quot; is set then the BSR is used for Reg, else the Access Bank is used. One Instruction Cycle if Skip Not Executed, Two if Skip</td>
<td>cpfseq Reg, a</td>
<td>if ((Reg - wreg) == 0) PC = NextIns</td>
<td>0110 001a ffff ffff</td>
</tr>
</tbody>
</table>
Executed.

Compare the Specified Register with wreg and skip if the Register > wreg. If "a" is set then the BSR is used for Reg, else the Access Bank is used. One Instruction Cycle if Skip Not Executed, Two if Skip Executed.

\[
\text{cplsgt Reg, a} \quad \text{if } ((\text{Reg} - \text{wreg}) > 0) \quad \text{PC} = \text{NextIns}
\]
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>cpfslt</td>
<td>Reg, a if ((Reg - wreg) &lt; 0) PC = NextIns</td>
<td>0110 0000 a ffff ffff</td>
</tr>
<tr>
<td>Specified</td>
<td></td>
<td>Register with wreg and skip</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td></td>
<td>if the Register &lt; wreg</td>
<td></td>
</tr>
<tr>
<td>with</td>
<td></td>
<td>If &quot;a&quot; is set then the BSR is used for</td>
<td></td>
</tr>
<tr>
<td>wreg</td>
<td></td>
<td>Reg, else the Access Bank is used. One</td>
<td></td>
</tr>
<tr>
<td>and skip</td>
<td></td>
<td>Instruction Cycle if Skip</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not Executed, Two if Skip Executed</td>
<td></td>
</tr>
<tr>
<td>Do a Decimal</td>
<td>daw</td>
<td>if ((wreg &amp; 0x0F) &gt; 9)</td>
<td>0000 0000 0000 0111</td>
</tr>
<tr>
<td>Adjust after</td>
<td></td>
<td>wreg = (wreg &amp; 0x0F) + 0x010</td>
<td></td>
</tr>
<tr>
<td>Addition of</td>
<td></td>
<td>Two BCD Values</td>
<td></td>
</tr>
</tbody>
</table>

Note: The bit pattern 0110 0000 a ffff ffff represents the instruction format for the compare and skip operation.
Decrement the Contents of the Specified Register.
If "a" is set then Reg is in BSR Bank else Access Bank is used.

```
def decf Reg, d, a:
    if ("d" == 0):
        wreg = Reg - 1
    else:
        Reg = Reg - 1
    if ((Reg - 1) == 0x000):
        Z = 1
    else:
        Z = 0
    if ((Reg > 0) & ((Reg - 1) < 0x080)):
        OV = 1
    else:
        OV = 0
    if ((Reg - 1) & 0x080) != 0):
        N = 1
    else:
        N = 0
    if ((Reg & 0x00F) - 1) & 0x080) != 0):
        DC = 0
    else:
        DC = 1
    if ((Reg - 1) < 0):
        C = 0
    else:
        C = 1
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement the Contents of the Specified Register and skip the Next Instruction if Result == 0. If “a” is set then the BSR is used for Reg, else the Access Bank is used. One Instruction Cycle if Skip Not Executed, Two if Skip Executed</td>
<td>decfsz Reg,d,a</td>
<td>if (“d” == 0) wreg = Reg - 1 else Reg = Reg - 1 if ((Reg - 1) == 0x000) PC = NextIns</td>
<td>0010 11da ffff ffff</td>
</tr>
</tbody>
</table>
Decrement the Contents of the Specified Register and skip the Next Instruction if Result != 0. If "a" is set then the BSR is used for Reg, else the Access Bank is used. One Instruction Cycle if Skip Not Executed, Two if Skip Executed.

Goto the 20-Bit "Label" Address. Two Instruction Cycles.

dcfsnz Reg.d,a  
if ("d" == 0)  
  wreg = Reg - 1  
else  
  Reg = Reg - 1  
if ((Reg - 1)  
  l= 0x000)  
  PC = NextIns

goto Label  
PC = Label  
1110 1111 kkkk kkkk  
1111 kkkk kkkk kkkk
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment</td>
<td>incf</td>
<td>if (&quot;d&quot; == 0)</td>
<td>0010 10da ffff ffff</td>
</tr>
<tr>
<td>the Contents</td>
<td>Reg, d, a</td>
<td>wreg = Reg + 1</td>
<td></td>
</tr>
<tr>
<td>of the</td>
<td>else</td>
<td>Reg = Reg + 1</td>
<td></td>
</tr>
<tr>
<td>Specified</td>
<td></td>
<td>if (Reg + 1)</td>
<td></td>
</tr>
<tr>
<td>Register.</td>
<td></td>
<td>== 0x000</td>
<td></td>
</tr>
<tr>
<td>If &quot;a&quot; is set</td>
<td></td>
<td>Z = 1</td>
<td></td>
</tr>
<tr>
<td>then Reg is</td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>in the BSR</td>
<td></td>
<td>Z = 0</td>
<td></td>
</tr>
<tr>
<td>Bank else</td>
<td></td>
<td>if ((Reg &gt; 0)</td>
<td></td>
</tr>
<tr>
<td>Access Bank</td>
<td></td>
<td>&amp; ((Reg + 1)</td>
<td></td>
</tr>
<tr>
<td>is used.</td>
<td></td>
<td>&gt; 0x07F)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OV = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OV = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((Reg + 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; 0x080) != 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((Reg &amp; 0x00F) + 1) &amp; 0x010) != 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ((Reg + 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>== 0x0100)</td>
<td></td>
</tr>
</tbody>
</table>
If "a" is set then the BSR is used for Reg, else the Access Bank is used. One Instruction Cycle if Skip Not Executed, Two if Skip Executed.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment</td>
<td>infsz Reg,d,a</td>
<td>if (&quot;d&quot; == 0)</td>
<td>0100 10da ffff ffff</td>
</tr>
<tr>
<td>of the Specified Register</td>
<td></td>
<td>wreg = Reg + 1</td>
<td></td>
</tr>
<tr>
<td>and skip the Next Instruction if</td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>Result != 0. If &quot;a&quot; is set then</td>
<td></td>
<td>Reg = Reg + 1</td>
<td></td>
</tr>
<tr>
<td>the BSR is used for Reg, else</td>
<td></td>
<td>if ((Reg + 1) &gt;&gt;= 0x000)</td>
<td></td>
</tr>
<tr>
<td>the Access Bank is used. One</td>
<td></td>
<td>PC = NextIns</td>
<td></td>
</tr>
<tr>
<td>Instruction Cycle if Skip Not</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executed, Two if Skip Executed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OR a Constant to the "wreg" and store the Result in "wreg"

**iorlw Constant**

```assembly
iorlw Constant
wreg = wreg
if ((wreg | Constant) == 0x000)
Z = 1
else
Z = 0
if ((wreg | Constant) & 0x080) != 0)
N = 1
else
N = 0
```

OR "wreg" to the Contents of "Reg" and store the Result

According to "d" Result in "wreg", if "a" is set then Reg is in the BSR Bank, else it is in the Access Bank

**iorwf Reg,d,a**

```assembly
iorwf Reg,d,a
if ("d" == 1)
if ((wreg | Reg) == 0x000)
Z = 1
else
Z = 0
if ((wreg | Reg) & 0x080) != 0)
N = 1
else
N = 0
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load the Specified FSR Register</td>
<td><code>lfsr f, Const</code></td>
<td><code>FSR(f) = Const</code></td>
<td><code>1110 1110 00ff kkkk</code></td>
</tr>
<tr>
<td>Move data from 256 Address</td>
<td><code>movf Reg, d, a</code></td>
<td>if <code>(d == 0)</code></td>
<td><code>0101 00da ffff ffff</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>wreg = Reg</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if <code>(Reg == 0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Z = 1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>else</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Z = 0</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if <code>(Reg &amp; 0x080)</code></td>
<td>if <code>(Reg &amp; 0x080)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>! = 0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>N = 1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>else</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>N = 0</code></td>
<td></td>
</tr>
<tr>
<td>Move Contents of the Source Register into the Destination Register. The Full 12-Bit Addresses are Specified. Two Instruction Cycles</td>
<td><code>movff Regs,Regd</code></td>
<td><code>Regd = Regs</code></td>
<td><code>1100 ffff ffff ffff</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>1111 fffdf fffdf ffffd</code></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Code (Hex)</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>movlb Constant</td>
<td>Move low Nybble of BSR to Constant</td>
<td>0000 0001 kkkk kkkk</td>
<td></td>
</tr>
<tr>
<td>movlw Constant</td>
<td>Move Constant to wreg</td>
<td>0000 1110 kkkk kkkk</td>
<td></td>
</tr>
<tr>
<td>movwf Reg, a</td>
<td>Move contents of &quot;wreg&quot; into &quot;Reg&quot; if &quot;a&quot; is set, BSR is used for Reg, else Access Bank is used</td>
<td>0110 111a ffff ffff</td>
<td></td>
</tr>
<tr>
<td>mullw Constant</td>
<td>Multiply Constant by &quot;wreg&quot;</td>
<td>0000 1101 kkkk kkkk</td>
<td></td>
</tr>
<tr>
<td>mullwf Reg</td>
<td>Multiply Register by &quot;wreg&quot; if &quot;a&quot; is set, BSR is used for Reg, else Access Bank is used</td>
<td>0000 0010a ffff ffff</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Operation</td>
<td>Bit pattern</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Negate the</td>
<td>negw Reg, a</td>
<td>Reg = ¬Reg</td>
<td>0110 110a ffff ffff</td>
</tr>
<tr>
<td>Contents of</td>
<td></td>
<td>if (¬Reg &lt; 0x080)</td>
<td></td>
</tr>
<tr>
<td>“Reg” and</td>
<td></td>
<td>OV = 1</td>
<td></td>
</tr>
<tr>
<td>store the</td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>result back</td>
<td></td>
<td>OV = 0</td>
<td></td>
</tr>
<tr>
<td>in “Reg”. If</td>
<td></td>
<td>if (!(¬Reg &amp; 0x080))</td>
<td></td>
</tr>
<tr>
<td>“a” is set, then</td>
<td></td>
<td>N = 1</td>
<td></td>
</tr>
<tr>
<td>Reg is in the BSR Bank, else Reg</td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>is in the Access Bank</td>
<td></td>
<td>N = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (¬Reg &gt; 0x0ff)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (!(¬Reg &amp; 0x0F))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 0x0f)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DC = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (¬Reg == 0x00)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z = 0</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Op Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>Do Nothing</td>
<td><code>nop</code></td>
<td>0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111 1111 1111 1111</td>
<td></td>
</tr>
<tr>
<td>Pop the top</td>
<td><code>pop</code></td>
<td>POP ( ) 0000 0000 0000 0110</td>
<td></td>
</tr>
<tr>
<td>of the Instruction Pointer Stack and Discard the Result.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push the top</td>
<td><code>push</code></td>
<td>PUSH (PC + 2) 0000 0000 0000 0101</td>
<td></td>
</tr>
<tr>
<td>of the Instruction Pointer Stack.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call the 11-Bit 2's Complement &quot;Offset&quot;, Two Instruction Cycles</td>
<td><code>rcall Label</code></td>
<td>PUSH (PC) PC = PC + 2 + Label</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Operation</td>
<td>Bit pattern</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Reset the PICmicro® MCU Processor and all the Registers Affected by _MCLR Reset</td>
<td>Reset</td>
<td>_MCLR = 0</td>
<td>0000 0000 1111 1111</td>
</tr>
<tr>
<td>Reset the PICmicro® MCU Processor and all the Registers Affected by _MCLR Reset</td>
<td>Reset</td>
<td>_MCLR = 1</td>
<td>0000 0000 1111 1111</td>
</tr>
<tr>
<td>Return from Interrupt Handler. If &quot;s&quot; is set, Restore the wreg, STATUS and BSR Registers. Two Instruction Cycles</td>
<td>retfie, s</td>
<td>PC = POP ()</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Interrupt Handler. If &quot;s&quot; is set, Restore the wreg, STATUS and BSR Registers. Two Instruction Cycles</td>
<td>retfie, s</td>
<td>GIE = 0</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Interrupt Handler. If &quot;s&quot; is set, Restore the wreg, STATUS and BSR Registers. Two Instruction Cycles</td>
<td>retfie, s</td>
<td>if (s == 1)</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Interrupt Handler. If &quot;s&quot; is set, Restore the wreg, STATUS and BSR Registers. Two Instruction Cycles</td>
<td>retfie, s</td>
<td>wreg = POP ()</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Interrupt Handler. If &quot;s&quot; is set, Restore the wreg, STATUS and BSR Registers. Two Instruction Cycles</td>
<td>retfie, s</td>
<td>STATUS = POP ()</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Interrupt Handler. If &quot;s&quot; is set, Restore the wreg, STATUS and BSR Registers. Two Instruction Cycles</td>
<td>retfie, s</td>
<td>BSR = POP ()</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Subroutine with new value in wreg. Two Instruction Cycles</td>
<td>retlw Constant</td>
<td>wreg = Constant</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Subroutine with new value in wreg. Two Instruction Cycles</td>
<td>retlw Constant</td>
<td>PC = POP ()</td>
<td>0000 0000 0000000 s</td>
</tr>
<tr>
<td>Return from Subroutine with new value in wreg. Two Instruction Cycles</td>
<td>retlw Constant</td>
<td>kkkk</td>
<td>0000 0000 0000000 s</td>
</tr>
</tbody>
</table>
### Return from Subroutine
- If "s" is set, restore the Wreg, STATUS and BSR Registers. Two Instruction Cycles

### Rotate Left Through the Carry Flag
- If "a" is set then Reg is in BSR Bank else Reg is in the Access Bank

<table>
<thead>
<tr>
<th>Return from Subroutine</th>
<th>Return, s</th>
<th>PC = POP ( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>If &quot;s&quot; is set,</td>
<td></td>
<td>0000 0000 0001 001s</td>
</tr>
<tr>
<td>Restore the Wreg, STATUS and BSR Registers. Two Instruction Cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate Left Through the Carry Flag.</td>
<td>if (d == 0)</td>
<td>0011 01da ffff ffff</td>
</tr>
<tr>
<td>If &quot;a&quot; is set then Reg is in BSR Bank else Reg is in the Access Bank</td>
<td>if (Reg (6) != 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (Reg (6) != 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N = 0</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Operation</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Rotate Left       | rlnf Reg, d, a | if (d == 0)  
|                   |            |     wreg(7:1) = Reg(6:0)  
|                   |            |     wreg(0) = Reg(7)  
|                   |            |     else Reg(7:1) = Reg(6:0)  
|                   |            |     Reg(0) = Reg(7)  
|                   |            |     if (Reg(6) != 0)  
|                   |            |     N = 1  
|                   |            |     else N = 0  
|                   |            |                       | 0100 01da ffff ffff |
| Rotate Right      | rrcf Reg, d, a | if (d == 0)  
|                   |            |     wreg(6:0) = Reg(7:1)  
|                   |            |     wreg(7) = C  
|                   |            |     else Reg(6:0) = Reg(7:1)  
|                   |            |     Reg(7) = C  
|                   |            |     C = Reg(0)  
|                   |            |     if (Reg(0) != 0)  
|                   |            |     N = 1  
|                   |            |     else N = 0  
|                   |            |                       | 0011 00da ffff ffff |
| Rotate Right. | rrcnf Reg, d, a | \[
\begin{align*}
\text{if (d == 0)} \\
\text{wreg (6:0) = Reg (7:1)} \\
\text{wreg (7) = Reg (0)} \\
\text{else} \\
\text{Reg (6:0) = Reg (7:1)} \\
\text{Reg (7) = Reg (0)} \\
\text{if (Reg (0) != 0)} \\
N = 1 \\
\text{else} \\
N = 0
\end{align*}
\]

| Set the Specified Register and Optionally "wreg". | setf Reg, s, a | \[
\begin{align*}
\text{Reg = 0xOFF} \\
\text{if (s == 0)} \\
\text{wreg = 0xOFF}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the PICmicro® MCU in a “Power Down” State</td>
<td>sleep</td>
<td>WDT = 0</td>
<td>0000 0000 0000 0011</td>
</tr>
</tbody>
</table>
| Subtract the Contents of “Reg” and C from wreg and store the Result according to “d” Result in “wreg”, if “a” is set then Reg is in the BSR Bank else it is in the Access Bank | subwfb Reg,d,a | if (“d” == 1)   
| | | wreg = wreg − Reg − IC    
| | | else    
| | | Reg = wreg − Reg − IC    
| | | if ((Reg > 0) & (wreg > 0)) & ((wreg = Reg − IC) < 0x080))    
| | | OV = 1    
| | | else    
| | | OV = 0    
| | | if (((wreg − Reg − C) & 0x080) != 0)    
| | | N = 1    
| | | else    
| | | N = 0 | 0101 01da ffff ffff |
if ((wreg - Reg - |C|) > 0x0FF)
  C = 1
else
  C = 0

if (((wreg & 0x0F) - (Reg & 0x0F) - |C|) > 0xFF)
  DC = 1
else
  DC = 0

if (((wreg - Reg - |C| & 0x0FF) == 0x000)
  Z = 1
else
  Z = 0

Subtract "wreg" from a Constant and store the Result in "wreg"

sublw Constant
wreg = Constant - wreg

if (((wreg < 0) & (Constant < 0)) & ((Constant - wreg) < 0x0080))
  OV = 1
else
  OV = 0

if (((Constant -
Instruction Format Operation Bit pattern

wreg) & 0x080) /H11005

if ((Constant
wreg) & 0x0FF) /H11002

if (((Constant & 0x0F)
wreg & 0x0F)) 0x0F) /H11002

if (((Constant
wreg) & 0x0FF) /H11002

else

else

else

else

DC = 0 = 0

DC = 1 = 1

Z = 0 = 0

Z = 1 = 1
Subtract "wreg" from the Contents of "Reg" and store the Result.

According to "d" Result in "wreg". If "a" is set then "Reg" is in the BSR Bank else Access Bank

\[
\text{subwf Reg,d,a}
\]

\[
\text{if ("d" == 1)}
\]

\[
\text{wreg = Reg} - \text{wreg}
\]

\[
\text{else}
\]

\[
\text{Reg = Reg} - \text{wreg}
\]

\[
\text{if (}(\text{wreg} < 0) \& (\text{Reg} < 0)) \& (\text{Reg} - \text{wreg}) < 0x080)\]

\[
\text{OV} = 1
\]

\[
\text{else}
\]

\[
\text{OV} = 0
\]

\[
\text{if (}(\text{Reg} - \text{wreg}) & 0x080)
\]

\[
\text{N} = 1
\]

\[
\text{else}
\]

\[
\text{N} = 0
\]

\[
\text{if (}(\text{Reg} - \text{wreg}) > 0x0FF)
\]

\[
\text{C} = 1
\]

\[
\text{else}
\]

\[
\text{C} = 0
\]

\[
\text{if (}(\text{Reg} & 0x0F) - (\text{wreg} & 0x0F)) > 0x0F)
\]

\[
\text{DC} = 1
\]
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtract &quot;wreg&quot; from the Contents of &quot;Reg&quot; and &quot;C&quot;, store the Result According to &quot;d&quot; Result in &quot;wreg&quot;. If &quot;a&quot; is set, then Reg is in the BSR Bank else Reg is in the Access Bank</td>
<td>subwfb Reg, d, a</td>
<td>if (&quot;d&quot; == 1) wreg = Reg − wreg − IC else Reg = Reg − wreg − IC if ((Reg &gt; 0) &amp; (wreg &lt; 0)) &amp; (Reg − wreg − IC &lt; 0x080)) OV = 1 else OV = 0 if ((Reg − wreg − C) &amp; 0x080) N = 1</td>
<td>0101 10da ffff ffff</td>
</tr>
</tbody>
</table>

else DC = 0 if ((Reg − wreg) & 0x000) == 0x000) Z = 1 else Z = 0

if (((Reg wreg) & 0x0FF) = 0x000) Z = 1 else Z = 0
else

N = 0
if ((Reg
wreg - IC) > 0x0FF)
C = 1
else
C = 0
if (((Reg & 0x0F) = (wreg & 0x0F) - IC) > 0x0F)
DC = 1
else
DC = 0
if (((Reg
wreg - IC) & 0x0FF) = 0x000)
Z = 1
else
Z = 0
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swap</td>
<td><code>swapf Reg,d,a</code></td>
<td>if (&quot;d&quot; == 1)</td>
<td><code>0011 10da ffff ffff</code></td>
</tr>
<tr>
<td>the Contents</td>
<td>of &quot;Reg&quot; and</td>
<td>store the</td>
<td>Result</td>
</tr>
<tr>
<td>Result</td>
<td>According to</td>
<td>&quot;wreg&quot;. If</td>
<td>&quot;d&quot; Result in</td>
</tr>
<tr>
<td>&quot;d&quot; Result in</td>
<td>&quot;a&quot; is set,</td>
<td>then Reg is,</td>
<td>wreg =</td>
</tr>
<tr>
<td>&quot;wreg&quot;. If</td>
<td>in the BSR</td>
<td></td>
<td><code>(Reg &amp; 0x00F)</code></td>
</tr>
<tr>
<td>&quot;a&quot; is set,</td>
<td>Bank else Reg</td>
<td></td>
<td><code>&lt;&lt; 4)</code></td>
</tr>
<tr>
<td>in the BSR</td>
<td>is in the</td>
<td></td>
<td><code>+</code></td>
</tr>
<tr>
<td>Bank else Reg</td>
<td>Access Bank</td>
<td></td>
<td><code>(Reg &amp; 0x0F0)</code></td>
</tr>
</tbody>
</table>

Read the `tabled Option` switch(`Option`) `0000 0000 0000 10nn`

Program

Contents at the Table

Pointer and Execute as "Option"

Two Instruction Cycles

```
case*:
    TABLAT = 00 *
    ProgMem (TBLPTR) =
    TBLPTR =
    TBLPTR + 1

    case+:
    TABLAT = 10 *
    ProgMem (TBLPTR) =
    TBLPTR =
    TBLPTR + 1

    case-:
    TABLAT = 11 *
    ProgMem (TBLPTR) =
    TBLPTR =
    TBLPTR + 1
```
Write the tablw Option

Contents of the Table
Latch into Program
Memory based on the "Option"
Specification.

Pointer.
If the Destination is Internal
EPROM, the Instruction does not
End until an Interrupt.
Two Instruction Cycles or Many if EPROM Write

<table>
<thead>
<tr>
<th>switch(Option)</th>
<th>0000 0000 0000 11nn</th>
</tr>
</thead>
<tbody>
<tr>
<td>case *</td>
<td>nn Option</td>
</tr>
<tr>
<td>ProgMem (TBLPTR)</td>
<td>00 *</td>
</tr>
<tr>
<td>case +</td>
<td>10 *</td>
</tr>
<tr>
<td>ProgMem(TBLPTR)</td>
<td>11 +</td>
</tr>
</tbody>
</table>

TBLPTR = TBLPTR – 1
TBLPTR = TBLPTR + 1
TABLAT = ProgMem (TBLPTR)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare the</td>
<td>tstfz Reg, a</td>
<td>if (Reg == 0)</td>
<td>0110 011a ffff ffff</td>
</tr>
<tr>
<td>Specified</td>
<td></td>
<td>PC = NextIns</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td></td>
<td>and skip</td>
<td></td>
</tr>
<tr>
<td>zero</td>
<td></td>
<td>if the Register == 0.</td>
<td></td>
</tr>
<tr>
<td>and skip</td>
<td></td>
<td>If &quot;a&quot; is set then the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSR is used for Reg, else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the Access Bank is used.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>One Instruction Cycle if</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Skip Not Executed, Two</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if Skip Executed</td>
<td></td>
</tr>
</tbody>
</table>
| XOR a Constant to the "wreg" and store the Result in "wreg" | xorlw Constant | wreg = wreg ^ Constant 
if ((wreg ^ Constant) == 0x000) 
Z = 1 
else 
Z = 0 
if ((wreg ^ Constant) & 0x080) != 0) 
N = 1 
else 
N = 0 |
|---|---|---|
| XOR "wreg" to the Contents of "Reg" and store the Result According to "d" Result in "wreg" If "a" is Set then Reg is in the BSR Bank else Reg is in the Access Bank | xorwf Reg,d,a | if ("d" == 1) 
wreg = wreg ^ Reg 
else 
Reg = wreg ^ Reg 
if ((wreg ^ Reg) == 0x000) 
Z = 1 
else 
Z = 0 
if ((wreg ^ Reg) & 0x080) != 0) 
N = 1 
else 
N = 0 |
Microchip Special Instruction Mnemonics

The following “special instructions” are macros built into MPASM by Microchip to help make some low-end and mid-range PICmicro® MCU instructions more intuitive. These instructions are built into MPASM and their labels should never be used for macros, addresses (code or variable), or defines.

Most of these special instructions are made up of one or more standard low-end or mid-range PICmicro® MCU instructions. Note that some of these special instructions may change the value of the zero flag.

“LCALL” should never be used because the PCLATH bits are not returned to the appropriate value for the code following “LCALL”. When a “goto” or “call” is executed after an “LCALL” statement and the PCLATH bits are not set appropriately for the current page, execution will jump into the “LCALL” page.

For the low-end PICmicro® MCUs, “LCALL” should be

```
bcf/bcf STATUS, PA0
bcf/bcf STATUS, PA1
bcf/bcf STATUS, PA2
call Label
bsf/bcf STATUS, PA0
bsf/bcf STATUS, PA1
bsf/bcf STATUS, PA2
```

and for the mid-range, “LCALL” should be

```
bcf/bcf PCLATH, 3
bcf/bcf PCLATH, 4
call Label
bsf/bcf PCLATH, 3
bsf/bcf PCLATH, 4
```

“negf” should never be used unless the destination is back into the file register source. If the destination is
"w", note that the contents of the file register source will be changed with the complement of the value. Because of this added complexity, use of this special instruction is not recommended.
<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Actual Instructions</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Carry to File Register</td>
<td><code>addcf Reg, d</code></td>
<td><code>btfsc STATUS, C</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>incf Reg, d</code></td>
<td>if (C == 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (d == 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reg = Reg + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>w = Reg + 1</td>
</tr>
<tr>
<td>Add Digit Carry to File Register</td>
<td><code>addcf Reg, d</code></td>
<td><code>btfsc STATUS, DC</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>incf Reg, d</code></td>
<td>if (DC == 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (d == 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reg = Reg + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>w = Reg + 1</td>
</tr>
<tr>
<td>Branch to Label</td>
<td>B Label</td>
<td><code>goto Label</code></td>
<td>PC = (PCLATH &lt;&lt; 8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp; 0x01800 + Label;</td>
</tr>
<tr>
<td>Branch on Carry Set</td>
<td>BC Label</td>
<td><code>goto Label</code></td>
<td>PC = (PCLATH &lt;&lt; 8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp; 0x01800 + Label;</td>
</tr>
<tr>
<td>Branch on Digit Carry Set</td>
<td>BDC Label</td>
<td><code>goto Label</code></td>
<td>PC = (PCLATH &lt;&lt; 8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp; 0x01800 + Label;</td>
</tr>
<tr>
<td>Branch on Carry Reset</td>
<td>BNC Label</td>
<td><code>goto Label</code></td>
<td>PC = (PCLATH &lt;&lt; 8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp; 0x01800 + Label;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Branch on Digit Carry</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reset</strong></td>
<td><strong>BNDC Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>btfss STATUS, DC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>goto Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[DC = 0]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PC = ((PCLATH &lt;&lt; 8) &amp; 0x01800) = Label;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Branch on Zero Reset</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>BNZ Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>btfss STATUS, Z</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>goto Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[Z = 0]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PC = ((PCLATH &lt;&lt; 8) &amp; 0x01800) = Label;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Branch on Zero Set</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>BE Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>btfsc STATUS, Z</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>goto Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[Z = 1]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PC = ((PCLATH &lt;&lt; 8) &amp; 0x01800) = Label;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Clear Carry</strong></td>
<td><strong>clrc</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf STATUS, C</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>C = 0;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Clear Digit Carry</strong></td>
<td><strong>clrdc</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf STATUS, DC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>DC = 0;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Clear Zero</strong></td>
<td><strong>clrz</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf STATUS, Z</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Z = 0;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Long Call — Do NOT use as Described Above</strong></td>
<td><strong>lcall Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Low-End:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf/bsf STATUS, PA0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf/bsf STATUS, PA1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf/bsf STATUS, PA2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>call Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Mid-Range:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf/bsf PCLATH, 3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>bcf/bsf PCLATH, 3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>call Label</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Instruction</td>
<td>Actual Instructions</td>
<td>Operation</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
<td>---------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Long Goto</td>
<td>goto Label</td>
<td>Low-End: bcf/bsf STATUS, PA0, bcf/bsf STATUS, PA1, bcf/bsf STATUS, PA2, goto Label</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mid-Range: bcf/bsf PCLATH, 3</td>
<td></td>
</tr>
<tr>
<td>Load &quot;w&quot; with Contents of &quot;Reg&quot;</td>
<td>movfw Reg</td>
<td>movf Reg, w</td>
<td>w = Reg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (Reg = 0) Z = 1; else Z = 0;</td>
</tr>
<tr>
<td>Negate a File Register. — Do NOT use as Described Above</td>
<td>negf Reg, d</td>
<td>comf Reg, f</td>
<td>Reg = Reg ^ 0xFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>incf Reg, d</td>
<td>if (d = 0) w = Reg + 1; else Reg = Reg + 1;</td>
</tr>
<tr>
<td>Set Carry</td>
<td>setc</td>
<td>bcf STATUS, C</td>
<td>C = 0;</td>
</tr>
<tr>
<td>Set Digit Carry</td>
<td>setdc</td>
<td>bcf STATUS, DC</td>
<td>DC = 0;</td>
</tr>
<tr>
<td>Set Zero</td>
<td>setx</td>
<td>bcf STATUS, Z</td>
<td>Z = 0;</td>
</tr>
</tbody>
</table>
Skip the next Instruction if the Carry Flag is Set
skpc     btfss STATUS, C if (C = 1)
         PC = PC + 1;
Skip the next Instruction if the Digit Carry Flag is Set
skpdc    btfss STATUS, DC if (DC = 1)
         PC = PC + 1;
Skip the next Instruction if the Carry Flag is Reset
skpnc    btfsc STATUS, C if (C = 0)
         PC = PC + 1;
Skip the next Instruction if the Digit Carry Flag is Reset
skpndc   btfsc STATUS, DC if (DC = 0)
         PC = PC + 1;
Skip the next Instruction if the Zero Flag is Reset
skpnz    btfsc STATUS, Z if (Z = 0)
         PC = PC + 1;
Skip the next Instruction if the Zero Flag is Set
skpz     btfss STATUS, Z if (Z = 1)
         PC = PC + 1;
Subtract Carry from File Register
   subcf Reg, d btfsc STATUS, C if (C = 1)
   decf Reg, d if (d = 1)
       Reg = Reg - 1;
       w = Reg - 1
<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction</th>
<th>Actual Instructions</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract Digit Carry</td>
<td><code>addc</code> Reg, d</td>
<td><code>btfsc</code> STATUS, DC</td>
<td>If (DC == 1)</td>
</tr>
<tr>
<td>To File Register</td>
<td><code>inc</code> Reg, d</td>
<td></td>
<td>if (d == 1) Reg = Reg - 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>else w = Reg - 1;</td>
</tr>
<tr>
<td>Load &quot;Z&quot; with 1 if Contents of &quot;Reg&quot;</td>
<td><code>movfw</code> Reg</td>
<td><code>movf</code> Reg, w</td>
<td>If (Reg == 0)</td>
</tr>
<tr>
<td>equal 0</td>
<td></td>
<td></td>
<td>Z = 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>else Z = 0;</td>
</tr>
</tbody>
</table>
Parallax PICmicro® MCU Instruction Set

Parallax Inc. (manufacturers of the “Basic Stamp”) have written a very popular assembler for the Microchip PICmicro® MCUs. "PASM" (as it is known) implements an assembler language that is similar to the Intel 8051 instruction set. The assembler also supports MPASM (standard Microchip) instruction formats as well.

Some of these instructions are designed specifically for the low-end PICmicro® MCUs (they have been noted). If you're working with a mid-range PICmicro® MCU, these instructions MUST NOT be used.

Note that many of these mnemonics result in multiple PICmicro® MCU instructions with unexpected changes to the STATUS and "w" register.

PASM is available from the Parallax web site.

# – Literal Instructions
fr – File Register
<table>
<thead>
<tr>
<th>PASM Data Instructions</th>
<th>Cycles</th>
<th>Context</th>
<th>Actual PICmicro® MCU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLR Parm</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>W</em></td>
<td>1</td>
<td>Zero</td>
<td>clrw</td>
</tr>
<tr>
<td>fr</td>
<td>1</td>
<td>Zero</td>
<td>clr fr</td>
</tr>
<tr>
<td><strong>WDT</strong></td>
<td>1</td>
<td>_TO, _PD</td>
<td>clrwdt</td>
</tr>
<tr>
<td><strong>MOV Parm</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>W</em>, #</td>
<td>1</td>
<td>None</td>
<td>movlw #</td>
</tr>
<tr>
<td><em>W</em>, fr</td>
<td>1</td>
<td>Zero</td>
<td>movf fr, w</td>
</tr>
<tr>
<td><em>W</em>, /fr</td>
<td>1</td>
<td>Zero</td>
<td>comf fr, w</td>
</tr>
<tr>
<td><em>W</em>, fr-W</td>
<td>1</td>
<td>Z, C, DC</td>
<td>subwf fr, w</td>
</tr>
<tr>
<td><em>W</em>, &lt;&lt;fr</td>
<td>1</td>
<td>Z</td>
<td>incf fr, w</td>
</tr>
<tr>
<td><em>W</em>, &gt;&gt;fr</td>
<td>1</td>
<td>Z</td>
<td>decf fr, w</td>
</tr>
<tr>
<td><em>W</em>, tris</td>
<td>1</td>
<td>Carry</td>
<td>rrf fr, w</td>
</tr>
<tr>
<td><em>W</em>, tris</td>
<td>1</td>
<td>Carry</td>
<td>rlf fr, w</td>
</tr>
<tr>
<td><em>fr</em>, W</td>
<td>1</td>
<td>None</td>
<td>movwf fr</td>
</tr>
<tr>
<td>tris, W</td>
<td>2</td>
<td>w</td>
<td>movl w</td>
</tr>
<tr>
<td>tris, #</td>
<td>2</td>
<td>w, Zero</td>
<td>movf fr, w</td>
</tr>
<tr>
<td>option, W</td>
<td>1</td>
<td>None</td>
<td>option</td>
</tr>
<tr>
<td>option, #</td>
<td>2</td>
<td>w</td>
<td>movlw #</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>R1, R2</td>
<td>R0, RC, RT</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>--------</td>
<td>------------</td>
</tr>
<tr>
<td>OPTION, fr</td>
<td>OPTION = fr</td>
<td>2 w, Zero</td>
<td>movf fr, w</td>
</tr>
<tr>
<td>fr, #</td>
<td>fr = #</td>
<td>2 None</td>
<td>movlw #</td>
</tr>
<tr>
<td>fr, fr2</td>
<td>fr = fr2</td>
<td>2 Zero</td>
<td>movf fr2, w</td>
</tr>
</tbody>
</table>

**ADD Parm**

ADD two Values

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>R1, R2</th>
<th>R0, RC, RT</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>W, fr</td>
<td>W = W + fr</td>
<td>1 Z, C, DC</td>
<td>addwf fr, w</td>
<td></td>
</tr>
<tr>
<td>fr, W</td>
<td>fr = W + fr</td>
<td>1 Z, C, DC</td>
<td>addwf fr, f</td>
<td></td>
</tr>
<tr>
<td>fr, #</td>
<td>fr = fr + #</td>
<td>2 W, Z, C, DC</td>
<td>addwf fr, f</td>
<td></td>
</tr>
<tr>
<td>fr, fr2</td>
<td>fr = fr + fr2</td>
<td>2 W, Z, C, DC</td>
<td>addwf fr2, w</td>
<td></td>
</tr>
</tbody>
</table>

**SUB Parm**

Subtraction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>R1, R2</th>
<th>R0, RC, RT</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr, W</td>
<td>fr = fr+(W^0x0FF)+1</td>
<td>1 Z, C, DC</td>
<td>subwf fr, f</td>
<td></td>
</tr>
<tr>
<td>fr, #</td>
<td>fr = fr+(#^0x0FF)+1</td>
<td>2 W, Z, C, DC</td>
<td>subwf fr, f</td>
<td></td>
</tr>
<tr>
<td>fr, fr2</td>
<td>fr=fr+(fr2^0x0FF)+1</td>
<td>2 W, Z, C, DC</td>
<td>subwf fr2, w</td>
<td></td>
</tr>
</tbody>
</table>

**AND Parm**

Bitwise AND

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>R1, R2</th>
<th>R0, RC, RT</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>W, #</td>
<td>W = W &amp; #</td>
<td>1 Zero</td>
<td>andlw #</td>
<td></td>
</tr>
<tr>
<td>fr, W</td>
<td>fr = W &amp; fr</td>
<td>1 Zero</td>
<td>andwf fr, w</td>
<td></td>
</tr>
<tr>
<td>fr, #</td>
<td>fr = fr &amp; #</td>
<td>2 W, Zero</td>
<td>andlw #</td>
<td></td>
</tr>
<tr>
<td>fr, fr2</td>
<td>fr = fr &amp; fr2</td>
<td>2 W, Zero</td>
<td>andwf fr2, w</td>
<td></td>
</tr>
</tbody>
</table>
### PASM Data Instructions

<table>
<thead>
<tr>
<th>Instruction Description</th>
<th>Cycles</th>
<th>Context</th>
<th>Resources</th>
<th>Actual PICmicro® MCU</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR Parm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;w, #&quot;</td>
<td>1</td>
<td>Zero</td>
<td>w</td>
<td>iorlw #</td>
</tr>
<tr>
<td>&quot;w, fr&quot;</td>
<td>1</td>
<td>Zero</td>
<td>fr</td>
<td>iorwf fr, w</td>
</tr>
<tr>
<td>&quot;fr, #&quot;</td>
<td>1</td>
<td>Zero</td>
<td>fr, #</td>
<td>iorwf fr, f</td>
</tr>
<tr>
<td>&quot;fr, fr2&quot;</td>
<td>2</td>
<td>w, Zero</td>
<td>fr2</td>
<td>movlw #</td>
</tr>
<tr>
<td>&quot;fr, fr2&quot;</td>
<td>2</td>
<td>w, Zero</td>
<td>fr, fr2</td>
<td>movf fr2, w</td>
</tr>
<tr>
<td>XOR Parm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;w, #&quot;</td>
<td>1</td>
<td>Zero</td>
<td>w</td>
<td>xorlw #</td>
</tr>
<tr>
<td>&quot;w, fr&quot;</td>
<td>1</td>
<td>Zero</td>
<td>w, fr</td>
<td>xorwf fr, w</td>
</tr>
<tr>
<td>&quot;fr, #&quot;</td>
<td>1</td>
<td>Zero</td>
<td>fr, #</td>
<td>xorwf fr, f</td>
</tr>
<tr>
<td>&quot;fr, fr2&quot;</td>
<td>2</td>
<td>w, Zero</td>
<td>fr, fr2</td>
<td>movf fr2, w</td>
</tr>
<tr>
<td>&quot;fr, fr2&quot;</td>
<td>2</td>
<td>w, Zero</td>
<td>fr, fr2</td>
<td>xorwf fr, f</td>
</tr>
<tr>
<td>DEC Parm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;fr&quot;</td>
<td>1</td>
<td>Zero</td>
<td>fr</td>
<td>decf fr, f</td>
</tr>
<tr>
<td>INC Parm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;fr&quot;</td>
<td>1</td>
<td>Zero</td>
<td>fr</td>
<td>incf fr, f</td>
</tr>
<tr>
<td>NEG Parm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;fr&quot;</td>
<td>2</td>
<td>Zero</td>
<td>0 - fr</td>
<td>comf fr, f</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>incf fr, f</td>
</tr>
</tbody>
</table>
### NOT Parm
**Bitwise Complement**

- **W**
  
  \[ w = w \oplus 0x0FF \]
  
  1 \hspace{1em} \text{Zero} \hspace{1em} \text{xorlw} \hspace{1em} 0x0FF

- **fr**
  
  \[ fr = fr \oplus 0x0FF \]
  
  1 \hspace{1em} \text{Zero} \hspace{1em} \text{comf} \hspace{1em} fr

### TEST Parm
**Test Parm Equal to Zero**

- **W**
  
  \[ Z = ( w = 0 ) \]
  
  1 \hspace{1em} \text{Zero} \hspace{1em} \text{iorlw} \hspace{1em} 0

- **fr**
  
  \[ Z = ( fr = 0 ) \]
  
  1 \hspace{1em} \text{Zero} \hspace{1em} \text{movf} \hspace{1em} fr, f

### RR Parm
**Rotate Register to Right**

- **fr**
  
  \[ fr = fr >> 1 \]
  
  1 \hspace{1em} \text{Carry} \hspace{1em} \text{rrf} \hspace{1em} fr, f

### RL Parm
**Rotate Register to Left**

- **fr**
  
  \[ fr = fr << 1 \]
  
  1 \hspace{1em} \text{Carry} \hspace{1em} \text{rlf} \hspace{1em} fr, f

### SWAP Parm
**Swap Nybbles of Register**

- **fr**
  
  \[ fr = \text{<>}fr \]
  
  1 \hspace{1em} \text{None} \hspace{1em} \text{swapf} \hspace{1em} fr, f

### PASM Bit Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cycles</th>
<th>Context</th>
<th>Actual PICmicro® MCU Resources Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRB fr, bit fr.bit = 0</td>
<td>( fr.\text{bit} = 0 )</td>
<td>1</td>
<td>None</td>
<td>bcf fr, bit</td>
</tr>
<tr>
<td>SETB fr, bit fr.bit = 1</td>
<td>( fr.\text{bit} = 1 )</td>
<td>1</td>
<td>None</td>
<td>baf fr, bit</td>
</tr>
<tr>
<td>CLC</td>
<td>( \text{Carry} = 0 )</td>
<td>1</td>
<td>None</td>
<td>bcf STATUS, C</td>
</tr>
<tr>
<td>STC</td>
<td>( \text{Carry} = 1 )</td>
<td>1</td>
<td>None</td>
<td>baf STATUS, C</td>
</tr>
<tr>
<td>CLS</td>
<td>( \text{Zero} = 0 )</td>
<td>1</td>
<td>None</td>
<td>bcf STATUS, Z</td>
</tr>
<tr>
<td>STZ</td>
<td>( \text{Zero} = 0 )</td>
<td>1</td>
<td>None</td>
<td>baf STATUS, Z</td>
</tr>
<tr>
<td>ADDB fr, bit fr = fr + Bit</td>
<td>( fr = fr + \text{Bit} )</td>
<td>2</td>
<td>Zero</td>
<td>btfsc fr, bit</td>
</tr>
<tr>
<td>SUBB fr, bit fr = fr - bit</td>
<td>( fr = fr - \text{Bit} )</td>
<td>2</td>
<td>Zero</td>
<td>btfsc fr, bit</td>
</tr>
</tbody>
</table>
### PASM Bit Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cycles</th>
<th>Context</th>
<th>Actual PICmicro® MCU Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVB fr.b, fr2.b2</td>
<td>Move Bit</td>
<td>4</td>
<td>None</td>
<td>btfss fr2, b2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcf fr, b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfsc fr2, b2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bsf fr, b</td>
</tr>
<tr>
<td>MOVB fr.b, /fr2.b2</td>
<td>Move Invert</td>
<td>4</td>
<td>None</td>
<td>btfsc fr2, b2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bcf fr, b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfss fr2, b2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bsf fr, b</td>
</tr>
</tbody>
</table>

### PASM PICmicro® MCU Microcontroller Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cycles</th>
<th>Context</th>
<th>Actual PICmicro® MCU Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>Do Nothing</td>
<td>1</td>
<td>Nothing</td>
<td>nop</td>
</tr>
<tr>
<td>SLEEP</td>
<td>Put PICmicro® MCU to Sleep</td>
<td>N/A</td>
<td>_TO, _PD</td>
<td>sleep</td>
</tr>
<tr>
<td>LSET Addr</td>
<td>Jump Setup</td>
<td>0-2</td>
<td>PA0, PA1</td>
<td>bcf/bsf STATUS, PA0</td>
</tr>
</tbody>
</table>

* - Low End Instruction, bcf/bsf of STATUS PAx Bits Address Dependant
<table>
<thead>
<tr>
<th>Instruction Description</th>
<th>Cycles</th>
<th>Context</th>
<th>Resources Affected</th>
<th>Actual PICmicro® MCU Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZ Parm Skip if Result = 0</td>
<td>1/2</td>
<td>w</td>
<td>incfsz fr, w</td>
<td></td>
</tr>
<tr>
<td>&quot;W, ++fr&quot; w = fr + 1</td>
<td>1/2</td>
<td>w</td>
<td>decfsz fr, w</td>
<td></td>
</tr>
<tr>
<td>&quot;W, --fr&quot; w = fr - 1</td>
<td>1/2</td>
<td>w</td>
<td>incfsz fr, f</td>
<td></td>
</tr>
<tr>
<td>INCSZ fr w = fr + 1, if Z Skip</td>
<td>1/2</td>
<td>w</td>
<td>decfsz fr, f</td>
<td></td>
</tr>
<tr>
<td>DECSZ fr w = fr - 1, if Z Skip</td>
<td>1/2</td>
<td>w</td>
<td>incfsz fr, f</td>
<td></td>
</tr>
<tr>
<td>SB fr, bit Skip if Bit Set</td>
<td>1/2</td>
<td>None</td>
<td>btfss fr, bit</td>
<td></td>
</tr>
<tr>
<td>SBB fr, bit Skip if Bit Reset</td>
<td>1/2</td>
<td>None</td>
<td>btfsc fr, bit</td>
<td></td>
</tr>
<tr>
<td>SC Skip if Carry Set</td>
<td>1/2</td>
<td>None</td>
<td>btfss STATUS, C</td>
<td></td>
</tr>
<tr>
<td>SNC Skip if C Reset</td>
<td>1/2</td>
<td>None</td>
<td>btfsc STATUS, C</td>
<td></td>
</tr>
<tr>
<td>EZ Skip if Zero Set</td>
<td>1/2</td>
<td>None</td>
<td>btfsc STATUS, Z</td>
<td></td>
</tr>
<tr>
<td>SNZ Skip if Zero Reset</td>
<td>1/2</td>
<td>None</td>
<td>btfsc STATUS, Z</td>
<td></td>
</tr>
<tr>
<td>CJA fr, # if fr &gt; #</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movlw #</td>
<td></td>
</tr>
<tr>
<td>Skip_Next</td>
<td></td>
<td></td>
<td>addwf fr, w</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>btfss STATUS, C</td>
<td></td>
</tr>
<tr>
<td>CJA fr, fr2 if fr &gt; fr2</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movf fr, w</td>
<td></td>
</tr>
<tr>
<td>Skip_Next</td>
<td></td>
<td></td>
<td>subwf fr2, w</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>btfss STATUS, C</td>
<td></td>
</tr>
<tr>
<td>CJA fr, # if fr &gt; #</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movlw #</td>
<td></td>
</tr>
<tr>
<td>Skip_Next</td>
<td></td>
<td></td>
<td>subwf fr, w</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>btfss STATUS, C</td>
<td></td>
</tr>
<tr>
<td>CJA fr, fr2 if fr &gt; fr2</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movf fr2, w</td>
<td></td>
</tr>
<tr>
<td>Skip_Next</td>
<td></td>
<td></td>
<td>subwf fr, w</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>btfss STATUS, C</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Cycles</td>
<td>Context</td>
<td>Actual PICmicro® MCU Instructions</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>--------</td>
<td>---------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>CSB fr, # if fr &lt; #</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movlw #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>subwf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfsc STATUS, C</td>
</tr>
<tr>
<td>CSB fr, fr2 if fr &lt; fr2</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movf fr2, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>subwf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfsc STATUS, C</td>
</tr>
<tr>
<td>CSBE fr, # if fr &lt;= #</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movlw #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addwf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfsc STATUS, C</td>
</tr>
<tr>
<td>CSBE fr, fr2 if fr &lt;= fr2</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>subwf fr2, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfsc STATUS, C</td>
</tr>
<tr>
<td>CSE fr, # if fr == #</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movlw #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>subwf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfss STATUS, Z</td>
</tr>
<tr>
<td>CSE fr, fr2 if fr == fr2</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movf fr2, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>subwf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfss STATUS, Z</td>
</tr>
<tr>
<td>CSNE fr, # if fr == #</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movlw #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>subwf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfsc STATUS, Z</td>
</tr>
<tr>
<td>CSNE fr, fr2 if fr == fr2</td>
<td>Skip_Next</td>
<td>3/4</td>
<td>w, C, DC, Z</td>
<td>movf fr2, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>subwf fr, w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>btfsc STATUS, Z</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Cycles</td>
<td>Context</td>
<td>Resources</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------</td>
<td>--------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>JMP Parm</td>
<td>Jump to Address</td>
<td>2</td>
<td>None</td>
<td>PC</td>
</tr>
<tr>
<td>&quot;addr9&quot;</td>
<td>PC = 9 Bit Address</td>
<td>2</td>
<td>None</td>
<td>PC</td>
</tr>
<tr>
<td>* - Low End Instruction</td>
<td>&quot;PC+W&quot;</td>
<td>PC = PC + Offset w</td>
<td>2</td>
<td>Z, C, DC</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>PC = w</td>
<td>2</td>
<td>None</td>
<td>PCL</td>
</tr>
<tr>
<td>CALL addr#</td>
<td>Call Subroutine</td>
<td>2</td>
<td>None</td>
<td>addr#</td>
</tr>
<tr>
<td>* - Low End Instruction</td>
<td>RET</td>
<td>Return &amp; w = 0</td>
<td>2</td>
<td>w</td>
</tr>
<tr>
<td>SKIP</td>
<td>Skip Over Next Ins</td>
<td>2</td>
<td>None</td>
<td>PSR, 7</td>
</tr>
<tr>
<td>* - Low End Instruction</td>
<td>LMP Addr</td>
<td>LSET before JMP</td>
<td>2-5</td>
<td>PA0-PA2</td>
</tr>
<tr>
<td>LJMP Addr</td>
<td>LSET before CALL</td>
<td>2-5</td>
<td>PA0-PA2</td>
<td>bcf/bsf STATUS, PAx</td>
</tr>
<tr>
<td>* - Low End Instruction</td>
<td>LCALL Addr</td>
<td>LSET before CALL</td>
<td>2-5</td>
<td>PA0-PA2</td>
</tr>
<tr>
<td>* - Low End Instruction</td>
<td>RETW 'String' Table Return</td>
<td>2</td>
<td>w</td>
<td>relw</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Cycles</td>
<td>Context</td>
<td>Resources</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------</td>
<td>--------</td>
<td>---------</td>
<td>-----------------</td>
</tr>
<tr>
<td>IJNZ fr, addr9</td>
<td>Increment/Jump</td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>DJNZ fr, addr9</td>
<td>Decrement/Jump</td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>JB fr, bit, addr9</td>
<td>Jump on Bit</td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>JNB fr, bit, addr9</td>
<td></td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>JC addr9</td>
<td>Jump on Carry</td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>JNC addr9</td>
<td>Jump on !Carry</td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>JZ addr9</td>
<td>Jump on Zero</td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>JNZ addr9</td>
<td>Jump on !Zero</td>
<td>2/3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>CJA fr, #, addr9</td>
<td></td>
<td>4/5</td>
<td>w, C, DC, Z</td>
<td>movlw # goto addr9</td>
</tr>
<tr>
<td></td>
<td>if fr &gt; #</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>goto addr9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CJA fr, fr2, addr9</td>
<td></td>
<td>4/5</td>
<td>w, C, DC, Z</td>
<td>movlw fr2, w btfss STATUS, C goto addr9</td>
</tr>
<tr>
<td></td>
<td>if fr &gt; fr2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>goto addr9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Machine Code</td>
<td>Flags</td>
<td>Note</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>--------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>CJAE fr, #, addr9</td>
<td>if fr &gt;= #</td>
<td>movlw #</td>
<td>-</td>
<td>goto addr9</td>
</tr>
<tr>
<td>CJAE fr, fr2, addr9</td>
<td>if fr &gt;= fr2</td>
<td>subwf fr, w</td>
<td>btfss STATUS, C</td>
<td>goto addr9</td>
</tr>
<tr>
<td>CJE fr, #, addr9</td>
<td>if fr == #</td>
<td>movlw #</td>
<td>subwf fr, w</td>
<td>goto addr9</td>
</tr>
<tr>
<td>CJE fr, fr2, addr9</td>
<td>if fr == fr2</td>
<td>subwf fr, w</td>
<td>btfss STATUS, C</td>
<td>goto addr9</td>
</tr>
</tbody>
</table>

The given code snippet contains assembly language instructions for conditional jumps based on the register `fr`. Each instruction checks if the value in register `fr` meets a specific condition (greater than or equal to, less than, less than or equal to, or equal to) and then jumps to a specific address if the condition is true. The `movlw` instruction moves a literal value to the `w` register, and the `btfss` instruction tests the status flag after an arithmetic operation. The `goto` instruction then jumps to the specified address if the condition is met.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cycles</th>
<th>Context</th>
<th>Resources</th>
<th>Actual PICmicro® MCU Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CJE fr, fr2, addr9</td>
<td>if fr == fr2 goto addr9</td>
<td>4/5</td>
<td>w, C, DC, Z</td>
<td>movf fr2, w subwf fr, w btfsc STATUS, Z goto addr9</td>
<td></td>
</tr>
<tr>
<td>CJNE fr, fr2, addr9</td>
<td>if fr == fr2 goto addr9</td>
<td>4/5</td>
<td>w, C, DC, Z</td>
<td>movf fr2, w subwf fr, w btfss STATUS, Z goto addr9</td>
<td>btfss STATUS, Z goto addr9</td>
</tr>
<tr>
<td>CJNE fr, #, addr9</td>
<td>if fr == # goto addr9</td>
<td>4/5</td>
<td>w, C, DC, Z</td>
<td>movlw # subwf fr, w btfss STATUS, Z goto addr9</td>
<td>btfss STATUS, Z goto addr9</td>
</tr>
<tr>
<td>CJNE fr, fr2, addr9</td>
<td>if fr == fr2 goto addr9</td>
<td>4/5</td>
<td>w, C, DC, Z</td>
<td>movf fr2, w subwf fr, w btfss STATUS, Z goto addr9</td>
<td>btfss STATUS, Z goto addr9</td>
</tr>
</tbody>
</table>

[#### - End Table]
The PICmicro® MCU's Arithmetic Logic Unit

<table>
<thead>
<tr>
<th>Standard PICmicro® MCU Processor ALU Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>Move</td>
</tr>
<tr>
<td>Addition</td>
</tr>
<tr>
<td>Subtraction</td>
</tr>
<tr>
<td>Negation</td>
</tr>
<tr>
<td>Increment</td>
</tr>
<tr>
<td>Decrement</td>
</tr>
</tbody>
</table>
Along with these functions, the PIC17Cxx and PIC18Cxx also have an 8-bit \( \times \) 8-bit multiplier.

The PICmicro\textsuperscript{®} MCU’s “ALU” ("Arithmetic Logic Unit") could be blocked out as shown in Fig. 5.1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>ALU Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>None</td>
</tr>
<tr>
<td>OR</td>
<td>None</td>
</tr>
<tr>
<td>XOR</td>
<td>None</td>
</tr>
<tr>
<td>Complement</td>
<td>XOR with 0xFF</td>
</tr>
<tr>
<td>Shift Left</td>
<td>None</td>
</tr>
<tr>
<td>Shift Right</td>
<td>None</td>
</tr>
</tbody>
</table>

Figure 5.1 ALU Implementation Using Multiplexed Inputs
**PICmicro® MCU Processor Architectures**  125

**Low-End PICmicro® MCUs**

The “Low-End” PICmicro® MCUs have the part numbers:

- PIC12C5xx
- PIC16C5x
- PIC16C50x

where “x” can be any digit.

A sample low-end PICmicro® MCU processor architecture is shown in Fig. 5.2.

*Figure 5.2  Low-end PICmicro® MCU Architecture*
Reset addresses are at the last location in program memory. The following table lists the different reset vectors for different device program memory sizes.

<table>
<thead>
<tr>
<th>Program Memory Size</th>
<th>Reset Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0x01FF</td>
</tr>
<tr>
<td>1024</td>
<td>0x03FF</td>
</tr>
<tr>
<td>2048</td>
<td>0x07FF</td>
</tr>
</tbody>
</table>

It is recommended that the reset address is ignored in the Low-End PICmicro® MCUs (or used to load the oscillator calibration register ["OSCCAL"] value into "w") before rolling over the Program Counter and starting at address 0x0000, like the other PICmicro® MCU devices.

Register access

The low-end register space is shown in Fig. 5.3.

The first 16 addresses of each bank are common. The 16 bank unique file registers are located in the last 16 addresses of the bank. This limitation of only being able to address data 16 bytes at a time prevents the construction of arrays or other data structures longer than 16 bytes.

Bank 0 can be accessed directly within instructions. Other banks can only be accessed using the FSR (and INDF) index registers. The following table lists bank offsets.
Note that the PICmicro® MCU’s FSR ("index") register can never equal zero. The table below lists which bits will be set in the low-end’s FSR depending on how many bank registers the PICmicro® MCU has.

### Low-end PICmicro® MCU Unique Bank Address Table

<table>
<thead>
<tr>
<th>Bank</th>
<th>FSR</th>
<th>Start of 16 Unique Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000</td>
<td>0x010</td>
</tr>
<tr>
<td>1</td>
<td>0x020</td>
<td>0x030</td>
</tr>
<tr>
<td>2</td>
<td>0x040</td>
<td>0x050</td>
</tr>
<tr>
<td>3</td>
<td>0x060</td>
<td>0x070</td>
</tr>
</tbody>
</table>

### Low-end PICmicro® MCU Minimum FSR Value to Number of Banks

<table>
<thead>
<tr>
<th>Number of Banks</th>
<th>Set FSR bits</th>
<th>Minimum FSR value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7, 6, 5</td>
<td>0x0E0</td>
</tr>
<tr>
<td>2</td>
<td>7, 6</td>
<td>0x0C0</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>0x080</td>
</tr>
</tbody>
</table>
128  Chapter 5

STATUS register

<table>
<thead>
<tr>
<th>Address</th>
<th>Bits</th>
<th>Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x003</td>
<td>7</td>
<td>GFUWF - in PIC12C5xx and PIC16C505: when Set, Reset from Sleep on Pin Change. When Set, power up or MCLR reset. In other Devices the Bit 7 is Unused.</td>
</tr>
<tr>
<td>6-5</td>
<td>PA1-PA0 - Select the Page to execute out of: 00 - Page 0 (0x0000 to 0x01FF) 01 - Page 1 (0x0200 to 0x03FF) 10 - Page 2 (0x0400 to 0x05FF) 11 - Page 3 (0x0600 to 0x07FF)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>_TO - Set after Power Up, clrwdt and sleep instructions</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>_PD - Set after Power Up, clrwdt instruction. Reset after sleep instruction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Z - Set if the eight bit result is equal to zero</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DC - Set for low order Nybble carry after addition or subtraction instruction</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>C - Set for Carry after addition or subtraction instruction</td>
<td></td>
</tr>
</tbody>
</table>

Program counter

The low-end PICmicro® MCU’s program counter block diagram is given in Fig. 5.4.

The “PA0” and “PA1” bits of the STATUS register (bits five and six) perform the same function as the “PCLATH” register of the other PICmicro® MCUs. Bit PA0 is used to provide bit nine of the destination address to jump to during a “goto” or “call” instruction or when “PCL” is
written to. Bit PA1 is address bit ten. In some low-end PICmicro® MCUs, bit seven of the STATUS register is referred to as “PA2”. This bit is not used by any of the current PICmicro® MCUs.

To jump to a new “page” address, the following instruction sequence is used:

```
STATUS = (STATUS & 0x01F) + ((HIGH new_address & 0x0FE) << 4);
PCL = LOW new_address;
```

Note that subroutines and tables at addresses 0x0100 to 0x01FF, 0x0300 to 0x03FF, 0x0500 to 0x05FF, and
00x700 to 0x07FF cannot be accessed directly. Instead, redirection using a “goto” instruction is required. The “call stack” is two elements deep.

**Mid-Range PICmicro® MCUs**

The mid-range PICmicro® MCUs have the part numbers:

- PIC12C6xx
- PIC14000
- PIC16C55x
- PIC16C6x (x)
- PIC16C7x (x)
- PIC16C8x
- PIC16F8x (x)
- PIC16C9xx

The mid-range PICmicro® MCU’s have the block diagram shown in Fig. 5.5. Upon Reset, execution starts at address 0x00000. Interrupts are handled at address 0x00004. The configuration registers are located at address 0x02007.

**Register access**

The Mid-Range PICmicro® MCUs can have up to four register “banks” of 0x080 (128) registers. Each register is accessed using the “RPx” bits of the STATUS register. For the different Register Banks and Register Addresses, the following table is used to set the RPx bits.
The "XOR Value" is the value that is XORed with the Register Address to ensure it is within the bank address range of 0 to 0x07F.

When the FSR (Index) register is used to access data in Banks 1 through 3, the “IRP” bit of the STATUS register will be set appropriately and the least significant 8 bits of the address are loaded into the FSR register.
The Register Address Map looks like the following:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>INDF</td>
<td>INDF</td>
<td></td>
</tr>
<tr>
<td>0x001</td>
<td>TMRO</td>
<td>OPTION</td>
<td></td>
</tr>
<tr>
<td>0x002</td>
<td>PCL</td>
<td>PCL</td>
<td></td>
</tr>
<tr>
<td>0x003</td>
<td>STATUS</td>
<td>STATUS</td>
<td></td>
</tr>
<tr>
<td>0x004</td>
<td>FSR</td>
<td>FSR</td>
<td></td>
</tr>
<tr>
<td>0x005</td>
<td>PORTA</td>
<td>TRISA</td>
<td></td>
</tr>
<tr>
<td>0x006</td>
<td>PORTB</td>
<td>TRISB</td>
<td></td>
</tr>
<tr>
<td>0x007</td>
<td>PORTC</td>
<td>TRISC</td>
<td>Available in 28/40 Pin Parts</td>
</tr>
<tr>
<td>0x008</td>
<td>PORTD</td>
<td>TRISD</td>
<td>Available in 40 Pin Parts</td>
</tr>
<tr>
<td>0x009</td>
<td>PORTE</td>
<td>TRISE</td>
<td>Available in 40 Pin Parts</td>
</tr>
<tr>
<td>0x00A</td>
<td>PCLATH</td>
<td>PCLATH</td>
<td></td>
</tr>
<tr>
<td>0x00B</td>
<td>INTCON</td>
<td>INTCON</td>
<td></td>
</tr>
</tbody>
</table>

The File Registers (Variable registers) start at either 0x00C or 0x020 of the bank depending on the “Hardware I/O” or “Special Function Registers” (“SFRs”) built into the device. It is recommended to start all variable declarations at 0x020 to avoid issues porting between a PICmicro® MCU that has file registers starting at 0x00C and one that has file registers starting at 0x020.

**STATUS register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>IRP - FSR Select Between the High and Low Register Banks</td>
</tr>
<tr>
<td>6-5</td>
<td>RPI:RPO - Direct Addressing Select Banks (0 through 3)</td>
</tr>
<tr>
<td>4</td>
<td>_TO - Time Out Bit. Reset after a Watchdog Timer Reset</td>
</tr>
</tbody>
</table>
Program counter

The mid-range PICmicro® MCU's program counter can be represented by the block diagram shown in Fig. 5.6.

![Block Diagram of Program Counter](image-url)
134 Chapter 5

To jump to another bank, the following instruction sequence is used:

<table>
<thead>
<tr>
<th>Mid-range PICmicro® MCU Program Counter Update Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCLATH = (HIGH new_address;</td>
</tr>
<tr>
<td>PCL    = LOW new_address;</td>
</tr>
</tbody>
</table>

All addresses within the mid-range PICmicro® MCU can be accessed with the PCLATH and PCL registers.

The “call stack” is eight elements deep.

**Interrupt operation**

Interrupts are controlled by the state of the INTCON Register and optionally the “PIE” and “PIR” registers. Interrupt handlers always start executing at address 0x004.

<table>
<thead>
<tr>
<th>Mid-Range INTCON Register Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

For an Interrupt Request (which sets the bit ending in “F”), the corresponding “Enable” bit (which is the bit ending in “E”) has to be set along with the “GIE” bit.
Some Enable and Interrupt Request bits may be in auxiliary registers or “PIR” or “PIE”.

Interrupt handler skeleton

The mid-range PICmicro® MCU has an interrupt skeleton of:

```assembly
or 4
movwf _w ; Save Context Registers
movf STATUS, w
bcf STATUS, RP1 ; Make Bank 0 Active
bcf STATUS, RP0
movwf _status
movf FSR, w
movwf_fsr
movf PCLATH, w
movwf _pclath
clrf PCLATH ; Make sure Execution in Page 0
; ; Execute Interrupt Handler
movf _pclath, w ; Restore the Context Registers
movf PCLATH
movf_fsr, w
movf FSR
movf _status, w
movwf STATUS
swapf _w, f
swapf _w, w
retfie
```

To enable the TMR0 Interrupt Request, the following code is used:

```
clrf TMR0 ; Reset TMR0
bcf INTCON, TOIF ; Reset TMR0 Interrupt Request
bsf INTCON, TOIE ; Enable TMR0 Interrupt Request
bsf INTCON, GIE ; Enable PICmicro® MCU Interrupts
```
Chapter 5

PIC17Cxx

The PIC17Cxx architecture encompasses parts with the part numbers:

    PIC17Cxx(x)

The unique features of the PIC17Cxx, as compared to the other PICmicro® MCUs, include:

1. The ability to access external, parallel memory.
2. Up to seven I/O ports.
3. A built-in \(8 \times 8\) multiplier.
4. Up to 902 file registers in up to 16 banks.
5. Up to 64K address space.
6. The ability to read and write program memory.
7. Multiple interrupt vectors.

The PIC17Cxx processor has a block diagram as shown in Fig. 5.7.

The important differences between the PIC17Cxx architecture and the low-end and mid-range PICmicro® MCU architectures are as follows:

1. The accumulator, “WREG,” can be addressed in the register space.
2. The STATUS and OPTION register functions are spread across different registers.
3. The program counter works slightly differently from the other architectures.
4. The “registers” are accessed differently and accesses can bypass the “WREG”.

The reset address of the PIC17Cxx is 0x00000.

**Register access**

The PIC17Cxx has a single 256 Register address space. Addresses 0x010 to 0x017 are banked and addressed using the lower nybble of the “BSR” register and contain the register selection information of the “Special Function Registers” (“SFRs”) or Hardware I/O Registers. The high nybble of BSR is used to select the File Register Bank at addresses 0x020 to 0x0FF.
The PIC17Cxx has two registers that provide the same functions as the single “STATUS” register of the other three PICmicro® MCU architectures. The PIC17Cxx Bank Selection is made by the “BSR” Register.
### PIC17Cxx ALUSTA Register Definition

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>FSR1 Mode Select</td>
</tr>
<tr>
<td>5-4</td>
<td>FSR0 Mode Select</td>
</tr>
<tr>
<td>3</td>
<td>OV - Set when there is a two's complement overflow after addition/subtraction</td>
</tr>
<tr>
<td>2</td>
<td>Z - Set when the eight bit result is equal to Zero</td>
</tr>
<tr>
<td>1</td>
<td>DC - Set for low order Nybble carry after addition or subtraction instruction</td>
</tr>
<tr>
<td>0</td>
<td>C - Set for Carry after addition or subtraction instruction</td>
</tr>
</tbody>
</table>

### PIC17Cxx CPUSTA Register Definition

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Unused</td>
</tr>
<tr>
<td>5</td>
<td>STRAV - When Set, there is Program Counter Stack Space Available</td>
</tr>
<tr>
<td>4</td>
<td>GLINTD - When Set, all Interrupts Are Disabled</td>
</tr>
<tr>
<td>3</td>
<td>_TO - Set after Power Up or clrwdt Instruction. When Reset a Watchdog Timeout has occurred</td>
</tr>
<tr>
<td>2</td>
<td>_PD - Set after Power Up or clrwdt Instruction. Reset by a &quot;sleep&quot; instruction</td>
</tr>
<tr>
<td>1</td>
<td>_POR - Reset After Power Up in PIC17C5x. Not Available in All PIC17Cxx devices</td>
</tr>
<tr>
<td>0</td>
<td>_BOR - Reset After Brown Out Reset. Not Available in All PIC17Cxx devices</td>
</tr>
</tbody>
</table>
Program counter

The PIC17Cxx's processor can access 64k 16-bit words of program memory, either internally or externally to the chip. Each instruction word is given a single address; so to address the 64k words (or 128k bytes), 16 bits are required. From the application developer's perspective, these 16 bits can be accessed via the "PCL" and "PCLATH" registers in exactly the same way as the low-end and mid-range PICmicro® MCUs. The PIC17Cxx's program counter block diagram is shown in Fig. 5.8.

The block diagram in Fig. 5.8 differs from the mid-range PICmicro® MCU's program counter block diagram in one important respect; when the "goto" and "call" instructions are executed, the upper 5 bits of the specified instruction overwrite the lower 5 bits of the PCLATH register. After execution of a "goto" or "call" instruction PCLATH has been changed to the current address.

![Figure 5.8 PIC17Cxx Program Counter](image-url)
Interrupt operation

The PIC17Cxx can have four different interrupt vector addresses, depending on their source and priority. The Interrupts and their vectors are listed below:

### PIC17Cxx Interrupt Vector Address and Priorities for Different Sources

<table>
<thead>
<tr>
<th>Priority</th>
<th>Vector Address</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>0x0008</td>
<td>RA0/INT Pin Interrupt</td>
</tr>
<tr>
<td></td>
<td>0x00010</td>
<td>TMRO Overflow Interrupt</td>
</tr>
<tr>
<td></td>
<td>0x00018</td>
<td>TOCKI Pin Interrupt</td>
</tr>
<tr>
<td>Low</td>
<td>0x00020</td>
<td>Peripheral Device</td>
</tr>
</tbody>
</table>

### PIC17Cxx CPUSTA Register Definition

<table>
<thead>
<tr>
<th>Any 0x007 INTSTA Interrupt Status and Control Register Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 PEIE - Set when Peripheral Interrupt is Pending</td>
<td></td>
</tr>
<tr>
<td>6 T0CKIF - Set when RA1/TOCKI Pin has Interrupt Source.</td>
<td>Cleared by Hardware when Interrupt Vector 0x0018 is executed</td>
</tr>
<tr>
<td>5 T0IF - Set when TMRO Overflows. Cleared by Hardware</td>
<td>When Interrupt Vector 0x0010 is executed</td>
</tr>
<tr>
<td>4 INTF - Set when RA0/INT Pin Interrupt Request Active.</td>
<td>Cleared by Hardware when Interrupt Vector 0x0008 is executed</td>
</tr>
<tr>
<td>3 PEIE - Set to Enable Peripheral Interrupt Requests</td>
<td></td>
</tr>
</tbody>
</table>
142 Chapter 5

PIC17Cxx CPUSTA Register Definition (Continued)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>TOCKIE - Set to Enable RA1/T0CKI Interrupt Request</td>
</tr>
<tr>
<td>1</td>
<td>T0IE - Set to Enable TMR0 Overflow Interrupt Request</td>
</tr>
<tr>
<td>0</td>
<td>INTE - Set to Enable RA0/INT Pin Interrupt Request</td>
</tr>
</tbody>
</table>

Interrupt handler skeleton

```assembly
org ?? ; Vector According to Source
movpf ALUSTA, _alusta ; Save Context Registers
movpf WREG, _w
movpf BSR, _bar
movpf PCLATH, _pclath
clrf PCLATH ; Make sure Execution in Page 0
movfp _pclath, PCLATH ; Execute Interrupt Handler
movfp _bar, BSR
movfp _w, WREG
movfp _alusta, ALUSTA
retfie
```

PIC18Cxx

The PIC18Cxx architecture encompasses parts with the part numbers:

PIC18Cxx2

The unique features of the PIC18Cxx, as compared to the other PICmicro® MCUs, include:

1. A built-in $8 \times 8$ multiplier.
2. Up to 3,840 file registers in up to 16 banks.
3. Up to 1,048,576 words of program memory address space.
4. The ability to read and write program memory.
5. Prioritized Interrupt Requests.

The PIC18Cxx processor has a block diagram as shown in Fig. 5.9. The important differences between the PIC18Cxx architecture and the low-end and mid-range PICmicro® MCU architectures are as follows:

1. The accumulator, “WREG”, can be addressed in the register space.
2. The “Access Bank”, which is used to allow access to the first 128 file registers and the Hardware I/O registers without involving the BSR.
3. The program counter works slightly differently from the other architectures.

4. The “registers” are accessed differently and accesses can bypass the “WREG”.

The reset address of the PIC18Cxx is 0x00000.

Register access

The PIC18Cxx can access up to 4,096 8-bit registers that are available in a contiguous memory space. Twelve address bits are used to access each address within the “Register Map” space shown in Fig. 5.10.

To access a register directly, the PIC18Cxx’s “BSR” (“Bank Select Register”) register must be set to the bank the register is located in. The BSR register contains the upper 4 bits of the register’s address, with the
lower 8 bits explicitly specified within the instruction. The direct address is calculated using the formula:

\[
\text{Address} = (\text{BSR} \ll 8) + \text{Direct Address}
\]

The index register operation of the PIC18Cxx is very well organized and will make it much easier for compiler writers to create PIC18Cxx compilers than for other PICmicro® MCUs. Along with the three 12-bit-long FSR registers, when data is accessed it can result in the FSR being incremented before or after the data access, decremented after or access to the address of the FSR contents added to the contents of the “w” register. A specific access option is selected by accessing different “INDF” register addresses. The table below lists the different INDF registers and their options concerning their respective FSR registers:

<table>
<thead>
<tr>
<th>PIC18Cxx FSR Change Access Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDF Register Operation</td>
</tr>
<tr>
<td>INDF# Access the Register Pointed to by FSR#</td>
</tr>
<tr>
<td>POSTINC# Access the Register Pointed to by FSR# and then Increment FSR#</td>
</tr>
<tr>
<td>POSTDEC# Access the Register Pointed to by FSR# and then Decrement FSR#</td>
</tr>
<tr>
<td>PREINC# Increment FSR# and then Access the Register Pointed to by FSR#</td>
</tr>
<tr>
<td>PLUSW# Access the Register Pointed to by the Contents of the WREG added to FSR#</td>
</tr>
</tbody>
</table>

To simulate a “push” of the contents of the “WREG” using FSR0 as a Stack Pointer, use the operation:

\[
\text{POSTDEC0 = WREG;}
\]

A “pop WREG” could be implemented as:

\[
\text{WREG = PREINCO;}
\]
146 Chapter 5

Specific elements relative to the start of the stack could be accessed using the code:

```c
WREG = 3;
WREG = PLUSWO;
```

**STATUS register**

Two registers contain the status information for the PIC18Cxx and control the operation of the PICmicro® MCU.

### PIC18Cxx STATUS Register Definition

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-5</td>
<td>Unused</td>
</tr>
<tr>
<td>4</td>
<td>N - Set when the two's complement result after addition/subtraction is negative</td>
</tr>
<tr>
<td>3</td>
<td>OV - Set when there is a two's complement overflow after addition/subtraction</td>
</tr>
<tr>
<td>2</td>
<td>Z - Set when the eight bit result is equal to zero</td>
</tr>
<tr>
<td>1</td>
<td>DC - Set when the low nybble of addition/subtraction result carries to the high nybble</td>
</tr>
<tr>
<td>0</td>
<td>C - Set when the addition/subtraction result carries to the next byte. Also used with the Rotate Instructions</td>
</tr>
</tbody>
</table>

### PIC18Cxx RCON Register Definition

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>IFBN - When Set Interrupt Priority Levels are enabled</td>
</tr>
<tr>
<td>6</td>
<td>LWRT - When Set, Enable writes to internal program memory</td>
</tr>
<tr>
<td>5</td>
<td>Unused</td>
</tr>
<tr>
<td>4</td>
<td>RI - When Reset, the “Reset” Instruction was executed. This bit must be set in Software</td>
</tr>
</tbody>
</table>
Program counter

The PIC18Cxx program counter and stack is similar to the hardware used in the other devices except for three important differences. The first difference is the need for accessing more than 16 address bits for the maximum one million possible instructions of program memory. The second difference is the availability of the “fast stack”, which allows interrupt context register saves and restores to take place without requiring any special code. The last difference is the ability to read and write from the stack. These differences add a lot of capabilities to the 18Cxx that allow applications that are not possible in the other PICmicro® MCU architectures to be implemented.

In the PIC18Cxx, when handling addresses outside the current program counter, not only does a “PCLATH” register (or “PA” bits as in the low-end devices) update as required, but it is also a high-order register for addresses above the first 64 instruction words. This register is known as “PCLATU”. “PCLATU” works identically to the “PCLATH” register and its contents are loaded into the PIC18Cxx PICmicro® MCUs program counter when “PCL” is updated.

Each instruction in the PIC18Cxx starts on an “even” address. This means that the first instruction starts at ad-
address zero, the second at address two, the third at address four and so on. Setting the program counter to an odd address will result in the MPLAB simulator halting and the PIC18Cxx working unpredictably. Changing the convention used in the previous PICmicro® MCUs to one, where each byte is addressed, means that some rules about addressing will have to be relearned for the PIC18Cxx.

The stack itself, at 31 entries, is deeper than the other PICmicro® MCU stacks and the hardware monitoring the stack is available as the "STKPTR" register. A block diagram of the stack is shown in Fig. 5.11.

The STKPTR register is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>STKFUL - Stack Full Flag which is set when the Stack is Full or Overflowed</td>
</tr>
</tbody>
</table>
The "STKUNF" and "STKFUL" bits will be set if their respective conditions are met. If the "STVREN" bit of the configuration fuses is set, then when the STKUNF and STKFUL conditions are true the PICmicro® MCU will be reset.

The “fast stack” is used to simplify subroutine calls in applications that don’t have interrupts enabled as well as working with interrupt handlers. To use the fast stack in the “call” and “return” instructions a “1” parameter is put at the end of the instructions. To prevent the fast stack from being used, a “0” parameter is put at the end of the “call” and “return” instructions. The “fast stack” is a 3-byte memory location where the “w”, “STATUS” and “BSR” registers are stored automatically when an interrupt request is acknowledged and execution jumps to the interrupt vector. If interrupts are not used in an application, then these registers can be saved or restored with a “call” and “return”, for instance:

```
Call sub, 1 ; Call "sub" after saving "w", "STATUS"
; and "BSR"

sub:
    ; Execute "Sub", Ignore "w", "STATUS"
    ; and "BSR"

return 1 ; Restore "w", "STATUS" and "BSR" before
        ; Return to Caller
```

<table>
<thead>
<tr>
<th>PIC18Cxx STKPTR Register Bit Definitions (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4-0</td>
</tr>
</tbody>
</table>

PICmicro® MCU Processor Architectures 149
The reason the “fast” option is not recommended in applications in which interrupts are enabled is due to the interrupt overwriting the saved data when it executes. For this reason, the “fast” option cannot be used with nested subroutines or interrupts.

Interrupt operation

Interrupt Operation works similarly to the mid-range PICmicro® MCU’s except for the addition of priority levels to the interrupt sources. If the “P” bit for the interrupt source is specified along with the “IPEN” bit of the RCON register, then the interrupt handler at address 0x00008 will execute. If the “IPEN” bit of the RCON register is set and the “P” bit for the interrupt source is reset, the interrupt handler at address 0x00018 will execute.

If “IPEN” is reset, then all interrupts will execute at address 0x00008.

Interrupt handler skeleton

If the “Fast Stack” is not used with PIC18Cxx interrupts, the code shown below can be used for the handler entry and exit code.

```assembly
Int
    movwf _w                      ; Save Context Registers
    movff STATUS, _status
    movff BSR, _bsr
    ;                       ; Interrupt Handler Code
    movff _bsr, BSR             ; Restore Context Registers
    movf _w, w
    movff _status, STATUS
    retfie
```
While the register addresses are very similar between PICmicro® MCUs of the same architecture family, remember that the bits in the different registers may change function with different PICmicro® MCU part numbers. To be absolutely sure of the bits and their function inside a register, consult the Microchip part datasheet.

Low-End PICmicro® MCUs

The low-end PICmicro® MCU devices have five register bank address bits for up to 32 unique file register ad-
addresses in each bank. Up to four register banks can be available in a low-end PICmicro® MCU with the first 16 addresses of each bank being common throughout the banks and the second 16 addresses being unique to the bank. This is shown in Fig. 6.1.

Using this scheme, low-end PICmicro® MCUs have anything from 25 to 73 unique file registers available for an application.

There are a few things to note with the low-end register addressing:

1. The “OPTION” and “TRIS” registers can only be written to by the “option” and “tris” instructions, respectively.

2. If the device has a built-in oscillator, the “OSCCAL” register is located in address five, which is normally the “PORTA” address.

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr - Reg</td>
<td>Addr - Reg</td>
<td>Addr - Reg</td>
<td>Addr - Reg</td>
</tr>
<tr>
<td>00 - TMDR</td>
<td>01 - TMDR</td>
<td>00 - TMDR</td>
<td>01 - TMDR</td>
</tr>
<tr>
<td>02 - TMDR</td>
<td>03 - TMDR</td>
<td>02 - TMDR</td>
<td>03 - TMDR</td>
</tr>
<tr>
<td>04 - STATUS</td>
<td>05 - STATUS</td>
<td>04 - STATUS</td>
<td>05 - STATUS</td>
</tr>
<tr>
<td>06 - PORTA*</td>
<td>07 - PORTA*</td>
<td>06 - PORTA*</td>
<td>07 - PORTA*</td>
</tr>
<tr>
<td>08 - PORTB</td>
<td>09 - PORTB</td>
<td>08 - PORTB</td>
<td>09 - PORTB</td>
</tr>
<tr>
<td>10 - PORTC</td>
<td>11 - PORTC</td>
<td>10 - PORTC</td>
<td>11 - PORTC</td>
</tr>
</tbody>
</table>

*OSCCAL* may take place of *PORTA* in PICmicro® MCUs with internal oscillators

<table>
<thead>
<tr>
<th>OPTION</th>
<th>TRIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessed via “option” Instruction</td>
<td>Accessed via “TRIS PORTA” Instruction</td>
</tr>
</tbody>
</table>

**Figure 6.1 Low-end PICmicro® MCU Register Map**
### PICmicro MCU® Register Mappings

3. The “STATUS” and “OPTION” registers are always the same for Low-End Devices.

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Bits</th>
<th>Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x003</td>
<td>STATUS</td>
<td>7</td>
<td>GPWUF - in PIC12C5xx and PIC16C505: when Set, Reset from Sleep on Pin Change. When Set, power up or MCLR reset. In other Devices the Bit 7 is Unused.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6-5</td>
<td>6-5 PA1-PA0 - Select the Page to execute out of: 00 - Page 0 (0x0000 to 0x01FF) 01 - Page 1 (0x0200 to 0x03FF) 10 - Page 2 (0x0400 to 0x05FF) 11 - Page 3 (0x0600 to 0x07FF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>_TO - Set after Power Up, clrwdt and sleep instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>_PD - Set after Power Up, clrwdt instruction. Reset after sleep instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>_Z - Set if the eight bit result is equal to zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>_DC - Set for low order Nybble carry after addition or subtraction instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>_C - Set for Carry after addition or subtraction instruction</td>
</tr>
<tr>
<td>N/A</td>
<td>OPTION</td>
<td>7</td>
<td>GPWU - in PIC12C5xx or PIC16C505: Reset to Enable Wake up on Pin Change. In other devices, Bit 7 is Unused</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>_GPFU - In PIC12C5xx or PIC16C505: Enable Pin Pull-Ups. In other devices, Bit 6 is Unused</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>T0CS - TMR0 Clock Source Select. When Set, T0CKI pin is Source. When Reset, Instruction Clock</td>
</tr>
</tbody>
</table>
Chapter 6

4. The Low-End PICmicro® MCU FSR register can never equal zero.

Mid-Range PICmicro® MCUs

The standard mid-range PICmicro® MCU device addresses are as follows:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>INDF</td>
<td>INDF</td>
<td></td>
</tr>
<tr>
<td>0x001</td>
<td>TMRO</td>
<td>OPTION</td>
<td></td>
</tr>
<tr>
<td>0x002</td>
<td>PCL</td>
<td>PCL</td>
<td></td>
</tr>
<tr>
<td>0x003</td>
<td>STATUS</td>
<td>STATUS</td>
<td></td>
</tr>
<tr>
<td>0x004</td>
<td>FSR</td>
<td>FSR</td>
<td></td>
</tr>
<tr>
<td>0x005</td>
<td>PORTA</td>
<td>TRISA</td>
<td>Available in 28/40 Pin Parts</td>
</tr>
<tr>
<td>0x006</td>
<td>PORTB</td>
<td>TRISB</td>
<td></td>
</tr>
<tr>
<td>0x007</td>
<td>PORTC</td>
<td>TRISC</td>
<td>Available in 28/40 Pin Parts</td>
</tr>
<tr>
<td>0x008</td>
<td>PORTD</td>
<td>TRISD</td>
<td>Available in 40 Pin Parts</td>
</tr>
<tr>
<td>0x009</td>
<td>PORTE</td>
<td>TRISE</td>
<td>Available in 40 Pin Parts</td>
</tr>
<tr>
<td>0x00A</td>
<td>PCLATH</td>
<td>PCLATH</td>
<td></td>
</tr>
<tr>
<td>0x00B</td>
<td>INTCON</td>
<td>INTCON</td>
<td></td>
</tr>
</tbody>
</table>
From these basic addresses, peripheral I/O registers (discussed below) are added to the register banks with file registers starting at either offset 0x00C or 0x020. For most modern mid-range PICmicro® MCUs, the file registers start at address 0x020 of the bank.

The specific part number datasheets will have to be checked to find where the file registers that are shared across the banks are located.

The STATUS Register, in the mid-range PICmicro® MCU is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>IRP - FSR Select Between the High and Low Register Banks</td>
</tr>
<tr>
<td>6-5</td>
<td>RP1.RP0 - Direct Addressing Select Banks (0 through 3)</td>
</tr>
<tr>
<td>4</td>
<td>_TO - Time Out Bit. Reset after a Watchdog Timer Reset</td>
</tr>
<tr>
<td>3</td>
<td>_PD - Power-down Active Bit. Reset after sleep instruction</td>
</tr>
<tr>
<td>2</td>
<td>Z - Set when the eight bit result is equal to zero</td>
</tr>
<tr>
<td>1</td>
<td>DC - Set when the low Nybble of addition/subtraction result carries to the high Nybble</td>
</tr>
<tr>
<td>0</td>
<td>C - Set when the addition/subtraction result carries to the next byte. Also used with the Rotate Instructions</td>
</tr>
</tbody>
</table>

The OPTION Register (which has the label “OPTION_REG” in the Microchip include files) is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>_RBPU - when reset, the PORTB Pin Pull Up is Enabled</td>
</tr>
<tr>
<td>6</td>
<td>INTEDG - When Set, Interrupt Request on Rising Edge of RB0/INT Pin</td>
</tr>
<tr>
<td>5</td>
<td>TOCS - When Set, TMR0 is incremented from the TOCKI Pin, else by the internal instruction clock</td>
</tr>
<tr>
<td>4</td>
<td>TOSE - When Set, TMR0 is Incremented on the High to Low (&quot;Falling Edge&quot;) of TOCKI</td>
</tr>
</tbody>
</table>
Chapter 6

3 PSA - Prescaler Assignment Bit. When Set, the
Prescaler is assigned to the Watchdog Timer
else to TMR0

2-0 PS2:PS0 - Prescaler Rate Select.

<table>
<thead>
<tr>
<th>Bits</th>
<th>TMR0 Rate</th>
<th>WDT Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>256:1</td>
<td>128:1</td>
</tr>
<tr>
<td>110</td>
<td>128:1</td>
<td>64:1</td>
</tr>
<tr>
<td>101</td>
<td>64:1</td>
<td>32:1</td>
</tr>
<tr>
<td>000</td>
<td>32:1</td>
<td>16:1</td>
</tr>
<tr>
<td>011</td>
<td>16:1</td>
<td>8:1</td>
</tr>
<tr>
<td>010</td>
<td>8:1</td>
<td>4:1</td>
</tr>
<tr>
<td>001</td>
<td>4:1</td>
<td>2:1</td>
</tr>
<tr>
<td>000</td>
<td>2:1</td>
<td>1:1</td>
</tr>
</tbody>
</table>

Many devices have the “PCON” register that enhances
the returned information contained in the “_TO” and
“_PD” bits of the STATUS Register

Bit Function
7 MPEEN - Set if there is a Memory Parity Error.
This capability is built into a small number
of PICmicro® MCUs
6-3 Unused
2 _PER - Reset when there was a Program Memory
Parity Error. This capability is built into a
small number of PICmicro® MCUs
1 _POR - Reset when execution is from a Power On
Reset takes place
0 _BOR - Reset when execution is from a Brown
Out Reset

The PCLATH Register's contents are written to the
Program Counter each time a “goto” or “call” instruction
is executed or if the contents of PCL are changed.

Bit Function
7-5 Unused
4 Select High and Low Pages
3 Select Odd or Even Pages
2-0 Select the 256 Instruction Address Block
within Current Page. This data is used when
PCL is written to
Some mid-range devices are now available with built-in RC oscillators. To make the operation of the oscillators more accurate, the “OSCCAL” register is written to with a factory specified “Calibration Value”.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>CAL3:CAL0 - Sixteen Bit Calibration Value</td>
</tr>
<tr>
<td>3</td>
<td>CALFST - Increase the speed of the RC Oscillator</td>
</tr>
<tr>
<td>2</td>
<td>CALEOW - Decrease the speed of the RC Oscillator</td>
</tr>
<tr>
<td>1-0</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Interrupts are controlled from the “INTCON” register, which controls the basic mid-range PICmicro® MCU interrupts as well as access to enhanced interrupt features.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GIE - Global Interrupt Enable. For any Interrupt Requests to be acknowledged, this bit must be set</td>
</tr>
<tr>
<td>6</td>
<td>Device Specific Interrupt Enable. See Below</td>
</tr>
<tr>
<td>5</td>
<td>TOIE - TMR0 Interrupt Overflow Request Enable</td>
</tr>
<tr>
<td>4</td>
<td>INTE - RB0/INT Pin Interrupt Request Enable</td>
</tr>
<tr>
<td>3</td>
<td>RBIE - PORTB Change Interrupt Request Enable</td>
</tr>
<tr>
<td>2</td>
<td>T0IF - TMR0 Interrupt Overflow Request</td>
</tr>
<tr>
<td>1</td>
<td>INTF - RB0/INT Pin Interrupt Request</td>
</tr>
<tr>
<td>0</td>
<td>RBIF - PORTB Change Interrupt Request</td>
</tr>
</tbody>
</table>

Bit 6 of INTCON may be a peripheral device interrupt enable/request bit or it can be “PEIE”, which when set will enable Peripheral Interrupts set in “PIR” and “PIE” registers. The “PIR” register(s) contains the “F” bits (Interrupt Request Active), while “PIE” contains the “E” bits (Interrupt Request Enable). As I work through the different peripherals, the “E” and “F” bits will be listed, but their actual location is part number specific and the datasheet will have to be consulted.

Data EEPROM is accessed via the EEADR and
EEDATA registers with EECON1 and EECON2 providing the access control. EECON2 is a “pseudo-register” and the act of writing to it is used to “verify” that the operation request is valid. EECON1 is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-5</td>
<td>Unused</td>
</tr>
<tr>
<td>4</td>
<td>EEIF - EEPROM Write Complete Interrupt Request</td>
</tr>
<tr>
<td>3</td>
<td>WRERR - Bit Set when EEPROM Write was invalid</td>
</tr>
<tr>
<td>2</td>
<td>WREN - Set to Enabling Writing to EEPROM</td>
</tr>
<tr>
<td>1</td>
<td>WR - Write control Bit</td>
</tr>
<tr>
<td>0</td>
<td>RD - Set to Allow an EEPROM Data Read</td>
</tr>
</tbody>
</table>

The Data EEPROM Write Interrupt Request Bit (“EEIE”) is either in a PIE register or INTCON.

The Parallel Slave Port (“PSP”, available only in 40-Pin mid-range PICmicro® MCUs) is enabled by setting the PSPMODE bit. Interrupt requests are enabled by the PSPIE flag and requested by the PSPIF flag of the PIE and PIR registers, respectively. The Parallel Slave Port is controlled from “TRISE”. Note that when the Parallel Slave Port is enabled, PORTD and PORTE cannot be used for I/O.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>IBF - Bit Set when a Word has been Written</td>
</tr>
<tr>
<td></td>
<td>into the PICmicro® MCU and has not been read</td>
</tr>
<tr>
<td>6</td>
<td>OBPF - Bit Set when a Byte has been written to</td>
</tr>
<tr>
<td></td>
<td>the PORTD Output Register and has not been</td>
</tr>
<tr>
<td></td>
<td>read</td>
</tr>
<tr>
<td>5</td>
<td>IBOV - Bit Set when a Word has been Written</td>
</tr>
<tr>
<td></td>
<td>into the PICmicro® MCU before the previous one</td>
</tr>
<tr>
<td></td>
<td>has been read</td>
</tr>
<tr>
<td>4</td>
<td>PSPMODE - Bit set to enable Parallel Slave</td>
</tr>
<tr>
<td></td>
<td>Port</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>TRISE2 - TRIS Bit for E2</td>
</tr>
<tr>
<td>1</td>
<td>TRISE1 - TRIS Bit for E1</td>
</tr>
<tr>
<td>0</td>
<td>TRIS0 - TRIS Bit for E0</td>
</tr>
</tbody>
</table>
Along with TMR0, some mid-range PICmicro® MCU's have TMR1 and TMR2, which are used for basic timing operations as well as “CCP” (“Compare, Capture, and PWM”) I/O. TMR1 is a 16-bit-wide register (accessed via “TMR1L” and “TMR1H”) that will request an interrupt on overflow (“TMR1IF”) if the “TMR1IE” bit is set. The T1CON register that is defined below controls the operation of TMR1:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Unused</td>
</tr>
<tr>
<td>5-4</td>
<td>T1CKPS1:T1CKPS2: TMR1 Input Prescaler Select</td>
</tr>
<tr>
<td>3</td>
<td>T1OSCEN - Set to Enable External TMR1 Oscillator</td>
</tr>
<tr>
<td>2</td>
<td>_T1SYNC - If External Clock used for TMR1, then Synchronize to it when this bit is Reset</td>
</tr>
<tr>
<td>1</td>
<td>TMR1CS - When Set, TMR1 is driven by External Clock/TMR1 Oscillator</td>
</tr>
<tr>
<td>0</td>
<td>TMR1ON - Set to Enable TMR1</td>
</tr>
</tbody>
</table>

TMR2 is an 8-bit register that is continually compared against a value in the PR2 register. To have TMR2 operate like TMR0 as an 8-bit Timer with a range of 0x000 to 0x0FF, then the “PR2” (the register TMR2 is compared against) is set to 0x000. The TMR2 output can be used to drive a PWM signal out. Interrupts (“TMR2IF”) can be requested after the TMR2 overflow has passed through a Postscaler and “TMR2IE” is set. The T2CON register controls the operation of TMR2:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Unused</td>
</tr>
<tr>
<td>6-3</td>
<td>TOUTPS3:TOUTPS0 - TMR2 Output Postscaler Select</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Postscaler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>16:1</td>
</tr>
<tr>
<td>1110</td>
<td>15:1</td>
</tr>
<tr>
<td>1101</td>
<td>14:1</td>
</tr>
<tr>
<td>1100</td>
<td>13:1</td>
</tr>
</tbody>
</table>
TMR1 and TMR2 are used with one of the two “CCP” Modules for advanced I/O. TMR1 is used for Capture and Compare and TMR2 is used for PWM Output. The CCPR2x Registers are used for Storing Compare/Capture Values and the CCPx register specifies the Pin used for CCP. The CCPxCON register is used for controlling the CCP operation:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Unused</td>
</tr>
<tr>
<td>5-4</td>
<td>DCxB1:DCxB0 - PWM Duty Cycle Bit 1 and Bit 0.</td>
</tr>
<tr>
<td></td>
<td>These bits are only accessed by the PWM for</td>
</tr>
<tr>
<td></td>
<td>its low output values</td>
</tr>
<tr>
<td>3-0</td>
<td>CCPxM3:CCPxM0 - CCPx Mode Select</td>
</tr>
<tr>
<td>Bits</td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td>lxx</td>
</tr>
<tr>
<td>1011</td>
<td>Compare Mode, Trigger Special Event</td>
</tr>
<tr>
<td>1010</td>
<td>Compare Mode, Trigger on Compare Match</td>
</tr>
<tr>
<td>1001</td>
<td>Compare Mode, Initialize CCP Pin High,</td>
</tr>
<tr>
<td></td>
<td>On Compare Match force CCP Low</td>
</tr>
<tr>
<td>1000</td>
<td>Compare Mode, Initialize CCP Pin Low,</td>
</tr>
<tr>
<td></td>
<td>On Compare Match force CCP High</td>
</tr>
<tr>
<td>0111</td>
<td>Capture on Every 16th Rising Edge</td>
</tr>
</tbody>
</table>
CCP Interrupts are requested via the “CCPxIF” flag and enabled by the “CCPXIE” flag where “x” is “1” or “2” depending on the active CCP Module.

There are three different “SSP” Modules built into the PICmicro® MCU. Each one provides somewhat different Options and understanding how they work will be critical to your applications and if I2C is going to be used with them. The basic SSP modules (“SSP” and “BSSP”) provide a full SPI Interface and I2C “Slave Mode” Interface. The SSPBUF Register provides simple buffering while the SSPADD buffers the received address for comparing against I/O operations. To control the Operation of the SSP, the “SSPCON” register is used:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>WCOL - Set if SSPBUF was written to while transmitting data or not in correct mode for transmit</td>
</tr>
<tr>
<td>6</td>
<td>SSPOV - Set when SSP Receive overflow occurs</td>
</tr>
<tr>
<td>5</td>
<td>SSPEN - Enables Pins for SSP Mode</td>
</tr>
<tr>
<td>4</td>
<td>CKP - In SPI, Set for Idle Clock High. In I2C Mode, set to Enable Clock</td>
</tr>
<tr>
<td>3-0</td>
<td>SSPM3:SSPM0 - SSP Mode Select</td>
</tr>
<tr>
<td>1111</td>
<td>I2C Slave Mode, 10 Bit Address</td>
</tr>
<tr>
<td>1110</td>
<td>I2C Slave Mode, 7 Bit Address</td>
</tr>
<tr>
<td>110x</td>
<td>Reserved</td>
</tr>
<tr>
<td>1101</td>
<td>I2C firmware controlled Master</td>
</tr>
<tr>
<td>1100</td>
<td>Reserved</td>
</tr>
<tr>
<td>1011</td>
<td>I2C Slave Mode, 10 Bit Address</td>
</tr>
<tr>
<td>1010</td>
<td>I2C Slave Mode, 7 Bit Address</td>
</tr>
<tr>
<td>1001</td>
<td>SSP Slave, _SS Disabled</td>
</tr>
<tr>
<td>1000</td>
<td>SSP Slave, _SS Enabled</td>
</tr>
</tbody>
</table>
Chapter 6

0011 - SPI Master, Clock = TMR2
0010 - SPI Master, Fosc/64
0001 - SPI Master, Fosc/16
0000 - SPI Master, Fosc/4

The SSPSTAT Register is also used to Control the SSP:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SMP - Data Sampled at end of data output time if Set, else middle</td>
</tr>
<tr>
<td>6</td>
<td>CKE - Data transmitted on rising edge of SCK when Set</td>
</tr>
<tr>
<td>5</td>
<td>D_/A - Used by I2C. When Set indicates last byte transferred was data. When Reset indicates last byte transferred was address</td>
</tr>
<tr>
<td>4</td>
<td>P - Set when Stop Bit Detected</td>
</tr>
<tr>
<td>3</td>
<td>S - Set when Start Bit Indicated</td>
</tr>
<tr>
<td>2</td>
<td>R_/W - Set when command received was a Read</td>
</tr>
<tr>
<td>1</td>
<td>UA - Set when application must update SSPADD Register</td>
</tr>
<tr>
<td>0</td>
<td>BF - Set when Buffer is full in RX and when TX is in process</td>
</tr>
</tbody>
</table>

The “Master” SSP (“MSSP”) accesses similar registers for the same functions, with a Second SSPCON Register. The important difference between the MSSP and the other SSP modules is the enabled I2C Master hardware in the MSSP. The MSSP’s “SSPCON1” register is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>WCOL - Set if SSPBUF was written to while transmitting data or not in correct mode for transmit</td>
</tr>
<tr>
<td>6</td>
<td>SSPOV - Set when SSP Receive overflow occurs</td>
</tr>
<tr>
<td>5</td>
<td>SSPEN - Enables Pins for SSP Mode</td>
</tr>
<tr>
<td>4</td>
<td>CKP - In SPI, Set for Idle Clock High. In I2C Mode, set to Enable Clock</td>
</tr>
<tr>
<td>3-0</td>
<td>SSPM3:SSPM0 - SSP Mode Select</td>
</tr>
</tbody>
</table>

0111 - Reserved
1001 - I2C Master Mode, Clock = Fosc/(4 * (SSPADD + 1))
**PICmicro MCU® Register Mappings 163**

0111 - I2C Slave Mode, 10 Bit Address  
0110 - I2C Slave Mode, 7 Bit Address  
0101 - SSP Slave, _SS Disabled  
0100 - SSP Slave, _SS Enabled  
0011 - SPI Master, Clock = TMRO  
0010 - SPI Master, Fosc/64  
0001 - SPI Master, Fosc/16  
0000 - SPI Master, Fosc/4

SSPCON2 is used for I2C Master mode and is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GCEN - Set to Enable Interrupt when General Call Address is Received</td>
</tr>
<tr>
<td>6</td>
<td>ACKSTAT - Set when Acknowledge Received from I2C Slave Device</td>
</tr>
<tr>
<td>5</td>
<td>ACKDT - Reset to send Acknowledge at the end of a Byte Receive</td>
</tr>
<tr>
<td>4</td>
<td>ACKEN - Acknowledge I2C Sequence when Set</td>
</tr>
<tr>
<td>3</td>
<td>RCEN - Set to Enable I2C Receive Mode</td>
</tr>
<tr>
<td>2</td>
<td>PEN - Reset to Initiate Stop Condition on I2C Clock and Data</td>
</tr>
<tr>
<td>1</td>
<td>RSEN - Set to Initiate Repeated Start Condition on I2C Clock and Data</td>
</tr>
<tr>
<td>0</td>
<td>SEN - Set to Initiate Start Condition on I2C Clock and Data</td>
</tr>
</tbody>
</table>

The SSPSTAT Register for MSSP is:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SMP - Data Sampled at end of data output time if Set, else middle</td>
</tr>
<tr>
<td>6</td>
<td>CKE - Data transmitted on rising edge of SCK when Set</td>
</tr>
<tr>
<td>5</td>
<td>D/A - Used by I2C. When Set indicates last byte transferred was data. When Reset indicates last byte transferred was address</td>
</tr>
<tr>
<td>4</td>
<td>P - Set when Stop Bit Detected</td>
</tr>
<tr>
<td>3</td>
<td>S - Set when Start Bit Indicated</td>
</tr>
<tr>
<td>2</td>
<td>R/W - Set when command received was a Read</td>
</tr>
<tr>
<td>1</td>
<td>UA - Set when application must update SSPADD Register</td>
</tr>
</tbody>
</table>
| 0   | BF - Set when Buffer is full in RX and when TX...
Interrupts are requested from the SSP via the “SSPIF” bit and enabled by the “SSPIE” bit.

“Non-Return to Zero” ("NRZ") Asynchronous Serial Communications are accomplished by the built-in “USART”. This circuit can also be used for Synchronous Serial Communications. The clock speed is determined by the SPBRG. The TXREG and RCREG registers are used to transfer data. The “RCSTA” is the primary USART control register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SPEN - Set to Enable the USART</td>
</tr>
<tr>
<td>6</td>
<td>RX9 - Set to Enable 9-Bit Serial Reception</td>
</tr>
<tr>
<td>5</td>
<td>ENR - Set to enable single receive for Synchronous Mode</td>
</tr>
<tr>
<td>4</td>
<td>CREN - Set to enable Continuous Receive Mode</td>
</tr>
<tr>
<td>3</td>
<td>ADDEN - Enables Address Detection in Asynchronous Mode</td>
</tr>
<tr>
<td>2</td>
<td>FERR - Framing Error Bit</td>
</tr>
<tr>
<td>1</td>
<td>OERR - Set after Overrun Error</td>
</tr>
<tr>
<td>0</td>
<td>RX9D - Ninth bit of Data Received</td>
</tr>
</tbody>
</table>

and the “TXSTA” is used to control the Serial Output.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CSRC - Set for Synchronous Clock Generated Internally</td>
</tr>
<tr>
<td>6</td>
<td>TX9 - Set to Enable Nine Bit Data Transmission</td>
</tr>
<tr>
<td>5</td>
<td>TXEN - Set to Enable Transmit</td>
</tr>
<tr>
<td>4</td>
<td>SYNC - Set to Select Synchronous Mode</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>BRGH - Set to Select the High Baud Rate</td>
</tr>
<tr>
<td>1</td>
<td>TRMT - Set when Transmit Shift Register is Empty</td>
</tr>
<tr>
<td>0</td>
<td>TX9D - Ninth bit of Transmit Data</td>
</tr>
</tbody>
</table>

The RCIF interrupt request bit, when set, means there is a character received in the USART. RCIF is enabled by
RCIE. TXIF is set when the TX Holding Register is empty and is enabled by TXIE.

Comparator Equipped PICmicro® MCUs have a built-in Reference Voltage Source that is controlled by the VRCON Register:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>VREN - Set to Turn on Voltage Reference Circuit</td>
</tr>
<tr>
<td>6</td>
<td>VROE - Set to Output Voltage Reference Externally</td>
</tr>
<tr>
<td>5</td>
<td>VRR - Set for Low Voltage Reference Range, Reset for High Voltage Reference Range</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
</tr>
<tr>
<td>3-0</td>
<td>VR3:VR0 - Select the Reference Voltage Output</td>
</tr>
</tbody>
</table>

The Voltage Reference Output is defined by the formula:

\[
V_{\text{ref}} = \frac{1}{4}V_{\text{dd}}(1-V_{\text{RR}}) + V_{\text{dd}} \left( \frac{V_{\text{R3:R0}}}{24 + (8 \cdot (1 - V_{\text{RR}}))} \right)
\]

For Vdd equal to 5.0 Volts, the following table lists different Vref values:

<table>
<thead>
<tr>
<th>VR3:VR0</th>
<th>VRR = 1</th>
<th>VRR = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>3.13 Volts</td>
<td>3.59 Volts</td>
</tr>
<tr>
<td>1110</td>
<td>2.92 Volts</td>
<td>3.44 Volts</td>
</tr>
<tr>
<td>1101</td>
<td>2.71 Volts</td>
<td>3.28 Volts</td>
</tr>
<tr>
<td>1100</td>
<td>2.50 Volts</td>
<td>3.13 Volts</td>
</tr>
<tr>
<td>1011</td>
<td>2.29 Volts</td>
<td>2.97 Volts</td>
</tr>
<tr>
<td>1010</td>
<td>2.08 Volts</td>
<td>2.81 Volts</td>
</tr>
<tr>
<td>1001</td>
<td>1.88 Volts</td>
<td>2.66 Volts</td>
</tr>
<tr>
<td>1000</td>
<td>1.67 Volts</td>
<td>2.50 Volts</td>
</tr>
<tr>
<td>0111</td>
<td>1.46 Volts</td>
<td>2.34 Volts</td>
</tr>
<tr>
<td>0110</td>
<td>1.25 Volts</td>
<td>2.19 Volts</td>
</tr>
<tr>
<td>0101</td>
<td>1.04 Volts</td>
<td>2.03 Volts</td>
</tr>
<tr>
<td>0100</td>
<td>0.83 Volts</td>
<td>1.88 Volts</td>
</tr>
<tr>
<td>0011</td>
<td>0.63 Volts</td>
<td>1.72 Volts</td>
</tr>
<tr>
<td>0010</td>
<td>0.42 Volts</td>
<td>1.56 Volts</td>
</tr>
<tr>
<td>0001</td>
<td>0.21 Volts</td>
<td>1.41 Volts</td>
</tr>
<tr>
<td>0000</td>
<td>0.00 Volts</td>
<td>1.25 Volts</td>
</tr>
</tbody>
</table>
166 Chapter 6

The Voltage Reference is normally used with the Voltage Comparator, which is controlled by the “CMCON” Register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>C2OUT - Set when C2Vin+ is Greater than C2Vin-</td>
</tr>
<tr>
<td>6</td>
<td>C1OUT - Set when C1Vin+ is Greater than C1Vin-</td>
</tr>
<tr>
<td>5-4</td>
<td>Unused</td>
</tr>
<tr>
<td>3</td>
<td>CIS - Comparator Input Switch, See CM2:CM0</td>
</tr>
<tr>
<td>2-0</td>
<td>CM2:CM0 - Comparator Mode Select</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>CIS</th>
<th>C1Vin+</th>
<th>C1Vin-</th>
<th>C2Vin+</th>
<th>C2Vin-</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>x</td>
<td>Gnd</td>
<td>Gnd</td>
<td>Gnd</td>
<td>Gnd</td>
<td>Comparators Off</td>
</tr>
<tr>
<td>110</td>
<td>x</td>
<td>AN2</td>
<td>AN0</td>
<td>AN2</td>
<td>AN1</td>
<td>AN3 = C1OUT, RA4 = C2OUT</td>
</tr>
<tr>
<td>101</td>
<td>x</td>
<td>Gnd</td>
<td>Gnd</td>
<td>AN2</td>
<td>AN1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>x</td>
<td>AN3</td>
<td>AN0</td>
<td>AN2</td>
<td>AN1</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>x</td>
<td>AN2</td>
<td>AN0</td>
<td>AN2</td>
<td>AN1</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>Vref</td>
<td>AN3</td>
<td>Vref</td>
<td>AN2</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>Vref</td>
<td>AN0</td>
<td>Vref</td>
<td>AN1</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>AN2</td>
<td>AN3</td>
<td>AN2</td>
<td>AN1</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>AN2</td>
<td>AN0</td>
<td>AN2</td>
<td>AN1</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>x</td>
<td>AN3</td>
<td>AN0</td>
<td>AN2</td>
<td>AN1</td>
<td>Comparators Off</td>
</tr>
</tbody>
</table>

Interrupts Requested by Change on Comparator Outputs are specified CMIF and enabled by CMIE.

There are also some Analog to Digital Converter (“ADC”) Options that can be used with the PICmicro® MCU. The operation of the ADC is controlled by ADCON0 Register:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>ADCS1:ADCS0 - ADC Conversion Clock Select.</td>
</tr>
<tr>
<td>5-3</td>
<td>CHS2:CHS0 - ADC Conversion Channel Select Bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>AN7</td>
</tr>
<tr>
<td>110</td>
<td>AN6</td>
</tr>
<tr>
<td>101</td>
<td>AN5</td>
</tr>
<tr>
<td>100</td>
<td>AN4</td>
</tr>
</tbody>
</table>
Selecting the PORTA, Analog/Digital Functions, there
are a number of different formats of ADCON1 that you
should be aware of. For basic 18-pin PICmicro® MCUs
ADCs, ADCON1 is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-2</td>
<td>Unused</td>
</tr>
<tr>
<td>1-0</td>
<td>PCFG1:PCFG0 - A/D Select</td>
</tr>
<tr>
<td></td>
<td>Bits AN3 AN2 AN1 AN0</td>
</tr>
<tr>
<td>11</td>
<td>D D D D</td>
</tr>
<tr>
<td>10</td>
<td>D D A A</td>
</tr>
<tr>
<td>01</td>
<td>Vref+ A A</td>
</tr>
<tr>
<td>00</td>
<td>A A A A</td>
</tr>
</tbody>
</table>

For more advanced 18-pin PICmicro® MCUs, ADCON1 is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-3</td>
<td>Unused</td>
</tr>
<tr>
<td>2-0</td>
<td>PCFG2:PCFG0 - A/D Select</td>
</tr>
<tr>
<td></td>
<td>Bits AN3 AN2 AN1 AN0</td>
</tr>
<tr>
<td>111</td>
<td>D D D D</td>
</tr>
<tr>
<td>110</td>
<td>D D Vref+ A</td>
</tr>
<tr>
<td>101</td>
<td>D D A A</td>
</tr>
<tr>
<td>100</td>
<td>D D A A</td>
</tr>
<tr>
<td>011</td>
<td>D D Vref+ A</td>
</tr>
<tr>
<td>010</td>
<td>D D A A</td>
</tr>
<tr>
<td>001</td>
<td>Vref+ A</td>
</tr>
<tr>
<td>000</td>
<td>A A A A</td>
</tr>
</tbody>
</table>

28- and 40-pin PICmicro® MCUs have the ADCON1 Register:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-3</td>
<td>Unused</td>
</tr>
</tbody>
</table>
The result of the ADC Operation is stored in ADRES and ADIF is set upon completion of the ADC operation to request an interrupt if ADIE is set.

Ten-bit ADCs are also available in the PICmicro® MCU, with a different ADCON1 Register:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Unused</td>
</tr>
<tr>
<td>5</td>
<td>ADFM - When Set, the Result is “Right Justified” else “Left Justified”</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
</tr>
</tbody>
</table>

In the case of 10-bit ADCs, the result is stored in ADRESL and ADRESH.

This mid-range register list does not include the PIC16C92x's LED Control Registers. This, as well as any
other I/O Hardware registers that were not available when this was written, can be found in the Microchip Datasheets.

**PIC17Cxx**

The PIC17Cxx banking scheme has register addresses 0x010 to 0x017 and 0x020 to 0x0FF being banked and accessed by the BSR register separately. All other register addresses are common regardless of the banks selected within the BSR. The register space of 0x010 to 0x017 consists of the I/O Hardware registers listed according to bank.

Address Range 0x000 to 0x01F is considered to be the “Primary” Register set (“p”) in the PIC17Cxx “move” instructions. For PIC17C4x devices there are four “Primary Banks” (address 0x010 to 0x017); in the PIC17C5x, there are eight “Primary Banks”.

**PICmicro MCU® Register Mappings 169**
<table>
<thead>
<tr>
<th>Addr</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>INDF0</td>
<td>Register Pointed to by FSR0</td>
</tr>
<tr>
<td>0x001</td>
<td>FSR0</td>
<td>Index Register 0</td>
</tr>
<tr>
<td>0x002</td>
<td>PCL</td>
<td>Low Byte of the Program Counter</td>
</tr>
<tr>
<td>0x003</td>
<td>PCLATH</td>
<td>Latched High Byte of the Program Counter</td>
</tr>
<tr>
<td>0x004</td>
<td>ALUSTA</td>
<td>Processor Status and Control Register</td>
</tr>
</tbody>
</table>

**Bit Function**

7-6  FSR1 Mode Select
- 1x - FSR1 Does not Change after Access
- 01 - Post Increment FSR1
- 00 - Post Decrement FSR1

5-4  FSR0 Mode Select
- 1x - FSR0 Does not Change after Access
- 01 - Post Increment FSR0
- 00 - Post Decrement FSR0

3  OV - Set when there is a two’s complement overflow after addition/subtraction
2  Z - Set when the eight bit result is equal to Zero
1  DC - Set for low order Nybble carry after addition or subtraction instruction
0  C - Set for Carry after addition or subtraction instruction

<table>
<thead>
<tr>
<th>Addr</th>
<th>T0STA</th>
<th>TMR0 Status and Control Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x005</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bit Function**

7  INTEDG - Select the RA0/INT Pin
- Interrupt request Edge. When Reset, Rising Edge Increments TMR0 else, Falling Edge Increments TMR0

6  TOSE - TMR0 Clock Input Edge Select - When Set, the Rising edge of the incoming clock increments TMR0 else Falling Edge increments TMR0
5  T0CS - TMR0 Clock Source Select - When Set the Instruction Clock is used, else the T0CKI pin is used
4-1  PS3-PS0 - TMR0 Prescaler Selection.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Prescaler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xxx</td>
<td>256:1</td>
</tr>
<tr>
<td>0111</td>
<td>128:1</td>
</tr>
<tr>
<td>0110</td>
<td>64:1</td>
</tr>
<tr>
<td>0101</td>
<td>32:1</td>
</tr>
<tr>
<td>0100</td>
<td>16:1</td>
</tr>
<tr>
<td>0111</td>
<td>8:1</td>
</tr>
<tr>
<td>0010</td>
<td>4:1</td>
</tr>
<tr>
<td>0001</td>
<td>2:1</td>
</tr>
<tr>
<td>0000</td>
<td>1:1</td>
</tr>
</tbody>
</table>

0  Unused

Any  0x006 CPUSTA     Processor Operating Status Register
Bit Function
7-6  Unused
5  STKAV - When Set, there is Program Counter Stack Space Available
4  GLIMTD - When Set, all Interrupts Are Disabled
3  _TO - Set after Power Up or clrwdt Instruction. When Reset a Watchdog Timeout has occurred
2  _PD - Set after Power Up or clrwdt Instruction. Reset by a "sleep" instruction
1  _POR - Reset After Power Up in PIC17C5x. Not Available in All PIC17Cxx devices
0  _BOR - Reset After Brown Out Reset. Not Available in All PIC17Cxx devices
<table>
<thead>
<tr>
<th>BSR</th>
<th>Addr</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>0x007</td>
<td>INTSTA</td>
<td>Interrupt Status and Control Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit Function</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>PEIE</td>
<td>Set when Peripheral Interrupt is Pending</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>TOCKIF</td>
<td>Set when RA1/T0CKI Pin has Interrupt Source. Cleared by Hardware when Interrupt Vector 0x0018 is executed</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>T0IF</td>
<td>Set when TMR0 Overflows. Cleared by Hardware when Interrupt Vector 0x0010 is executed</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>INTF</td>
<td>Set when RA0/INT Pin Interrupt Request Active. Cleared by Hardware when Interrupt Vector 0x0008 is executed</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>PEIE</td>
<td>Set to Enable Peripheral Interrupt Requests</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>T0CKIE</td>
<td>Set to Enable RA1/T0CKI Interrupt Request</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>T0IE</td>
<td>Set to Enable TMR0 Overflow Interrupt Request</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>INTE</td>
<td>Set to Enable RA0/INT Pin Interrupt Request</td>
</tr>
</tbody>
</table>

| Any | 0x008 | INDF1 | Register Pointed to by PSR1 |
| Any | 0x009 | FSR1  | Index Register 1 |
| Any | 0x00A | WREG  | Processor Accumulator |
| Any | 0x00B | TMR0L | Low Byte of TMR0 |
| Any | 0x00C | TMR0H | High Byte of TMR0 |
| Any | 0x00D | TBLPTRL | Low Byte of the Table Pointer |
| Any | 0x00E | TBLPTREH | High Byte of the Table Pointer |
| Any | 0x00F | BSR   | Bank Select Register |
|     |      |       | Bit Function             |
| 7-4 |      |       | Select General Purpose RAM Bank at Addresses 0x020 to 0x0FF |
| 3-0 |      |       | Select the I/O Hardware Register Bank at Addresses 0x010 to 0x017 |
### PORTA I/O Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>RBPU - When Reset, Pull Up on PORTB is Enabled</td>
</tr>
<tr>
<td>6</td>
<td>Unused</td>
</tr>
<tr>
<td>5</td>
<td>RA5/TX - Input or USART TX Pin. Schmidt Trigger Input</td>
</tr>
<tr>
<td>4</td>
<td>RA4/RX - Input or USART RX Pin. Schmidt Trigger Input</td>
</tr>
<tr>
<td>3</td>
<td>RA3 - Schmidt Trigger Input/Open Drain Output</td>
</tr>
<tr>
<td>2</td>
<td>RA2 - Schmidt Trigger Input/Open Drain Output</td>
</tr>
<tr>
<td>1</td>
<td>RA1/T0CKI - Bit Input or TMR0 Input. Schmidt Trigger Input</td>
</tr>
<tr>
<td>0</td>
<td>RA0/INT - Bit Input or External Interrupt</td>
</tr>
</tbody>
</table>

### PORTB I/O Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>RB7:RB6 - I/O Pin with Interrupt in Input Change. Schmidt Trigger Input</td>
</tr>
<tr>
<td>5</td>
<td>RB5 - I/O Pin with TMR3 Clock Input. Interrupt on Input Change. Schmidt Trigger Input</td>
</tr>
<tr>
<td>4</td>
<td>RB4 - I/O Pin with TMR1/TMR2 Clock Input. Interrupt on Input Change. Schmidt Trigger Input</td>
</tr>
<tr>
<td>3</td>
<td>RB3 - I/O Pin with CCP2 PWM Output. Schmidt Trigger Input</td>
</tr>
<tr>
<td>2</td>
<td>RB2 - I/O Pin with CCP1 PWM Output. Schmidt Trigger Input</td>
</tr>
<tr>
<td>1</td>
<td>RB1 - I/O Pin with CCP2 Capture Input. Schmidt Trigger Input</td>
</tr>
<tr>
<td>0</td>
<td>RB0 - I/O Pin with CCP1 Capture Input. Schmidt Trigger Input</td>
</tr>
<tr>
<td>BSR Addr</td>
<td>Register</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>0x013</td>
<td>RCSTA</td>
</tr>
<tr>
<td>0</td>
<td>Bit Function</td>
</tr>
<tr>
<td>7</td>
<td>SPEN - Set to Enable the USART</td>
</tr>
<tr>
<td>6</td>
<td>RX9 - Set to Enable 9-Bit Serial Reception</td>
</tr>
<tr>
<td>5</td>
<td>SREN - Set to enable single receive for Synchronous Mode</td>
</tr>
<tr>
<td>4</td>
<td>CREN - Set to Enable Continuous Receive Mode</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>FERR - Framing Error Bit</td>
</tr>
<tr>
<td>1</td>
<td>OERR - Set after Overrun Error</td>
</tr>
<tr>
<td>0</td>
<td>RX9D - Ninth bit of Data Received</td>
</tr>
<tr>
<td>0x014</td>
<td>RCREG</td>
</tr>
<tr>
<td>0x015</td>
<td>TXSTA</td>
</tr>
<tr>
<td>0</td>
<td>Bit Function</td>
</tr>
<tr>
<td>7</td>
<td>CSRC - Set for Synchronous Clock Generated Internally</td>
</tr>
<tr>
<td>6</td>
<td>TX9 - Set to Enable Nine Bit Data Transmission</td>
</tr>
<tr>
<td>5</td>
<td>TXEN - Set to Enable Transmit</td>
</tr>
<tr>
<td>4</td>
<td>SYNC - Set to Select Synchronous Mode</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>BRGH - Set to Select the High Baud Rate</td>
</tr>
<tr>
<td>1</td>
<td>TBMT - Set when Transmit Shift Register is Empty</td>
</tr>
<tr>
<td>0</td>
<td>TX9D - Ninth bit of Transmit Data</td>
</tr>
<tr>
<td>0x016</td>
<td>TXREG</td>
</tr>
<tr>
<td>0x017</td>
<td>SPBRG</td>
</tr>
<tr>
<td>1</td>
<td>DDRC</td>
</tr>
<tr>
<td>1</td>
<td>PORTC</td>
</tr>
<tr>
<td>1</td>
<td>DDRD</td>
</tr>
</tbody>
</table>
1 0x013 PORTD PORTD I/O Pins or External Memory Data/Address Pins
1 0x014 DDRE PORTE Data Direction Port. When bit is reset, PORTE bit is in "Output" mode
1 0x015 PORTE PORTE I/O Pins or External Memory Data/Address Pins Control Pins
   Bit  Function
   2   RE2/_WR - I/O Pin or System Bus Write
   1   RE1/_OE - I/O Pin or System Bus Read
   0   RE0/ALE - I/O Pin or System Bus Address Latch Enable
1 0x016 PIR1 Interrupt Status Register 1. This may be the only Interrupt Status Register in Some Devices (in which case it is labeled "PIR")
   Bit  Function
   7   RBIF - Set if PORTB Interrupt on Change Active
   6   TMR3IF - Set if TMR3 has Overflowed or Capture Timer has rolled over
   5   TMR2IF - Set if TMR2 has Overflowed
   4   TMR1IF - Set if TMR1 has Overflowed
   3   CA2IF - Set if Capture2 Event Occurred
   2   CA1IF - Set if Capture1 Event Occurred
   1   TXIF - USART Transmit Interrupt Request
   0   RCIF - USART Receive Interrupt Request
1 0x017 PIE1 Interrupt Control Register 1. This may be the only Interrupt Control Register in Some Devices (in which case it is labeled "PIE")
   Bit  Function
   7   RBIE - Set to Enable PORTB Interrupt on Change
   6   TMR3IE - Set to Enable TMR3 Interrupt
   5   TMR2IE - Set to Enable TMR2 Interrupt
<table>
<thead>
<tr>
<th>RSR</th>
<th>Addr</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0x010</td>
<td>TMR1</td>
<td>TMR1 Data Register</td>
</tr>
<tr>
<td>3</td>
<td>0x011</td>
<td>TMR2</td>
<td>TMR2 Data Register</td>
</tr>
<tr>
<td>2</td>
<td>0x012</td>
<td>TMR3L</td>
<td>Low Byte of the TMR3 Data Register</td>
</tr>
<tr>
<td>1</td>
<td>0x013</td>
<td>TMR3H</td>
<td>High Byte of the TMR3 Data Register</td>
</tr>
<tr>
<td>0</td>
<td>0x014</td>
<td>PW1L</td>
<td>PWM1 Least Significant two Compare Bits</td>
</tr>
<tr>
<td>7</td>
<td>DC1</td>
<td>Bit 1 of the PWM Compare</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>DC0</td>
<td>Bit 0 of the PWM Compare</td>
<td></td>
</tr>
<tr>
<td>5-0</td>
<td>Unused</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x015</td>
<td>PW2L</td>
<td>PWM2 Least Significant two Compare Bits</td>
</tr>
<tr>
<td>7</td>
<td>DC1</td>
<td>Bit 1 of the PWM Compare</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>DC0</td>
<td>Bit 0 of the PWM Compare</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TM2PW2</td>
<td>Set to Select PWM2 Clock Source as TMR2 and PR2 else PWM2 Clock Source is TMR1 and PR1</td>
<td></td>
</tr>
<tr>
<td>4-0</td>
<td>Unused</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x016</td>
<td>PW1H</td>
<td>High Eight Bits of PW1 Compare</td>
</tr>
<tr>
<td>3</td>
<td>0x017</td>
<td>PW2H</td>
<td>High Eight Bits of PW2 Compare</td>
</tr>
<tr>
<td>3</td>
<td>0x018</td>
<td>CA2L</td>
<td>Low Byte of Capture 2 Data</td>
</tr>
<tr>
<td>Address</td>
<td>Register</td>
<td>Description</td>
<td>Bit Function</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>--------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>0x015</td>
<td>CA2H</td>
<td>High Byte of Capture 2 Data</td>
<td></td>
</tr>
<tr>
<td>0x016</td>
<td>TCON1</td>
<td>TMR1 and TMR2 Control Register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
<td></td>
</tr>
<tr>
<td>7-6</td>
<td>CA2ED1:CA2ED0</td>
<td>- Capture 2 Mode Select</td>
<td>11 - Capture on 16th Rising Edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 - Capture on 4th Rising Edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01 - Capture on Every Rising Edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00 - Capture on Every Falling Edge</td>
</tr>
<tr>
<td>5-4</td>
<td>CA1ED1:CA1ED0</td>
<td>- Capture 1 Mode Select</td>
<td>11 - Capture on 16th Rising Edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 - Capture on 4th Rising Edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01 - Capture on Every Rising Edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00 - Capture on Every Falling Edge</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>TMR2:TMR1 Mode Select. When Set, TMR2:TMR1 are a 16 bit Timer else two separate eight bit Timers</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>TMR3CS - When Set TMR3 Increments from Falling Edge of RB3/TCLK3 Pin else Increments from Instruction Clock</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>TMR2CS - When Set TMR2 Increments from Falling Edge of the RB4/TCLK12 Pin else Increments from Instruction Clock</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>TMR1CS - When Set TMR1 Increments from Falling Edge of the RB4/TCLK12 Pin else Increments from Instruction Clock</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x017</td>
<td>TCON2</td>
<td>TMR1 and TMR2 Control Register 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CA2OVF</td>
<td>- Set if Overflow Occurred in Capture2 Register</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CA1OVF</td>
<td>- Set if Overflow Occurred in Capture1 Register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSR</td>
<td>Addr</td>
<td>Register</td>
<td>Function/Bit Definition</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>PWM2ON</td>
<td>Set if PWM2 is Enabled</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>PWM1ON</td>
<td>Set if PWM1 is Enabled</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>CA1_PR3</td>
<td>Set to Enable Capture1 if Enables the Period Register</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>TMR3ON</td>
<td>Set to Enable TMR3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>TMR2ON</td>
<td>Set to Enable TMR2. Must be Set if TMR2/TMR1 are combined</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>TMR1ON</td>
<td>Set to Enable TMR1. When TMR2/TMR1 are combined, controls operation of 16 bit Timer</td>
</tr>
</tbody>
</table>

4 0x010 PIR2 Interrupt Status Register 2

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SSPIF - Set if SSP Interrupt has been Requested</td>
</tr>
<tr>
<td>6</td>
<td>BCLIF - Set if there is a Bus Collision Interrupt Request</td>
</tr>
<tr>
<td>5</td>
<td>ADIF - Set if there is an ADC Interrupt Request</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
</tr>
<tr>
<td>3</td>
<td>CA4IF - Set if Capture4 Event has Requested an Interrupt</td>
</tr>
<tr>
<td>2</td>
<td>CA3IF - Set if Capture4 Event has Requested an Interrupt</td>
</tr>
<tr>
<td>1</td>
<td>TX2IF - Set if USART2 Transmit Interrupt Requested</td>
</tr>
<tr>
<td>0</td>
<td>RC2IF - Set if USART2 Receive Interrupt Requested</td>
</tr>
</tbody>
</table>

4 0x011 PIE2 Interrupt Control Register 2

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SSFIE - Set to Enable SSP Interrupt</td>
</tr>
<tr>
<td>6</td>
<td>BCLIE - Set to Enable Bus Collision Interrupt</td>
</tr>
<tr>
<td>5</td>
<td>ADIE - Set to Enable ADC Interrupts</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
</tr>
</tbody>
</table>
3  CA4IE - Set to Enable Capture4 Interrupt
2  CA3IE - Set to Enable Capture3 Interrupt
1  TX2IE - Set to Enable USART2 Transmit Interrupt
1  RC2IE - Set to Enable USART2 Receive Interrupt

4  0x012 RCSTA2   USART2 Receive Status and Control Register
   Bit Function
   7  SPEN - Set to Enable the USART
   6  RX9  - Set to Enable 9-Bit Serial Reception
   5  SREN - Set to enable single receive for Synchronous Mode
   4  CREN  - Set to Enable Continuous Receive Mode
   3  Unused
   2  FERR  - Framing Error Bit
   1  OERR  - Set after Overrun Error
   0  RX9D - Ninth bit of Data Received

4  0x014 RCREG2   USART2 Receiver Holding Register
4  0x015 TXSTA2   USART2 Transmit Status and Control Register
   Bit Function
   7  CSRC - Set for Synchronous Clock Generated Internally
   6  TX9  - Set to Enable Nine Bit Data Transmission
   5  TXEN - Set to Enable Transmit
   4  SYNC  - Set to Select Synchronous Mode
   3  Unused
   2  BRGH - Set to Select the High Baud Rate
   1  TBMT - Set when Transmit Shift Register is Empty
   0  TX9D - Ninth bit of Transmit Data

4  0x016 TXREG2   USART2 Transmit Holding Register
4  0x017 SPBRG2   USART2 Clock Divisor Register
5  0x010 DDRF   PORTF Data Direction Port. When bit is reset, PORTF bit is in "Output" mode
<table>
<thead>
<tr>
<th>Addr</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x011</td>
<td>PORTF</td>
<td>PORTF I/O Pins or Analog Inputs 4 through 11</td>
</tr>
<tr>
<td>0x012</td>
<td>DDRG</td>
<td>PORTG Data Direction Port. When bit is reset, PORTG bit is in &quot;Output&quot; mode</td>
</tr>
<tr>
<td>0x013</td>
<td>PORTG</td>
<td>PORTG I/O Pins</td>
</tr>
<tr>
<td></td>
<td>Bit Function</td>
<td>QB7/TX2 - Schmidt Trigger I/O or USART2 TX Pin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QB6/RX2 - Schmidt Trigger I/O or USART2 RX Pin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QB5/PWM3 - Schmidt Trigger I/O or PWM3 Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QB4/CAP3 - Schmidt Trigger I/O or Capture3 Pin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QB3/AN0 - Schmidt Trigger I/O or Analog Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QB2/AN1 - Schmidt Trigger I/O or Analog Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QB1/AN2 - Schmidt Trigger I/O or Analog Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QB0/AN3 - Schmidt Trigger I/O or Analog Input</td>
</tr>
<tr>
<td>0x014</td>
<td>ADCON0</td>
<td>ADC Control Register 1</td>
</tr>
<tr>
<td></td>
<td>Bit Function</td>
<td>CHS2:CHS0 - Analog Channel Select</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11xx Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011 AN11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010 AN10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001 AN9</td>
</tr>
</tbody>
</table>
1000 AN8
0111 AN7
0110 AN6
0101 AN5
0100 AN4
0011 AN3
0010 AN2
0001 AN1
0000 AN0
3 Unused
2 GD/_DONE - Set to Start A/D Conversion, 
Reset by Hardware when Finished
1 Unused
0 ADON - Set to Turn on the ADC

5 0x015 ADCON1
ADC Control Register 2
Bit Function
7-6 ADCS1:ADCS0 - ADC Clock Select
11 - Internal RC Clock
10 - Fosc / 64
01 - Fosc / 32
00 - Fosc / 8
5 ADFM - Set for Right Justified Result
Format, else Left Justified Result Format
4 Unused
3-1 PCFG1:PCFG0 - Specify A/D Pins
Bits AN11 AN10 AN9 AN8 AN7 AN6 AN5 AN4 AN3 AN2 AN1 AN0
111 D D D D D D D D D D
110 D D A A D D D D D D A A
BSR Addr Register Function/Bit Definition
101 D A A A D D D D D A A A
100 A A A A D D D D D A A A
011 A A A A D D D D A A A
010 A A A A D D A A A A A
001 A A A A D A A A A A A A
000 A A A A A A A A A A A A
0 PCFG0 - When Set VR+ and VR- Pins are used for Vref+ and Vref- else Vdd and Vss

5 0x016 ADRESL Low Byte of ADC Result
5 0x017 ADRESH High Byte of ADC Result
6 0x010 SSPADD MSSP Address Compare Register
6 0x011 SSPCON1 MSSP Control Register
Bit Function
7 WCOL - Set if SSPBUF was written to while transmitting data or not in correct mode for transmit
6 SSPOV - Set when SSP Receive overflow occurs
5 SSPEN - Enables Pins for SSP Mode
4 CKP - In SPI, Set for Idle Clock High.
    In I2C Mode, set to Enable Clock
3-0 SSPM3:SSPM0 - SSP Mode Select
   1111 - I2C Slave Mode, 10 Bit Address
   1110 - I2C Slave Mode, 7 Bit Address
   110X - Reserved
   1101 - I2C firmware controlled Master
   1100 - Reserved
   1011 - Reserved
   1010 - Reserved
   1001 - Reserved
<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GCEN - Set to Enable Interrupt when General Call Address is Received</td>
</tr>
<tr>
<td>6</td>
<td>ACKSTAT - Set when Acknowledge Received from I2C Slave Device</td>
</tr>
<tr>
<td>5</td>
<td>ACKDT - Reset to send Acknowledge at the end of a Byte Receive</td>
</tr>
<tr>
<td>4</td>
<td>ACKEN - Acknowledge I2C Sequence when Set</td>
</tr>
<tr>
<td>3</td>
<td>RCEN - Set to Enable I2C Receive Mode</td>
</tr>
<tr>
<td>2</td>
<td>PEN - Reset to Initiate Stop Condition on I2C Clock and Data</td>
</tr>
<tr>
<td>1</td>
<td>RSEN - Set to Initiate Repeated Start Condition on I2C Clock and Data</td>
</tr>
<tr>
<td>0</td>
<td>SEN - Set to Initiate Start Condition on I2C Clock and Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SMP - Data Sampled at end of data output time if Set, else middle</td>
</tr>
<tr>
<td>RS Register</td>
<td>Addr</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>0x014</td>
</tr>
<tr>
<td></td>
<td>0x010</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x011</td>
</tr>
<tr>
<td></td>
<td>0x012</td>
</tr>
<tr>
<td></td>
<td>0x013</td>
</tr>
<tr>
<td></td>
<td>0x014</td>
</tr>
<tr>
<td></td>
<td>0x015</td>
</tr>
<tr>
<td>Bit</td>
<td>Function</td>
</tr>
<tr>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>7</td>
<td>Unused</td>
</tr>
<tr>
<td>6</td>
<td>CA4OVF</td>
</tr>
<tr>
<td>5</td>
<td>CA3OVF</td>
</tr>
<tr>
<td>4-3</td>
<td>CA4ED1:CA4ED0</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>2-1</td>
<td>CA3ED1:CA3ED0</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>PWM3ON</td>
</tr>
</tbody>
</table>

Any 0x018 PRODL | Low Byte of Multiplication Product |
Any 0x019 PRODH | High Byte of Multiplication Product |
PIC18Cxx

The hardware registers built into the PIC18Cxx are defined in the following table. Note that these registers are either accessed via the "Access Bank" or using the BSR set to 0x0F.
<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0#80</td>
<td>PORTA</td>
<td>PORTA Read/Write Register. Pin options are defined below:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7  Unused</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6  OSC2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5  Slave Select/Optional AN4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4  Open Drain Output/Schmidt Trigger Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3-0  Optional AN3-AN0</td>
</tr>
<tr>
<td>0x0#81</td>
<td>PORTB</td>
<td>PORTB Read/Write Register. I/O Pins can be pulled by software. Pin options are defined below:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7-6  ICSP Programming Pins/Interrupt on Pin Change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5  Interrupt on Pin Change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4  Interrupt on Pin Change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3  CCP2 I/O and PWM Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2  Interrupt Source 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  Interrupt Source 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0  Interrupt Source 1</td>
</tr>
<tr>
<td>0x0#82</td>
<td>PORTC</td>
<td>PORTC Read/Write Registers. I/O Pins have Schmidt Trigger Inputs. Pin options are defined below:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7  UART Receive Pin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6  UART Transmit Pin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5  Synchronous Serial Port Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4  SPI Data or I2C Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3  SPI Clock or I2C Clock</td>
</tr>
<tr>
<td>Address</td>
<td>Register</td>
<td>Function/Bit Definition</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>0x0#83</td>
<td>PORTD</td>
<td>PORTD Only Available on 40 Pin PIC18Cxx Devices. Schmidt Trigger Inputs. Used for Data Slave Port.</td>
</tr>
<tr>
<td>0x0#84</td>
<td>PORTE</td>
<td>PORTE Only Available on 40 Pin PIC18Cxx. Schmidt Trigger Inputs for I/O Mode. Used for Data Slave Port as Defined below: Bit Function 7-3 Unused 2 Negative Active Chip Select 1 Negative Active Write Enable to PIC18Cxx 0 Negative Active Output Enable (&quot;_RD&quot;) from PIC18Cxx</td>
</tr>
<tr>
<td>0x0#89</td>
<td>LATA</td>
<td>Data Output Latch/Bypassing PORTA</td>
</tr>
<tr>
<td>0x0#8A</td>
<td>LATB</td>
<td>Data Output Latch/Bypassing PORTB</td>
</tr>
<tr>
<td>0x0#8B</td>
<td>LATC</td>
<td>Data Output Latch/Bypassing PORTC</td>
</tr>
<tr>
<td>0x0#8C</td>
<td>LATD</td>
<td>Data Output Latch/Bypassing PORTD. Only available on 40 Pin PIC18Cxx</td>
</tr>
<tr>
<td>0x0#8D</td>
<td>LATE</td>
<td>Data Output Latch/Bypassing PORTE. Only available on 40 Pin PIC18Cxx</td>
</tr>
<tr>
<td>0x0#92</td>
<td>TRISA</td>
<td>I/O Pin Tristate Control Register. Set bit to &quot;0&quot; for output mode</td>
</tr>
<tr>
<td>0x0#93</td>
<td>TRISB</td>
<td>I/O Pin Tristate Control Register. Set bit to &quot;0&quot; for output mode</td>
</tr>
<tr>
<td>0x0#94</td>
<td>TRISC</td>
<td>I/O Pin Tristate Control Register. Set bit to &quot;0&quot; for output mode</td>
</tr>
<tr>
<td>0x0#95</td>
<td>TRISD</td>
<td>I/O Pin Tristate Control Register. Only available on 40 Pin PIC18Cxx. Set bit to &quot;0&quot; for output mode</td>
</tr>
</tbody>
</table>
### TRISE
I/O Pin Tristate Control Register. Only available on 40 Pin PIC18Cxx. Set bit to "0" for output mode. Special function bits specified below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>IBF - Set when PSP is enabled and a byte has been written to the PICmicro® MCU</td>
</tr>
<tr>
<td>6</td>
<td>OBF - Set when PSP is enabled and a byte output has not been read from the PICmicro® MCU</td>
</tr>
<tr>
<td>5</td>
<td>IBOV - Set when PSP is enabled and the byte written to the PICmicro® MCU has been overwritten by a subsequent byte</td>
</tr>
<tr>
<td>4</td>
<td>PSPMODE - Set to Enable PICmicro® MCU's PSP I/O Port</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>TRISE2 - TRIS Bit for RE2</td>
</tr>
<tr>
<td>1</td>
<td>TRISE1 - TRIS Bit for RE1</td>
</tr>
<tr>
<td>0</td>
<td>TRISE0 - TRIS Bit for RE0</td>
</tr>
</tbody>
</table>

### PIE1
Peripheral Interrupt Enable Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>PSPIE - Set to Enable PSP Interrupt Request on Read/Write</td>
</tr>
<tr>
<td>6</td>
<td>ADIE - Set to Enable Interrupt Request on Completion of A/D Operation</td>
</tr>
<tr>
<td>5</td>
<td>RCIE - Set to Enable Interrupt Request on USART Data Receive</td>
</tr>
<tr>
<td>4</td>
<td>TXIE - Set to Enable Interrupt Request on USART Transmit Holding Register Empty</td>
</tr>
<tr>
<td>3</td>
<td>SSPIE - Master Synchronous Serial Port Interrupt Enable Bit</td>
</tr>
<tr>
<td>Address</td>
<td>Register</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>0x0#9E</td>
<td>PIR1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PSPIF</td>
</tr>
<tr>
<td>6</td>
<td>ADIF</td>
</tr>
<tr>
<td>5</td>
<td>RCIF</td>
</tr>
<tr>
<td>4</td>
<td>TXIF</td>
</tr>
<tr>
<td>3</td>
<td>SSPIF</td>
</tr>
<tr>
<td>2</td>
<td>CCP1IF</td>
</tr>
<tr>
<td>1</td>
<td>TMR2IF</td>
</tr>
<tr>
<td>0</td>
<td>TMR1IF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0#9F</td>
<td>IPR1</td>
<td>Peripheral Interrupt Priority Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
</tr>
<tr>
<td>7</td>
<td>PSPIP</td>
<td>Set to Give PSP Interrupt Request on Read/Write Priority</td>
</tr>
<tr>
<td>7</td>
<td>ADIP</td>
<td>Set to Give Interrupt Request on Completion of A/D Operation Priority</td>
</tr>
<tr>
<td>5</td>
<td>RCIP</td>
<td>Set to Give Interrupt Request on USART Data Receive Priority</td>
</tr>
<tr>
<td>4</td>
<td>TXIP</td>
<td>Set to Enable Interrupt Request on USART Transmit Holding Register Empty Priority</td>
</tr>
<tr>
<td>3</td>
<td>SSPIP</td>
<td>Master Synchronous Serial Port Interrupt Priority when Set</td>
</tr>
<tr>
<td>2</td>
<td>CCP1IP</td>
<td>Set to Give CCP1 Interrupt Request Priority</td>
</tr>
<tr>
<td>Address</td>
<td>Register</td>
<td>Function/Bit Definition</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>0x0#AB</td>
<td>RCSTA</td>
<td>USART Receive Status and Control Register</td>
</tr>
<tr>
<td>0x0#AC</td>
<td>TXSTA</td>
<td>USART Transmit Status and Control Register</td>
</tr>
<tr>
<td>0x0#AD</td>
<td>TXREG</td>
<td>USART Transmit Buffer Register</td>
</tr>
</tbody>
</table>

### RCSTA Bit Function
- **Bit 7 (SPEN)**: Set to Enable the USART
- **Bit 6 (RS9)**: Set to Enable 9-Bit Serial Reception
- **Bit 5 (SREN)**: Set to enable single receive for Synchronous Mode
- **Bit 4 (CREN)**: Set to Enable Continuous Receive Mode
- **Bit 3 (ADDEN)**: Enables Address Detection in Asynchronous Mode
- **Bit 2 (FERR)**: Framing Error Bit
- **Bit 1 (ORE)**: Set after Overrun Error
- **Bit 0 (RX9D)**: Ninth bit of Data Received

### TXSTA Bit Function
- **Bit 7 (CSRC)**: Set for Synchronous Clock Generated Internally
- **Bit 6 (TX9)**: Set to Enable Nine Bit Data Transmission
- **Bit 5 (TXEN)**: Set to Enable Transmit
- **Bit 4 (SYNC)**: Set to Select Synchronous Mode
- **Bit 3 (Unused)**
- **Bit 2 (EREN)**: Set to Select the High Baud Rate
- **Bit 1 (TX9T)**: Set when Transmit Shift Register is Empty
- **Bit 0 (TX9D)**: Ninth bit of Transmit Data

### Summary
- **TMR3IF**: Set for TMR3 Overflow Interrupt Request given Priority
- **CCP2IF**: Set for CCP2 interrupt Request given Priority
<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>RD16 - Enable Read/Write of TMR3 as a 16 Bit Operation</td>
</tr>
<tr>
<td>6,3</td>
<td>T3CCP2:T3CCP2 - TMR3 and TMR1 to CCPx Enable Bits</td>
</tr>
<tr>
<td>1</td>
<td>TMR3CS - Set to Select External Clock for TMR3. Reset to Select Instruction Clock</td>
</tr>
<tr>
<td>0</td>
<td>TMR3ON - Set to Enable TMR3</td>
</tr>
<tr>
<td>5-4</td>
<td>T3CKPS1:T3CKPS0 - TMR3 Input Clock Prescaler Control</td>
</tr>
<tr>
<td>2</td>
<td>_T3SYNC - When Reset, TMR3 External Clock is Synchronized</td>
</tr>
<tr>
<td>193</td>
<td>Predko Pocket Chapter 6  9/26/01  12:01 PM  Page 193</td>
</tr>
<tr>
<td>5194</td>
<td>Predko Pocket Chapter 6  9/26/01  12:01 PM  Page 193</td>
</tr>
<tr>
<td>Address</td>
<td>Register</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>3-0</td>
<td>CCP2M3:CCPM0 - CCP2 Mode Select Bits</td>
</tr>
<tr>
<td>11xx</td>
<td>PWM Mode</td>
</tr>
<tr>
<td>1011</td>
<td>Trigger Special Event Compare Mode</td>
</tr>
<tr>
<td>1010</td>
<td>Generate Interrupt on Compare Match</td>
</tr>
<tr>
<td>1001</td>
<td>Initialize CCP2 High and Force</td>
</tr>
<tr>
<td>1000</td>
<td>Low on Compare Match</td>
</tr>
<tr>
<td>0111</td>
<td>Initialize CCP2 High and Force High on Compare Match</td>
</tr>
<tr>
<td>0110</td>
<td>Capture on Every 16th Rising Edge</td>
</tr>
<tr>
<td>0101</td>
<td>Capture on Every 4th Rising Edge</td>
</tr>
<tr>
<td>0100</td>
<td>Capture on Every Falling Edge</td>
</tr>
<tr>
<td>0011</td>
<td>Reserved</td>
</tr>
<tr>
<td>0010</td>
<td>Toggle output on Compare Match</td>
</tr>
<tr>
<td>0001</td>
<td>Reserved</td>
</tr>
<tr>
<td>0000</td>
<td>Capture/Compare/PWM off</td>
</tr>
<tr>
<td>0x0#BB</td>
<td>CCPR2L</td>
</tr>
<tr>
<td>0x0#BC</td>
<td>CCPR2H</td>
</tr>
<tr>
<td>0x0#BD</td>
<td>CCP1CON</td>
</tr>
<tr>
<td>Bit 7-6</td>
<td>Unused</td>
</tr>
<tr>
<td>Bit 5-4</td>
<td>DC1BX1:DC1BX0 - Two Least Significant Bits for the 10 Bit PWM</td>
</tr>
<tr>
<td>3-0</td>
<td>CCP1M3:CCPM0 - CCP1 Mode Select Bits</td>
</tr>
<tr>
<td>11xx</td>
<td>PWM Mode</td>
</tr>
<tr>
<td>1011</td>
<td>Trigger Special Event Compare Mode</td>
</tr>
<tr>
<td>1010</td>
<td>Generate Interrupt on Compare Match</td>
</tr>
<tr>
<td>1001</td>
<td>Initialize CCP1 High and Force Low on Compare Match</td>
</tr>
</tbody>
</table>
1000 - Initialize CCP1 High and Force High on Compare Match
0111 - Capture on Every 16th Rising Edge
0110 - Capture on Every 4th Rising Edge
0101 - Capture on Every Rising Edge
0100 - Capture on Every Falling Edge
0011 - Reserved
0010 - Toggle output on Compare Match
0001 - Reserved
0000 - Capture/Compare/PWM off
0x0#BE  CCPR1L  Least Significant Capture/Compare/PWM1 Register
0x0#BF  CCPR1H  Most Significant Capture/Compare/PWM1 Register
0x0#C1  ADCON1  A/D Control Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>ADFM - Set to Return Result in Right Justified Format, Reset to Return</td>
</tr>
<tr>
<td>6</td>
<td>ADCS2 - Upper Bit of A/D Conversion Clock Select. See &quot;ADCON0&quot; for Bit Definition</td>
</tr>
<tr>
<td>5-4</td>
<td>Unused</td>
</tr>
<tr>
<td>3-0</td>
<td>PCFG3:PCFG0 - A/D Pin Configuration</td>
</tr>
</tbody>
</table>

Select Bits

<table>
<thead>
<tr>
<th>Bits</th>
<th>AN7</th>
<th>AN6</th>
<th>AN5</th>
<th>AN4</th>
<th>AN3</th>
<th>AN2</th>
<th>AN1</th>
<th>AN0</th>
<th>VR+</th>
<th>VR-</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>VR+</td>
<td>VR-</td>
<td>D</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>VR+</td>
<td>VR-</td>
<td>A</td>
<td>A</td>
<td>AN3</td>
<td>AN2</td>
</tr>
<tr>
<td>1100</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>A</td>
<td>VR+</td>
<td>VR-</td>
<td>A</td>
<td>A</td>
<td>AN3</td>
<td>AN2</td>
</tr>
<tr>
<td>1011</td>
<td>D</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td>VR+</td>
<td>VR-</td>
<td>A</td>
<td>A</td>
<td>AN3</td>
<td>Vss</td>
</tr>
<tr>
<td>1010</td>
<td>D</td>
<td>A</td>
<td>A</td>
<td>VR+</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>AN3</td>
<td>Vss</td>
</tr>
<tr>
<td>Address</td>
<td>Register</td>
<td>Function/Bit Definition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C2</td>
<td>ADCON0</td>
<td>A/D Control Register2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bit Function**

- **7-6** ADCS1:ADCS0 - ADC Conversion Clock
  - 111 - Internal RC Oscillator
  - 110 - Divide PICmicro® MCU clock by 64
  - 110 - Divide PICmicro® MCU clock by 32
  - 100 - Divide PICmicro® MCU clock by 16
  - 100 - Divide PICmicro® MCU clock by 8
  - 011 - Internal RC Oscillator
  - 010 - Divide PICmicro® MCU clock by 4

**5-3** CHS2:CHS0 - ADC Conversion Channel

- Select Bits
  - 111 - AN7
  - 110 - AN6
  - 101 - AN5
  - 100 - AN4
  - 011 - AN3
<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GCEN - Set to Enable Interrupt when General Call Address is Received</td>
</tr>
<tr>
<td>6</td>
<td>ACKSTAT - Set when Acknowledge Received from I2C Slave Device</td>
</tr>
<tr>
<td>5</td>
<td>ACKDT - Reset to send Acknowledge at the end of a Byte Receive</td>
</tr>
<tr>
<td>4</td>
<td>ACKEN - Acknowledge I2C Sequence when Set</td>
</tr>
<tr>
<td>3</td>
<td>RCEN - Set to Enable I2C Receive Mode</td>
</tr>
<tr>
<td>2</td>
<td>PEN - Reset to Initiate Stop Condition on I2C Clock and Data</td>
</tr>
<tr>
<td>1</td>
<td>RSEN - Set to Initiate Repeated Start Condition on I2C Clock and Data</td>
</tr>
<tr>
<td>0</td>
<td>SEN - Set to Initiate Start Condition on I2C Clock and Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>WCOL - Set if SSPBUF was written to while transmitting data or not in correct mode for transmit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADRESL</td>
<td>0x0C3</td>
<td>Low Byte of the ADC Result</td>
</tr>
<tr>
<td>ADRESH</td>
<td>0x0C4</td>
<td>High Byte of the ADC Result</td>
</tr>
<tr>
<td>SSPCON2</td>
<td>0x0C5</td>
<td>MSSP Control Register2</td>
</tr>
<tr>
<td>SSPCON1</td>
<td>0x0C6</td>
<td>MSSP Control Register1</td>
</tr>
</tbody>
</table>
### Address | Register | Function/Bit Definition
--- | --- | ---
6 | SSPOV | Set when SSP Receive overflow occurs
5 | SSPEN | Enables Pins for SSP Mode
4 | CKP | In SPI, Set for Idle Clock High. In I2C Mode, set to Enable Clock
3-0 | SSPM3:SSPM0 | SSP Mode Select
1111 | I2C Slave Mode, 10 Bit Address
1110 | I2C Slave Mode, 7 Bit Address
110x | Reserved
1101 | I2C firmware controlled Master
1010 | Reserved
1001 | Reserved
1000 | I2C Master, Fosc/(4*(SSPAD+1))
0111 | I2C Slave Mode, 10 Bit Address
0110 | I2C Slave Mode, 7 Bit Address
0101 | Reserved
0100 | I2C firmware, _SS Disabled
0011 | Reserved
0010 | Reserved
001 | Reserved
0000 | I2C Master, Fosc/4
0001 | I2C Master, Fosc/16
0000 | I2C Master, Fosc/4
0x00C7 | SSPSTAT | MSSP Status Register
Bit | Function
7 | SMP | Data Sampled at end of data output time if Set, else middle
6 | CKE | Data transmitted on rising edge of SCK when Set
5 | D/A | When Set indicates last byte transferred was data, When Reset indicates last byte transferred

---
was address

4  P - Set when Stop Bit Detected
3  S - Set when Start Bit Indicated
2  R/W - Set when command received was a Read
1  UA - Set when application must update SSPADD Register
0  BF - Set when Buffer is full in RX and when TX is in process

0x00C8  SSPADD  MSSP Address Compare Register
0x00C9  SSPBUF  MSSP Data Buffer
0x00CA  T2CON  TMR2 Control Register

Bit  Function
7  Unused
6:3  TOUTPS3:TOUTPS0 - TMR2 Output Postscaler
1111 - 16x
1110 - 15x
1101 - 14x
1100 - 13x
1011 - 12x
1010 - 11x
1001 - 10x
1000 - 9x
0111 - 8x
0110 - 7x
0101 - 6x
0100 - 5x
0011 - 4x
0010 - 3x
0001 - 2x
0000 - 1x
2  TMR2ON - Set to Enable TMR2
<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-0</td>
<td>T2CKPS1:T2CKPS0 - TMR2 Prescaler Select Bits</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>Prescaler is 16</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>Prescaler is 4</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>Prescaler is 1</td>
<td></td>
</tr>
<tr>
<td>0x0#CB</td>
<td>PR2</td>
<td>TMR2 Period Compare Register</td>
</tr>
<tr>
<td>0x0#CC</td>
<td>TMR2</td>
<td>TMR2 Register</td>
</tr>
<tr>
<td>0x0#CD</td>
<td>T1CON</td>
<td>TMR1 Control Register</td>
</tr>
<tr>
<td>7</td>
<td>RD16 - When Set, Enables 16 Bit TMR1 Operations</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>5:4</td>
<td>T1CKPS1:T1CKPS0 - TMR1 Input Clock Prescaler Select</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1:8 Prescaler</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1:4 Prescaler</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1:2 Prescaler</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1:1 Prescaler</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T1OSCEN - Set to Enable TMR1 Oscillator</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>_T1SYNC - Set to Synchronize External Clock Input</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TMR1CS - TMR1 Clock Source Select. Set to Select External Clock</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>TMR1ON - Set to Enable TMR1</td>
<td></td>
</tr>
<tr>
<td>0x0#CE</td>
<td>TMR1L</td>
<td>Low Byte of TMR1</td>
</tr>
<tr>
<td>0x0#CF</td>
<td>TMR1H</td>
<td>High Byte of TMR1</td>
</tr>
<tr>
<td>0x0#D0</td>
<td>RCON</td>
<td>Power UP Status Register</td>
</tr>
<tr>
<td>7</td>
<td>IPEN - Set to Enable Priority Levels on Interrupts</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LWRT - Set to Enable &quot;TBLWT&quot; Instruction to Internal Memory</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>Bit</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>7-1</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SWDTEN - Set to Enable the Watchdog Timer if &quot;_WDT_ON&quot; is specified in &quot;_CONFIG&quot;</td>
<td></td>
</tr>
<tr>
<td>0x0#D1 WDTCON Watchdog Timer Control Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7-6</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>IRVST - Set to indicate Low Voltage Detect Logic will Generate Interrupt</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LVDEN - Set to Enable Low Voltage Detect</td>
<td></td>
</tr>
<tr>
<td>3-0</td>
<td>LVDL3:LVDL0 - Specify the Low Voltage Detect Limits</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>External Voltage Used (LVDIN)</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>4.5V Min - 4.77V Max</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>4.2V Min - 4.45V Max</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>4.0V Min - 4.24V Max</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>3.8V Min - 4.03V Max</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>3.6V Min - 3.82V Max</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>3.5V Min - 3.71V Max</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>3.3V Min - 3.50V Max</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>3.0V Min - 3.18V Max</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>2.8V Min - 2.97V Max</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Register</td>
<td>Function/Bit Definition</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>0101 - 2.7V Min - 2.86V Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 - 2.5V Min - 2.65V Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 - 2.4V Min - 2.54V Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 - 2.2V Min - 2.33V Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 - 2.0V Min - 2.12V Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 - 1.8V Min - 1.91V Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ox0#D3</td>
<td>OSCON</td>
<td>Select PICmicro® MCU Clock Source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
</tr>
<tr>
<td>7-1</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SCS - Reset to use Primary Oscillator. Set to use TMR1’s Oscillator</td>
<td></td>
</tr>
<tr>
<td>Ox0#D5</td>
<td>T0CON</td>
<td>TMR0 Control Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit Function</td>
</tr>
<tr>
<td>7</td>
<td>TMR0ON - Set to Enable TMR0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>T08Bit - Set to Enable TMR0 as an 8 Bit Timer. Reset to Enable TMR0 as a 16 Bit Timer</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TOCS - Set to make TMR0 Clock Source T0CKI pin. Reset to use Instruction Clock</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T0SE - Set to Make TMR0 Increment on Falling Edge of Clock</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>PSA - Reset to Assign TMR0 Prescaler</td>
<td></td>
</tr>
<tr>
<td>2-0</td>
<td>T0PS2:T0PS0 - TMR0 Prescaler Value</td>
<td>111 - 1:256 Prescaler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110 - 1:128 Prescaler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101 - 1:64 Prescaler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 - 1:32 Prescaler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011 - 1:16 Prescaler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010 - 1:8 Prescaler</td>
</tr>
</tbody>
</table>
0x00 - 1:2 Prescaler
0x01 - 1:4 Prescaler

0x0D6 TMR0L Low Byte of TMR0
0x0D7 TMR0H High Byte of TMR0
0x0D8 STATUS PICmicro® MCU Processor Status Register

Bit Function
7-6 Unused
4 N - Set when the Result has bit seven set
3 O - Set when the Result overflows a two's complement number (bit seven changes polarity inadvertently)
2 Z - Set when the Least Significant eight bits of the Result are all Zero
1 DC - Set when the Lower Nybble of the addition/subtraction overflows
0 C - Set in Addition when the result is greater than 0xFF. Reset in Subtraction when the result is negative

0x0D9 FSR2L Low Byte of FSR Register 2
0x0DA FSR2H High Byte of FSR Register 2
0x0DB PLUSW2 INDF2 Consisting of FSR2 + WREG for Address
0x0DC PREINC2 INDF2 With FSR2 Incremented Before Access
0x0DD POSTDEC2 INDF2 With FSR2 Decremented After Access
0x0DE POSTINC2 INDF2 With FSR2 Incremented After Access
0x0DF INDF2 Register Pointed to by FSR2
0x0E0 BSR Bank Select Register - Select Register Bank

Bit Function
7-4 Unused
3-0 BSR3:BSR0, Bank Select Register Bits

0x0E1 FSR1L Low Byte of FSR Register 1
Address | Register | Function/Bit Definition
--- | --- | ---
0x0#E2 | FSR1H | High Byte of FSR Register 1
0x0#E3 | PLUSW1 | INDF1 Consisting of FSR1 + WREG for Address
0x0#E4 | PREINC1 | INDF1 With FSR1 Incremented Before Access
0x0#E5 | POSTDEC1 | INDF1 With FSR1 Decremented After Access
0x0#E6 | POSTINC1 | INDF1 With FSR1 Incremented After Access
0x0#E7 | INDF1 | Register Pointed to by FSR1
0x0#E8 | WREG | PICmicro® MCU Accumulator
0x0#E9 | FSR0L | Low Byte of FSR Register 0
0x0#EA | FSR0H | High Byte of FSR Register 0
0x0#EB | PLUSW0 | INDF0 Consisting of FSR0 + WREG for Address
0x0#EC | PREINC0 | INDF0 With FSR0 Incremented Before Access
0x0#ED | POSTDEC0 | INDF0 With FSR0 Decremented After Access
0x0#EE | POSTINC0 | INDF0 With FSR0 Incremented After Access
0x0#EF | INDF0 | Register Pointed to by FSR0
0x0#F0 | INTCON3 | Interrupt Control Register 3
0x0#F1 | INTCON2 | Interrupt Control Register 2

Bit Function
- 7 INT2IP - INT2 External Interrupt Priority. Set for "High"
- 6 INT1IP - INT1 External Interrupt Priority. Set for "High"
- 5 Unused
- 4 INT2IE - Set to Enable External Int2
- 3 INT1IE - Set to Enable External Int1
- 2 Unused
- 1 INT2IF - Set when External Int2 Requested
- 0 INT1IF - Set when External Int1 Requested
7 _RBPU - Reset to Enable PORTB Pull Ups
6 INTEDG0 - Set for External Int0 on Rising Edge
5 INTEDG1 - Set for External Int1 on Rising Edge
4 INTEDG2 - Set for External Int2 on Rising Edge
3 Unused
2 TMR0IP - High TMR0 Interrupt Request Priority when Set
1 Unused
0 RBIP - High PORTB Change Interrupt Request Priority when Set

0x0#F2 INTCON Interrupt Control Register

Bit Function
7 GIE/GIEH - When Set, Enables all Interrupt Request Sources (unmasked sources when "IPEN" is reset)
6 PEIE/GIEH - Enables all Low Priority Interrupt Request Sources when Set
5 TMR0IE - When Set Enable TMR0 Interrupt Requests
4 INT0IE - When Set Enable INTO Interrupt Requests
3 RBIE - When Set Enable PORTB Change
2 TMO1IF - When Set TMR0 Interrupt Request Active
1 INTOIF - When Set INTO External Interrupt Request Active
0 RBIF - When Set PORTB Change on Interrupt Request Active

0x0#F3 PRODL Low Byte of "Multiply" Instruction Product
0x0#F4 PRODH High Byte of "Multiply" Instruction Product
0x0#F5 TABLAT Table Read and Write Buffer
0x0#F6 TBLPTRL Low Byte of Program Memory Table Pointer
0x0#F7 TBLPTRH Middle Byte of Program Memory Table Pointer
<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Function/Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0#F8</td>
<td>TBLPTRU</td>
<td>High Byte of Program Memory Table Pointer</td>
</tr>
<tr>
<td>0x0#F9</td>
<td>PCL</td>
<td>Low Byte of PICmicro® MCU Program Counter</td>
</tr>
<tr>
<td>0x0#FA</td>
<td>PCLATH</td>
<td>Latched Middle Byte of PICmicro® MCU Program Counter</td>
</tr>
<tr>
<td>0x0#FB</td>
<td>PCKATHU</td>
<td>Latched High Byte of PICmicro® MCU Program Counter</td>
</tr>
<tr>
<td>0x0#FC</td>
<td>STKPTR</td>
<td>Stack Pointer/Index</td>
</tr>
<tr>
<td></td>
<td>Bit</td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>STKFUL - Bit Set when Stack is Full or Overflowed</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>STKUNF - Bit Set when Stack Underflows</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Unused</td>
</tr>
<tr>
<td></td>
<td>4:0</td>
<td>SP4:SP0 - Stack Pointer Location Bits</td>
</tr>
<tr>
<td>0x0#FD</td>
<td>TOSL</td>
<td>Low Byte Access to Top of Program Counter Stack</td>
</tr>
<tr>
<td>0x0#FE</td>
<td>TOSL</td>
<td>Middle Byte Access to Top of Program Counter Stack</td>
</tr>
<tr>
<td>0x0#FF</td>
<td>TOSU</td>
<td>High Byte Access to Top of Program Counter Stack</td>
</tr>
</tbody>
</table>
Chapter 7

Built-In Hardware Features

Configuration Registers

The Configuration Register Fuses are responsible for specifying:

- Oscillators Mode Used
- Program Memory Protection
- reset parameters
- Watchdog Timer
- 16F87x debug mode
The configuration register fuses are unique to each PICmicro® MCU part number. The addresses for the different registers are shown in the table below:

<table>
<thead>
<tr>
<th>Device Family</th>
<th>Configuration Register Address(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-End</td>
<td>0xF0FF</td>
</tr>
<tr>
<td>Mid-Range</td>
<td>0xF207</td>
</tr>
<tr>
<td>PIC17Cxx</td>
<td>0xFE00-0xFE07 Low Byte</td>
</tr>
<tr>
<td></td>
<td>0xFE0F-0xFE08 High Byte</td>
</tr>
<tr>
<td>PIC18Cxx</td>
<td>0x3000000-0x300007</td>
</tr>
</tbody>
</table>

In each PICmicro® MCU's MPLAB device "inc" files, there is a list of parameters for the different options. These parameters are used with the "__CONFIG" statement of an assembler file. I have a few recommendations about this that I will repeat throughout the book. For the PIC18Cxx, there are multiple "__CONFIG#" statements (where "#" is “0” through “7”) and each statement is given a set of bits that can specify different functions.

The "__CONFIG" options are ANDed together to form a word that is programmed into the configuration addresses.

## Oscillators

The basic oscillator options are as follows:

1. Internal Clocking
2. R/C Networks
3. Crystals
4. Ceramic Resonators
5. External Oscillators
Built-in Hardware Features

The “Internal Clocking” option is available in many new PICmicro® MCUs and consists of a capacitor and variable resistor for the oscillator. The “OSCCAL” register shown in Fig. 7.1 is a register that is loaded with a “calibration value”, which is provided by Microchip. This type of oscillator will have an accuracy of 1.5% or better while running at 4 MHz.

The second type of oscillator is the external “RC” oscillator in which a resistor/capacitor network provides the clocking for the PICmicro® MCU as is shown in Fig. 7.2.

The resistor capacitor charging/discharging voltage is buffered through a Schmidt Trigger noninverting buffer, which is used to enable or disable an N-Channel MOSFET transistor pull-down switch. The values for the resistor and capacitor can be found in the Microchip documentation.

Crystals and ceramic resonators use a similar connection scheme for operation. The crystal or ceramic resonator is wired into the circuit as shown in Fig. 7.3. The two capacitors are used to add impedance to the

Figure 7.1 PICmicro® MCU Built-In Oscillator
Chapter 7

Figure 7.2  PICmicro® MCU RC Oscillator

Figure 7.3  PICmicro® MCU Crystal Oscillator
crystal/resonator circuit and their values for reliable operation are specified by Microchip and their ranges are presented elsewhere in this book. As well, for best results, a “parallel circuit” crystal should be used.

There are three speed ranges defined for each device, with the speed specification defining the current output in the PICmicro® MCUs crystal/resonator oscillator circuit.

The speed ranges are

<table>
<thead>
<tr>
<th>Range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>0 – 200 kHz</td>
</tr>
<tr>
<td>XT</td>
<td>200 kHz – 4 MHz</td>
</tr>
<tr>
<td>HS</td>
<td>4 MHz – 20 MHz (or the Device Maximum)</td>
</tr>
</tbody>
</table>

These speed ranges are selected in the “configuration register”.

Using the crystal or ceramic resonator, the OSC2 pin can be used to drive one CMOS input as is shown in Fig. 7.4.

The last type of oscillator is the external oscillator and is driven directly into the OSC1 pin as shown in Fig. 7.5.

The PIC18Cxx has seven different oscillator modes that are available to the application designer. Along with the standard modes described above, there is a PLL clock four time multiplier circuit available, which allows the PICmicro® MCU to run with one instruction cycle per clock cycle. There is also the ability to run from the TMR1 Clock, which can be a slow-speed, power saving clock option for the application.
Figure 7.4  Buffered PICmicro® MCU Crystal Oscillator

Figure 7.5  External PICmicro® MCU Oscillator
1. RC oscillator
2. LP oscillator
3. XT oscillator
4. HS oscillator
5. 4x HS oscillator
6. External oscillator
7. TMR1 clock

The “external oscillator” option will take in an external clock signal and output a one-quarter speed clock on OSC2 unless the OSC2 pin is to be used as “RA6” (like the RC oscillator mode and known as “ECIO”). The external oscillator will work for all data speeds from DC to 40MHz that the 18Cxx can run at.

When the TMR1 oscillator is enabled (by setting the “SCS” bit), execution moves over immediately to the TMR1 clock and the standard oscillator is shut down. This transition is very fast, with only eight TMR1 clock cycles lost before execution resumes with TMR1 as the clock source.

When transitioning from TMR1 to the standard oscillator, the oscillator is restarted with a 1,024 cycle delay for the clock to stabilize before resuming execution. The oscillator circuit in the PIC18Cxx appears in block diagram form as shown in Fig. 7.6.

Sleep
The PICmicro® MCU’s “sleep” function and instruction provides the capability of “shutting down” the PICmicro® MCU by turning off the oscillator and making the PICmicro® MCU wait for reset (“_MCLR” or “WDT”),
an external interrupt, or an externally clocked timer interrupt. Most internal timer interrupt requests are not able to become active because the PICmicro® MCU instruction clock driving the internal hardware clocks is shut off.

Entering "sleep" is accomplished by simply executing the "sleep" instruction. "Sleep" will be terminated by the following events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Execution Resume</th>
</tr>
</thead>
<tbody>
<tr>
<td>_MCLR Reset</td>
<td>Reset Vector</td>
</tr>
<tr>
<td>WDT Reset</td>
<td>Reset Vector</td>
</tr>
<tr>
<td>External Interrupt</td>
<td>Next/Instructions or</td>
</tr>
<tr>
<td>TMR1 Interrupt</td>
<td>Interrupt Vector</td>
</tr>
</tbody>
</table>

Figure 7.6 PIC18Cxx Oscillator Block Diagram
Built-in Hardware Features

The interrupt requests can only wake the device if the appropriate “IE” bits are set. After the “sleep” instruction, the next instruction is always executed, even if the “GIE” bit is set. For this reason it is recommended that a “nop” be always placed after the sleep instruction to ensure no invalid instruction is executed before the interrupt handler:

Two Instruction Sequence used to Initiate “Sleep”

```
sleep
nop
```

The clock restart from “sleep” will be similar to that of a power-on reset, with the clock executing for 1,024 cycles before the “nop” instruction is executed (“Inst(PC + 1)” in the diagram below). This is shown in Fig. 7.7.

---

**Figure 7.7** Sleep Wave Form
216 Chapter 7

Option Register

In the low-end devices, the option register is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Label/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>_GPWU - Enable pull-up wakeup on pin change</td>
</tr>
<tr>
<td>6</td>
<td>_GPPU - Enable I/O PORTB Weak Pull-ups</td>
</tr>
<tr>
<td>5</td>
<td>TOCS - TMRO clock source select</td>
</tr>
<tr>
<td></td>
<td>1 - Tock1 pin</td>
</tr>
<tr>
<td></td>
<td>0 - Instruction clock</td>
</tr>
<tr>
<td>4</td>
<td>TOSE - TMRO Increment Source Edge Select</td>
</tr>
<tr>
<td></td>
<td>1 - High to Low on Tock1 Pin</td>
</tr>
<tr>
<td></td>
<td>0 - Low to High on Tock1 Pin</td>
</tr>
<tr>
<td>3</td>
<td>PSA - Prescaler Assignment Bit</td>
</tr>
<tr>
<td></td>
<td>1 - Prescaler Assigned to Watchdog Timer</td>
</tr>
<tr>
<td></td>
<td>0 - Prescaler Assigned to TMRO</td>
</tr>
<tr>
<td>2-0</td>
<td>PS2-PS0 - Prescaler Rate Select</td>
</tr>
<tr>
<td>000</td>
<td>1:1</td>
</tr>
<tr>
<td>001</td>
<td>1:2</td>
</tr>
<tr>
<td>010</td>
<td>1:4</td>
</tr>
<tr>
<td>011</td>
<td>1:8</td>
</tr>
<tr>
<td>100</td>
<td>1:16</td>
</tr>
<tr>
<td>101</td>
<td>1:32</td>
</tr>
<tr>
<td>110</td>
<td>1:64</td>
</tr>
<tr>
<td>111</td>
<td>1:128</td>
</tr>
</tbody>
</table>

Updating the OPTION register in the low-end is accomplished by the "option" instruction, which moves the contents of "w" into the OPTION_REG (which is the MPLAB label for the option register).

The mid-range devices option register is quite similar, but does not have any device specific bits:
The 17Cxx PICmicro® MCU’s do not have an option register as many of the functions continued by option are either not present (such as the prescaler and PORTB weak pull-ups) or are provided in other registers. The 18Cxx provides a mid-range “compatible” option register, but it is not at the same address as the mid-range devices and cannot be written to using an “option” instruction.

### Built-in Hardware Features

<table>
<thead>
<tr>
<th>Bit</th>
<th>Label/Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>_RBPU - Enable PORTB weak pull-ups</td>
<td>1 - Pull-ups Disabled, 0 - Pull-ups Enabled</td>
</tr>
<tr>
<td>5</td>
<td>INTEDG - Interrupt Request On:</td>
<td>1 - low to high on RBO/INT, 0 - high to low on RBO/INT</td>
</tr>
<tr>
<td>4</td>
<td>TOCS - TMRO clock source select</td>
<td>1 - Tock1 Pin, 0 - Instruction Clock</td>
</tr>
<tr>
<td>3</td>
<td>TOSE - TMRO Update Edge Select</td>
<td>1 - Increment on High to Low, 0 - Increment on Low to High</td>
</tr>
<tr>
<td>2-0</td>
<td>PS2-PS0 - prescaler rate select</td>
<td>000 - 1:1, 001 - 1:2, 010 - 1:4, 011 - 1:8, 100 - 1:16, 101 - 1:32, 110 - 1:64, 111 - 1:128</td>
</tr>
</tbody>
</table>

The block diagram of a “typical” PICmicro® MCU I/O pin is shown in Fig. 7.8. Each register “port” is made up of a
number of these circuits, one for each I/O bit. The convention used for accessing I/O pins is

where “%” is the port letter (port A, port B, etc.) and “#” is the bit number of the port.

The TRIS ("TRI–state buffer enable") register is used to control the output capabilities of the I/O pin. When the register is loaded with a “1” (which is the power-up default), the pin is input only (or in "input mode"), with the tristate buffer disabled and not driving the pin. When a “0” is loaded into a pin’s TRIS bit, the tristate buffer is enabled ("output mode") and the value that is in the “data out” register is driven onto the pin.

The use of the “tris” instruction is not recommended in the mid-range PICmicro® MCU as the instruction can only access PORTA, PORTB, and PORTC. PORTD and PORTE cannot be controlled by the “tris” instruction.

Figure 7.8 Standard PICmicro® MCU I/O Pin Block Diagram
Built-in Hardware Features 219

The recommended way of accessing the mid-range PICmicro® MCU's TRIS registers is to change the “RPO” bit of the STATUS register and read or write the register directly as is shown below:

```
bsf STATUS, RPO
movlw NewTRISA
movwf TRISA ^ 0x080
bcf STATUS, RPO
```

Note, Pin 4 of PORTA ("RA4") in the mid-range PICmicro® MCUs is an “open drain” only output and its design is shown in Fig. 7.9. This pin cannot source a positive voltage out unless it is pulled up.

The weak pull-up on the PORTB pins is enabled by the “_RPBU” bit of the OPTION register and is enabled when this bit is reset and the bit itself is set for out-

![Figure 7.9 PORTA Bit 4 I/O Pin Block Diagram](image-url)
put. The “weak” pull-up is approximately 50 k and can simplify button inputs, eliminating the need for an external pull-up resistor. The port B pin block diagram is shown in Fig. 7.10. When the built-in oscillator is selected by the “_IntRC_OSC” parameter of the “_CONFIG” statement in your source file the pins used for the oscillator are available for IO. When the PICmicro® MCU is programmed, a value for the “calibration register” (“OSCCAL”) has to be inserted. By convention, a

\[
\text{movlw OSCCAL-value}
\]

instruction is put in at the reset address and then at address zero (when the program counter overflows), this value is saved into the OSCCAL register using a

\[
\text{movwf OSCCAL}
\]

instruction.
Watchdog Timer

The Watchdog Timer is an 18 msec (approximately) RC delay, which will reset the PICmicro® MCU if it times out. Normally in an application, it is reset before timing out by executing a "clrwdt" instruction. The block diagram of the WDT is shown in Fig. 7.11.

TMR0

TMR0 is an 8-bit incrementing counter that can be "preset" (loaded) by application code with a specific

```
CONFIG _MCLRE_OFF & _IntRC_OSC ;

; Add Application Specific ; "CP" and "WDT" parameters
org 0
movf OSCCAL
movlw 0xFF ^ (1 << T0CS)
option
; All I/O pins are NOW Available and Internal 4 MHz Clock is Running
; - Start Application
```
value. The counter can either be clocked by an external source or by the instruction clock. Each TMR0 input is matched to two instruction clocks for “synchronization.” This feature limits the maximum speed of the timer to one half the instruction clock speed. The TMR0 block diagram is shown in Fig. 7.12. The “TOCS” and “TOCE” bits are used to select the clock source and the clock edge, which increments TMR0 (rising or falling edge). These bits are located in the “OPTION” register.

TMR0 can be driven by external devices through the “T0CKI” pin. The “T0CKI” pin is dedicated to this function in the low-end devices (although in the 12C5xx and 16C505 PICmicro® MCUs the pin can be used for digital I/O). In the other PICmicro® MCU architectures, the pin can also be used to provide digital I/O. When a clock is driven into the TMR0 input, the input is buffered by an internal “Schmidt Trigger” to help minimize noise-related problems with the input.

Figure 7.12 TMR0 Block Diagram
TMR0 in the mid-range PIC17Cxx and PIC18Cxx can be used to request interrupts when it “overflows” to 0x000 from 0xFF.

Input to TMR0 can be made with and without the “prescaler”, which provides a “divide by” feature to the TMR0 input. For the low-end and mid-range PICmicro® MCU’s, TMR0 is located at register address 0x001. The contents of TMR0 can be read from and written to directly.

Delays (time from which TMR0 is initialized until it overflows) are calculated by using the formula:

\[
\text{TMR0 Initial} = 256 - (\text{Delay Time} \times \text{Clock Frequency} / 8)
\]

**Prescaler**

The “prescaler” is a power-of-two counter that can be selected for use with either the Watchdog Timer or TMR0. Its purpose is to divide the incoming clock signals by a software selectable power-of-two value to allow the 8-bit TMR0 to time longer events or increase the watchdog delay from 18 msecs to 2.3 seconds (Fig. 7.13).

The prescaler’s operation is controlled by the four “PSA” bits in the OPTION register. “PSA” selects whether the watchdog timer uses the prescaler (when PSA is “set”) or TMR uses the prescaler (when PSA is “reset”). Note that the prescaler has to be assigned to either the watchdog timer or TMR0. Both functions are able to execute with no prescaler or with the prescaler’s delay count set to one.
TMR1

TMR1 is 16 bits and can have four different inputs as is shown in Fig. 7.14.

To access TMR1 data, the “TMR1L” and “TMR1H” registers are read and written. If the TMR1 value registers are written, the TMR1 prescaler is reset. A TMR1 inter-
Interrupt request ("TMR1IF") is made when TMR1 overflows and the TMR1IE bit is set.

TMR1IF and TMR1IE are normally located in the "PIR" and "PIE" registers. To request an interrupt, along with TMR1IE and "GIE" being set, the INTCON "PIE" bit must also be set.

To control the operation of TMR1, the T1CON register is accessed with its bits defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Unused</td>
</tr>
<tr>
<td>5-4</td>
<td>T1CPS1:T1CPS0 - Select TMR1 Prescaler Value</td>
</tr>
<tr>
<td>5</td>
<td>11 - 1:8 prescaler</td>
</tr>
<tr>
<td>4</td>
<td>10 - 1:4 prescaler</td>
</tr>
<tr>
<td>3</td>
<td>01 - 1:2 prescaler</td>
</tr>
<tr>
<td>2</td>
<td>00 - 1:1 prescaler</td>
</tr>
<tr>
<td>3</td>
<td>T1SLEN - Set to Enable TMR1's built in Oscillator</td>
</tr>
<tr>
<td>2</td>
<td>T1SYNCH - when TMR1CS reset the TMR1 clock is synchronized to the Instruction Clock</td>
</tr>
<tr>
<td>1</td>
<td>TMR1CS - When Set, External Clock is Used</td>
</tr>
<tr>
<td>0</td>
<td>TMR1ON - When Set, TMR1 is Enabled</td>
</tr>
</tbody>
</table>
The external oscillator is designed for fairly low-speed real-time clock applications. Normally a 32.768 kHz watch crystal is used along with two 33 pF capacitors. 100 kHz or 200 kHz crystals can be used with TMR1, but the capacitance required for the circuit changes to 15 pF. The TMR1 oscillator circuit is shown in Fig. 7.15.

In the PIC18Cxx devices, TMR1 can be specified as the processor clock to allow low-speed, low-power application execution without putting the PICmicro® MCU to "sleep".

The TMR1 prescaler allows 24-bit instruction cycle delay values to be used with TMR1. These delays can either be a constant value or an “overflow”, similar to TMR0. To calculate a delay, use the formula:

\[
\text{Delay} = \frac{(65,536 - \text{TMR1Init}) \times \text{prescaler}}{\text{T1frequency}}
\]

where the “T1frequency” can be the instruction clock, TMR1 oscillator or an external clock driving TMR1.

\[
\text{TMR1Init} = 65,536 - \frac{(\text{Delay} \times \text{T1Frequency})}{\text{prescaler}}
\]
TMR2

TMR2 (Fig. 7.16) is used as a recurring event timer. When it is used with the CCP module, it is used to provide a Pulse Width Modulated timebase frequency. In normal operations, it can be used to create a 16-bit instruction cycle delay.

TMR2 is continually compared against the value in “PR2”. When the contents of TMR2 and PR2 match, TMR2 is reset, the event is passed to the CCP as “TMR2 Reset”. If the TMR2 is to be used to produce a delay within the application, a postscaler is incremented when TMR2 overflows and eventually passes an interrupt request to the processor.

TMR2 is controlled by the T2CON register, which is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Unused</td>
</tr>
<tr>
<td>6–5</td>
<td>TOUTPS3:TOUTPS0 - TMR2 Postscaler Select</td>
</tr>
<tr>
<td></td>
<td>1111 - 16:1 Postscaler</td>
</tr>
<tr>
<td></td>
<td>1110 - 15:1 Postscaler</td>
</tr>
<tr>
<td></td>
<td>1100 - 14:1 Postscaler</td>
</tr>
<tr>
<td></td>
<td>1000 - 13:1 Postscaler</td>
</tr>
</tbody>
</table>

**Figure 7.16** Timer2 (“TMR2”) Block Diagram
The “TMR2” register can be read or written at any time with the caution that writes may cause the prescaler to be zeroed. Updates to TMR2 do not reset the TMR2 prescaler. The timer itself is not synchronized with the instruction clock because it can only be used with the instruction clock.

PR2 contains the reset, or count up to, value. The delay before reset is defined as:

\[
\text{Delay} = \frac{\text{prescaler}}{256} \times \left( \frac{PR2 + 1}{\text{Fosc} / 4} \right)
\]

If “PR2” is equal to zero, the delay is:

\[
\text{Delay} = \frac{\text{prescaler} \times 256}{\text{Fosc} / 4}
\]

Interrupts use the “TMR2IE” and “TMR2IF” bits that are similar to the corresponding bits in TMR1. These bits are located in the “PIR” and “PIE” registers. Because of the exact interrupt frequency, TMR2 is well suited for applications that provide “bit banging” functions like asynchronous serial communications or Pulse Width Modulated signal outputs.

**Compare/Capture/PWM (CCP) Module**

Included with TMR1 and TMR2 is a control register and a set of logic functions (known as the “CCP”), which enhances the operation of the timers and can simplify applications. This hardware may be provided singly or in
pairs, which allows multiple functions to execute at the same time. If there are two CCP modules built into the PICmicro® MCU, then one is known as “CCP1” and the other as “CCP2”. In the case where there are two CCP modules built-in, then all the registers are identified with the “CCP1” or “CCP2” prefix.

The CCP hardware is controlled by the “CCP1CON” (or “CCP2CON”) register, which is defined as:

<table>
<thead>
<tr>
<th>CCPxCON Bit Definition</th>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7-6</td>
<td>Unused</td>
</tr>
<tr>
<td></td>
<td>5-4</td>
<td>DC1B1 : DC1B0 - CEPST significant 2 bits of the PWM compare value.</td>
</tr>
<tr>
<td></td>
<td>3-0</td>
<td>CCP1M3 : CCP1M0 - CCP module operating mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11xx - PWM Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011 - Compare Mode - Trigger Special Event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010 - Compare Mode - Generate Software Interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001 - Compare Mode - on Match CCP pin low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000 - Compare Mode - on Match CCP pin high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111 - Capture on every 16th rising edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110 - Capture on every 4th rising edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101 - Capture on every rising edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100 - Capture on every falling edge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00xx - CCP off</td>
</tr>
</tbody>
</table>

“Capture mode” loads the CCPR registers (“CCPR14”, “CCPR1C”, “CCPR2H”, and “CCPR2L”) according to the mode the CCP register is set in. This function is shown in Fig. 7.17 and shows that the current TMR1 value is saved when the specified compare condition is met.

Before enabling the capture mode, TMR1 must be enabled (usually running with the PICmicro® MCU clock). The “edge detect” circuit in Fig. 7.17 is a four-to-one multiplexor, which chooses between the prescaled ris-
ing edge input or a falling edge input and passes the selected edge to latch the current TMR1 value and optionally request an interrupt.

In capture mode, TMR1 is running continuously and is loaded when the condition on the CCPx pin matches the condition specified by the CCPxMS:CCPxM0 bits. When a capture occurs, then an interrupt request is made. This interrupt request should be acknowledged and the contents of CCPRxH and CCPRxL saved to avoid having them written over and the value in them lost.

“Compare” mode changes the state of the CCPx pin of the PICmicro® MCU when the contents of TMR1 match the value in the CCPRxH and CCPRxL registers as shown in Fig. 7.18. This mode is used to trigger or control external hardware after a specific delay.

“PWM” CCP mode outputs a PWM signal using the TMR2 reset at a specific value capability. The block diagram of PWM mode is shown in Fig. 7.19. The mode is a combination of the normal execution of TMR2 and capture mode; the standard TMR2 provides the PWM period while the compare control provides the “on” time specification.
When the PWM circuit executes, TMR2 counts until its most significant 8 bits are equal to the contents of PR2. When TMR2 equals PR2, TMR2 is reset to zero and the CCPx pin is set “high”. TMR2 is run in a 10-bit mode.
(the 4:1 prescaler is enabled before PWM operation). This 10-bit value is then compared to a program value in CCPRxM (along with the two DCxBx bits in CCPxCON) and when they match, the CCPx output pin is reset low.

```
<table>
<thead>
<tr>
<th>Code to Setup up a 65% Duty Cycle PWM</th>
</tr>
</thead>
<tbody>
<tr>
<td>movlw 199</td>
</tr>
<tr>
<td>movwf PR2                              ; Set up TMR2 Operation</td>
</tr>
<tr>
<td>movlw (1 &lt;&lt; TMR2 on) +1</td>
</tr>
<tr>
<td>movwf T2CON                            ; Start it Running with a 50 msec Period</td>
</tr>
<tr>
<td>movlw 130                              ; 65% of the Period</td>
</tr>
<tr>
<td>movwf CCPRxH</td>
</tr>
<tr>
<td>movlw (1 &lt;&lt; DCxB1) +0x00F</td>
</tr>
<tr>
<td>movwf CCPxCON                          ; Start PWM</td>
</tr>
<tr>
<td>; PWM is operating</td>
</tr>
</tbody>
</table>
```

The table below gives the fractional DCxBX bit values:

```
<table>
<thead>
<tr>
<th>Fraction</th>
<th>DCxB1:DCxB0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>00</td>
</tr>
<tr>
<td>0.25</td>
<td>01</td>
</tr>
<tr>
<td>0.50</td>
<td>10</td>
</tr>
<tr>
<td>0.75</td>
<td>11</td>
</tr>
</tbody>
</table>
```

**USART Module**

There are three modules to the USART, the clock generator, the serial data transmission unit and the serial data reception unit. The two serial I/O units require the clock generator for shifting data out at the write interval. The clock generator’s block diagram is Fig. 7.20.
In the clock generator circuit, the “SPBRG” register is used as a comparison value for the counter. When the counter is equal to the “SPBRG” register’s value, a clock “tick” output is made and the counter is reset. The counter operation is gated and controlled by the “SPEN” (“serial port enable”) bit along with the “synch” (which selects whether the port is in synchronous or asynchronous mode) and “BRGH” which selects the data rate.

For asynchronous operation, the data speed is specified by the formula:

\[
\text{Data Rate} = \frac{\text{Fosc}}{(16 \times 4^{2 \times (1 - \text{BRGH})}) \times (\text{SPBRG} + 1)}
\]

This formula can be rearranged so that the SPBRG value can be derived from the desired data rate:

\[
\text{SPBRG} = \frac{\text{Fosc}}{(\text{Data Rate} \times 16 \times 4^{2 \times (1 - \text{BRGH})})} - 1
\]

The transmission unit of the USART can send 8 or 9 bits in a clocked (synchronous) or unclocked (asynchronous) manner. Data transmission is initiated by sending...
a byte to the “TXREG” register. The block diagram of the hardware is shown in Fig. 7.21.

The transmit hold register can be loaded with a new value to be sent immediately following the passing of the byte in the “Transmit shift register”. This single buffering of the data allows data to be sent continuously without the software polling the TXREG to find out when is the correct time to send out another byte. USART transmit interrupt requests are made when the TX holding register is empty. This feature is available for both synchronous and asynchronous transmission modes.

The USART receive unit is the most complex of the USART’s three parts. This complexity comes from the need for it to determine whether or not the incoming asynchronous data is valid or not using the “Pin Buffer and Control” unit built into the USART receive pin. The block diagram for the USART’s receiver is shown in Fig. 7.22.

If the port is in synchronous mode, data is shifted in
either according to the USART’s clock or using an external device’s clock.

Like the TX unit, the RX unit has a “holding register”, so if data is not immediately processed, and an incoming byte is received, the data will not be lost. But, if the data is not picked up by the time the next byte has been received, then an “overrun” error will occur. Another type of error is the “framing error”, which is set if the “stop” bit of the incoming NRZ packet is not zero. These errors are recorded in the “RCSTA” (receiver status) register and have to be reset by software.

For asynchronous data, the “Receiver Sensor Clock” is used to provide a polling clock for the incoming data. This sixteen time data rate clock’s input into the “Pin Buffer and Control” unit provides a polling clock for the hardware. When the input data line is low for three Receive Sensor Clock periods, data is then read in from the “middle” of the next bit as is shown in Fig. 7.23. When data is being received, the line is polled three
times and the majority of states read is determined to be the correct data value. This repeats for the 8 or 9 bits of data with the stop bit being the final check.

In some PICmicro® MCUs, the USART receive unit can also be used to receive two asynchronous bytes in the format “Data:Address”, where “Address” is a byte destined for a specific device on a bus. When the “AD-DEN” bit of the “RCSTA” register is set, no interrupts will be requested until both the address and data bytes have been received. To distinguish between the bytes, the ninth address bit is set (while the ninth bit of data packets are reset). When this interrupt request is received, the interrupt handler checks the device address for its value before responding to the data byte.

To control the USART, two registers are used explicitly. The “TXSTA” (“transmitter status”) register is located at address 0x098 in the mid-range PICmicro® MCUs and has the bit definitions:
The “SPBRG” register is usually at address 0x099 for the mid-range PICmicro® MCUs.

The “RCSTA” (receiver status) register is at address 0x018 in the mid-range PICmicro® MCUs and is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CSRC - Clock Source Select used in Synchronous Mode. When Set, the USART Clock Generator is Used</td>
</tr>
<tr>
<td>6</td>
<td>TX9 - Set to Enable nine bit Serial I/O</td>
</tr>
<tr>
<td>5</td>
<td>TXEN - Set to Enable Data Transmission</td>
</tr>
<tr>
<td>4</td>
<td>SYNC - Set to Enable Synchronous Transmission</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>BRGH - Used in Asynchronous Mode to Enable Fast Data Transmission. It is Recommended to keep this bit Reset</td>
</tr>
<tr>
<td>1</td>
<td>TRMT - Set if the Transmission Shift Register is Empty</td>
</tr>
<tr>
<td>0</td>
<td>RX9D - Received ninth bit</td>
</tr>
</tbody>
</table>

**USART TXSTA Bit Definition**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CSRC - Clock Source Select used in</td>
</tr>
<tr>
<td></td>
<td>Synchronous Mode. When Set, the USART Clock</td>
</tr>
<tr>
<td></td>
<td>Generator is Used</td>
</tr>
<tr>
<td>6</td>
<td>TX9 - Set to Enable nine bit USART Receive</td>
</tr>
<tr>
<td>5</td>
<td>TXEN - Set to Enable Data Transmission</td>
</tr>
<tr>
<td>4</td>
<td>SYNC - Set to Enable Synchronous Transmission</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>BRGH - Used in Asynchronous Mode to Enable</td>
</tr>
<tr>
<td></td>
<td>Fast Data Transmission. It is Recommended to</td>
</tr>
<tr>
<td></td>
<td>keep this bit Reset</td>
</tr>
<tr>
<td>1</td>
<td>TRMT - Set if the Transmission Shift Register</td>
</tr>
<tr>
<td></td>
<td>is Empty</td>
</tr>
<tr>
<td>0</td>
<td>TXD - Nine bit of Transmitted Data</td>
</tr>
</tbody>
</table>

The “USART TXSTA” register is used to control the transmission and reception of data. The “SYNC” bit is used to enable synchronous transmission, while the “BRGH” bit is used in asynchronous mode to enable fast data transmission. The “CSRC” bit is used to select the clock source for synchronous mode, and the “TX9” bit is used to enable nine-bit serial I/O.

The “USART TXSTA” register is defined as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CSRC - Clock Source Select used in Synchronous Mode. When Set, the USART Clock Generator is Used</td>
</tr>
<tr>
<td>6</td>
<td>TX9 - Set to Enable nine bit USART Receive</td>
</tr>
<tr>
<td>5</td>
<td>TXEN - Set to Enable Data Transmission</td>
</tr>
<tr>
<td>4</td>
<td>SYNC - Set to Enable Synchronous Transmission</td>
</tr>
<tr>
<td>3</td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td>BRGH - Used in Asynchronous Mode to Enable Fast Data Transmission. It is Recommended to keep this bit Reset</td>
</tr>
<tr>
<td>1</td>
<td>TRMT - Set if the Transmission Shift Register is Empty</td>
</tr>
<tr>
<td>0</td>
<td>TXD - Nine bit of Transmitted Data</td>
</tr>
</tbody>
</table>

**USART RCSTA Bit Definition**

The “USART RCSTA” register is used to control the reception of data. The “ADDEN” bit is used to receive data:address information. It may be unused in many PICmicro® MCU part numbers.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SPEN - Set to Enable the USART</td>
</tr>
<tr>
<td>6</td>
<td>RX9 - Set to Enable nine bit USART Receive</td>
</tr>
<tr>
<td>5</td>
<td>RXEN - Set to Enable Single Byte Synchronous</td>
</tr>
<tr>
<td></td>
<td>Data Receive. Reset when data has been received</td>
</tr>
<tr>
<td>4</td>
<td>CREN - Set to Enable Continuous Receive</td>
</tr>
<tr>
<td>3</td>
<td>ADDEN - Set to Receive Data:Address Information. May be unused in many PICmicro® MCU Part Numbers</td>
</tr>
<tr>
<td>2</td>
<td>FEERR - “Framing Error” bit</td>
</tr>
<tr>
<td>1</td>
<td>OERR - “Overrun Error” bit</td>
</tr>
<tr>
<td>0</td>
<td>RX9D - Received ninth bit</td>
</tr>
</tbody>
</table>
The TXREG is normally at address 0x019 and RCREG is normally at address 0x01A for the mid-range PICmicro® MCUs. The TXIF, TXIE, RCIE, and RCIF bits are in different interrupt enable request registers and bit numbers are specific to the part being used.

To set up asynchronous serial communication transmit, the following code is used:

Code to set up USART Asynchronous Serial Transmission

```
bsf STATUS, RPO
bcf TXSTA, SYNCH ; Not in Synchronous mode
bcf TXSTA, BRGH ; BRGH = 0
movlw DataRate ; Set USART Data Rate
movwf SPBRG
bcf STATUS, RPO ; Enable serial port
bsf RCSTA ^ 0x080, SPEN
bsf STATUS, RPO
bcf TXSTA, TX9 ; Only 8 bits to send
bsf TXSTA, TXEN ; Enable Data Transmit
bcf STATUS, RPO
```

To send a byte in "w", use the code:

```
btfs TXSTA, TMT
goto $ - 1 ; Wait for Holding Register
; to become Free/Empty
movwf TXREG ; Load Holding Register
; If Transmit Shift Register
; is Empty, byte will be sent
```
Built-in Hardware Features

To set up an asynchronous read, the following code is used:

```assembly
bsf STATUS, RPO          ; Want Asynch Communications
bcf TXSTA, SYNCH         ; Low Speed Clock
movlw DataRate           ; Set Data Rate
movwf SPBRG
bsf RCSTA ^ 0x080, SPEN  ; Enable Serial Port
bcf TCSTA ^ 0x080, RX9   ; Eight Bits to Receive
```

To receive data, use the code:

```assembly
btfss PIR1, RXIF          ; Wait for a Character to be Received
goto $ - 1
movf RCREG, w            ; Get the Byte Received
bcf RXIF                  ; Reset the RX byte Interrupt
```

**SSP Module**

The Synchronous Serial Protocol Module is used to send and receive data serially using a synchronous (with a clock) protocol like the data stream shown in Fig. 7.24.

**SPI operation**

SPI is an 8-bit synchronous serial protocol that uses three data bits to interface to external devices. Data is clocked out, with the most significant bit first, on rising or falling edges of the clock. The clock itself is generated.
within the PICmicro® MCU ("master mode"), or it is provided by an external device and used by the PICmicro® MCU ("slave mode") to clock out the data. The SPI data stream looks like Fig. 7.24.

The clock can be "positive" as shown in Fig. 7.24 with a "0" "idle" or negative (high "line idle") with a "1" idle and the clock pulsing to "0" and back again. The Data receive latch is generally on the return to idle state transition.

The "BSSP" module is the "Basic SSP" module and provides data pulling on the return to idle clock edge. The original SSP module provides the ability to vary when data is output and read. Controlling the operation of the different SSP modules is the "SSPCON" register.

**Figure 7.24** SPI Synchronous Serial Data Waveform

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>WCOL: Write collision, set when new byte written to SSPBUF while transfer is taking place</td>
</tr>
<tr>
<td>6</td>
<td>SSPOV: Receive Overflow, indicates that the unread byte is SSPBUF overwritten while in SPI slave mode</td>
</tr>
<tr>
<td>5</td>
<td>SSPEN: Set to enable the SSP module</td>
</tr>
<tr>
<td>4</td>
<td>CKP: Clock polarity select, set to have a high idle</td>
</tr>
</tbody>
</table>
The block diagram for the SSP module is shown in Fig. 7.25. In master mode, when a byte is written to SSPBUF, an 8-bit, most-significant-bit first data transfer process is initiated. The status of the transfer can be checked by the SSPSTAT register “BF” flag; the SSPSTAT register is defined as:

The block diagram for the SSP module is shown in Fig. 7.25.
The SSP SPI transfers can be used for single byte synchronous serial transmits of receivers with serial devices. Figure 7.26 shows the circuit to transmit a byte to

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SMP - Set to have data sampled after active to idle transition, reset to sample at active to idle transition, not available in BSSP</td>
</tr>
<tr>
<td>6</td>
<td>CKE - Set to TX data on idle to active transition, else TX data on active to idle transition, not available in BSSP</td>
</tr>
<tr>
<td>5</td>
<td>D_/A - Used by I2C</td>
</tr>
<tr>
<td>4</td>
<td>P - Used by I2C</td>
</tr>
<tr>
<td>3</td>
<td>S - Used by I2C</td>
</tr>
<tr>
<td>2</td>
<td>R_/W - Used by I2C</td>
</tr>
<tr>
<td>1</td>
<td>UA - Used by I2C</td>
</tr>
<tr>
<td>0</td>
<td>BF - Busy flag, reset while SPI operation active</td>
</tr>
</tbody>
</table>

Figure 7.26  SSP SPI Module Used to Shift Data Out
a 74LS374 wired as a serial in/parallel out shift register. Figure 7.27 shows a 74LS374 being used with a 74LS244 as a synchronous parallel in/serial out register. Both of these operations are initiated by a write to SSPBUF.

```
bsf IOPin ; Want to Latch Data into the '374
bcf SCK
bsf STATUS, RPO
bcf IOPin
bcf SCK
bsf STATUS, RPO
bsf SCK ; Latch the Data into the '374
bcf SCK
bcf IOPin ; Disable '244 output, Enable '374
movlw (I << SMP) + (I << CKE)
movwf SSPSTAT ; Set up the SSP Shift In
movlw (I << SSPEN) + (I << CKP) +0x000
```

Figure 7.27 SSP SPI Module Used to Shift Data In
When using the SSP, the data rate can either be selected as a multiple of the executing clock or use the TMR2 overflow output. The actual timing is dependent on the hardware the PICmicro® MCU SSP master is communicating with.

When in slave mode, along with an external clock being provided, there is a transmit reset pin known as "SS". When this pin is asserted high, the SSP output is stopped (the SDO TRIS bit is changed to input mode) and the SSP is reset with a count of zero. When the bit is reset, the clock will start up again, the original most significant bit is reset, followed by the remaining 7 bits.

I2C operation

The enhanced MSSP will be designed into all new devices that have the SSP module. In this section, the single master I2C interface is focused on.

There are five registers that are accessed for MSSP I2C operation, they are the SSP control registers ("SSPCON" and "SSPCON2"), the SSP status register ("SSPSTAT"),
the SSP receive/transmit register (“SSPBUF”) and the SSP address register (“SSPADD”). These registers are available in the SSP and BSSP, but are slightly different for the MSSP.

The MSSP control registers are defined as:

**MSSP SSPCON Bit Definition**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>WCOL - Write collision, set when new byte written to SSPBUF while transfer is taking place</td>
</tr>
<tr>
<td>6</td>
<td>SSPOV - Receive Overflow, indicates that the unread byte is SSPBUF overwritten</td>
</tr>
<tr>
<td>5</td>
<td>SSPEN - Set to enable the SSP module</td>
</tr>
<tr>
<td>4</td>
<td>CKP - In I2C Modes, if bit is reset, the I2C &quot;SCL&quot; Clock Line is Low. Keep this bit set.</td>
</tr>
<tr>
<td>3-0</td>
<td>SSPM3:SSPM0 SPI mode select</td>
</tr>
<tr>
<td>1111</td>
<td>I2C 10 Bit Master Mode/Start and Stop Bit Interrupts</td>
</tr>
<tr>
<td>1110</td>
<td>I2C 7 Bit Master Mode/Start and Stop Bit Interrupts</td>
</tr>
<tr>
<td>1101</td>
<td>Reserved</td>
</tr>
<tr>
<td>1100</td>
<td>Reserved</td>
</tr>
<tr>
<td>1011</td>
<td>I2C Master Mode with Slave Idle</td>
</tr>
<tr>
<td>1010</td>
<td>Reserved</td>
</tr>
<tr>
<td>1001</td>
<td>Reserved</td>
</tr>
<tr>
<td>1000</td>
<td>I2C Master Mode with SSPADD Clock Definition</td>
</tr>
<tr>
<td>0111</td>
<td>I2C Slave Mode, 10 Bit Address</td>
</tr>
<tr>
<td>0110</td>
<td>I2C Slave Mode, 7 Bit Address</td>
</tr>
<tr>
<td>0101</td>
<td>SPI slave mode, clock = SCK pin, _SS not used</td>
</tr>
<tr>
<td>0100</td>
<td>SPI slave mode, clock = SCK pin, _SS enabled</td>
</tr>
<tr>
<td>0011</td>
<td>SPI master mode, TMR2 clock used</td>
</tr>
<tr>
<td>0010</td>
<td>SPI master mode, INCK/16</td>
</tr>
<tr>
<td>0001</td>
<td>SPI master mode, INCK/4</td>
</tr>
<tr>
<td>0000</td>
<td>SPI master mode, INCK</td>
</tr>
</tbody>
</table>
The status of the transfer can be checked by the SSPSTAT register “BF” flag; the SSPSTAT register is defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SMP - Set to have data sampled after active to idle transition, reset to sample at active to idle transition, not available in BSSP</td>
</tr>
<tr>
<td>6</td>
<td>CKE - Set to TX data on idle to active transition, else TX data on active to idle transition, not available in BSSP</td>
</tr>
<tr>
<td>5</td>
<td>D/A - Used by I2C</td>
</tr>
<tr>
<td>4</td>
<td>P - Used by I2C</td>
</tr>
<tr>
<td>3</td>
<td>S - Used by I2C</td>
</tr>
<tr>
<td>2</td>
<td>R/W - Used by I2C</td>
</tr>
<tr>
<td>1</td>
<td>UA - Used by I2C</td>
</tr>
<tr>
<td>0</td>
<td>BF - Busy flag, reset while SPI operation active</td>
</tr>
</tbody>
</table>
I2C connections between the PICmicro® MCU’s I2C “SDA” (data) and “SCL” (clock) pins is very simple with just a Pull Up on each line as shown in Fig. 7.28. 1K resistors are recommended for 400 kHz data transfers and a 10K for 100 kHz data rates. Note that before any of the I2C modes are to be used, the “TRIS” bits of the respective “SDA” and “SCL” pins must be in input mode. Unlike many of the other built-in advanced I/O functions, MSSP does not control the TRIS bits. Not having the TRIS bits in input mode will not allow the I2C functions to operate.

In “Master Mode”, the PICmicro® MCU is responsible for driving the clock (“SCL”) line for the I2C network. This is done by selecting one of the SPI Master Modes and loading the SSPADD register with a value to provide a data rate that is defined by the formula:

\[
\text{I2C Data Rate} = \frac{\text{Fosc}}{4 \times (\text{SSPADD} + 1)}
\]

This can be rearranged to:

\[
\text{SSPADD} = \left(\frac{\text{Fosc}}{4 \times \text{I2C Data Rate}}\right) - 1
\]
To send data from the PICmicro® MCU to an I2C device using the MSSP, the following steps must be taken:

1. The SDA/SCL lines MUST be put into “Input Mode” (i.e., their respective “TRIS” bits must be set).
2. I2C Master Mode is enabled. This is accomplished by setting the “SSPEN” bit of SSPCON and writing 0b01000 to the SSPM3:SSPM0 bits of the SSPCON register.
3. A “Start Condition” is initiated by setting the “SEN” bit of SSPCON2. This bit is then polled until it is reset.
4. SSPBUF is loaded with the address of the device to access. Note that for many I2C devices, the least significant bit transmitted is the “Read/Write” bit. The “R/_W” bit of SSPSTAT is polled until it is reset (which indicates the transmit has been completed).
5. The ACK bit from the receiving device is checked by reading the “ACKDT” bit of the SSPCON2 register.
6. SSPBUF is loaded with the first 8 bits of data or a secondary address that is within the device being accessed. The “R/_W” bit of SSPSTAT is polled until it is reset.
7. The ACK bit from the receiving device is checked by reading the “ACKDT” bit of the SSPCON2 register.
8. A new “Start Condition” may have to be initiated between the first and subsequent data bytes. This is initiated by setting the “SEN” bit of SSPCON2. This bit is then polled until it is reset.
9. Operations six through eight are repeated until all data is sent or a “NACK” (negative Acknowledge) is received from the receiving device.
10. A “Stop Condition” is initiated by setting the “PEN” bit of SSPCON2. This bit is then polled until it is reset.

This sequence of operations is shown in Fig. 7.29. Note that in Fig. 7.29, the “SSPIF” interrupt request flag operation is shown. In the sequence above, I avoid interrupts, but the “SSPIF” bit can be used to either request an interrupt or to avoid the need to poll different bits to wait for the various operations to complete.

To receive data from a device requires a similar set of operations with the only difference being that after the address byte(s) have been sent, the MSSP is configured to receive data when the transfer is initiated:

1. The SDA/SCL lines MUST be put into “Input Mode” (i.e., their respective “TRIS” bits must be set).
2. I2C Master Mode is enabled. This is accomplished by setting the “SSPEN” bit of SSPCON and writing 0b01000 to the SSPM3:SSPM0 bits of the SSPCON register.

3. A “Start Condition” is initiated by setting the “SEN” bit of SSPCON2. This bit is then polled until it is reset.

4. SSPBUF is loaded with the address of the device to access. Note that for many I2C devices, the least significant bit transmitted is the “Read/Write” bit. The “R/W” bit of SSPSTAT is polled until it is reset (which indicates the transmit has been completed).

5. The ACK bit from the receiving device is checked by reading the “ACKDT” bit of the SSPCON2 register.

6. SSPBUF is optionally loaded with the secondary address within the device being read from. The “R/W” bit of SSPSTAT is polled until it is reset.

7. If a secondary address was written to the device being read from, reading the “ACKDT” bit of the SSPCON2 register checks the ACK bit from the receiving device.

8. A new “Start Condition” may have to be initiated between the first and subsequent data bytes. This is initiated by setting the “SEN” bit of SSPCON2. This bit is then polled until it is reset.

9. If the secondary address byte was sent, then a second device address byte (with the “Read” indicated) may have to be sent to the device being read. The “R/W” bit of SSPSTAT is polled until it is reset.

10. The “ACKDT” will be set (“NACK”) or reset (“ACK”) to indicate whether or not the data byte transfer is to be acknowledged in the device being read.
11. The “RCEN” bit in the SSPCON2 register is set to start a data byte receive. The “BF” bit of the SSPSTAT register is polled until the data byte has been received.

12. Operations ten through eleven are repeated until all data is received and a “NACK” (negative Acknowledge) is sent to the device being read.

13. A “Stop Condition” is initiated by setting the “PEN” bit of SSPCON2. This bit is then polled until it is reset.

Fig. 7.30 shows the data receive operation waveform.

Along with the single “Master” mode, the MSSP is also capable of driving data in “Multi-Master” mode. In this mode, if a data write “collision” is detected, it stops transmitting data and requests an interrupt to indicate there is a problem. An I2C “collision” is the case where
the current device is transmitting a “High” data value but there is a “Low” data value on the SDA line. This condition is shown in Fig. 7.31. The “WCOL” bit of the SSPCON register indicates that the collision has taken place.

When the collision occurs, the I2C software must wait some period of time before polling the SDA and SCL lines to ensure that they are high and then initiating a “Repeated Start Condition” operation. A “Repeated Start Condition” is the process of restarting the I2C data transfer right from the beginning (even if it was halfway through when the collision occurred).

**Built-In ADC**

All PICmicro® MCU devices that have a “seven” as the second to last character in the part number have a built-in analog to digital converter, which will indicate an analog voltage level from zero to Vdd, with 8- or 10-bit accuracy. The PORTA pins can be used as either digital I/O or analog inputs. The actual bit accuracy, utilization of pins and operating speed is a function of the
PICmicro® MCU part number and the clock speed the PICmicro® MCU runs at.

When a pin is configured for analog input, it follows the models shown in Fig. 7.32.

“Rs” in the “Vs source” circuit is the in-line resistance of the power supply. In order to get reasonable times for charging the ADC’s “holding capacitor”, this value should be less than 10K.

The time required for the holding capacitor to load the analog voltage and to stabilize is

\[
T_{\text{acc}} = 5\text{ms} + [(\text{temp} - 25\degree C) \times 0.05 \text{ ms/C}] \\
+ (3.19 \times 10^{-7}) \times (8k + Rs)
\]

which works out to anywhere from 7.6 usecs to 10.7 usecs at room temperature. For most applications, this calculation can be ignored and a “stabilization” time of 15 usecs can be used as a rule of thumb.

Once the voltage is stabilized at the capacitor, a test for each bit is made. 9.5 cycles are required to do an 8-bit conversion. The bit conversion cycle time (known as “TAD”) can be anywhere from 1.6 to 6.4 usecs and can

![Figure 7.32 PICmicro® MCU Internal ADC Equivalent Input](image-url)
either use the PICmicro® MCU's instruction clock or a built-in 250 kHz RC oscillator. To get a valid TAD time using the PICmicro® MCU's instruction clock, a two, eight, or thirty-two prescaler is built into the ADC.

A built-in 250 kHz oscillator is used to carry out the ADC conversion when the PICmicro® MCU is asleep or to avoid using the prescaler. For maximum ADC accuracy, Microchip recommends that the PICmicro® MCU be put to sleep during the ADC conversion for maximum accuracy (and minimum internal voltage or current upsets). If the PICmicro® MCU is put to sleep, then the minimum conversion time is much longer than what is possible using the built-in clock because the PICmicro® MCU has to restart when the ADC completion interrupt has been received.

The minimum conversion time is defined as the total time required for the holding capacitor to stabilize at the input voltage and for the ADC operation to complete.

To measure analog voltages, the analog input pins or the PICmicro® MCU, which are in “PORTA”, have to be set to analog input on power up, the analog input pins are normally set to analog input and not digital I/O. To specify the modes, the “ADCON1” register is written to.

In the following table, the two least significant bits (known as PCFG1:PCFG0) of the “ADCON1” register is shown with the types of I/O pin operation selected in a PIC16C71:

<table>
<thead>
<tr>
<th>ADCON1 bits</th>
<th>AN3</th>
<th>AN2</th>
<th>AN1</th>
<th>AN0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>D</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>01</td>
<td>Vref</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>00</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

Sample ADCON1 Bit Definitions for the PIC16C71
The “ADCON 0” register is used to control the operation of the ADC. The bits of the register are typically defined as:

### ADCON0 Bit Definitions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>ADCS1: ADCS0 bits used to select the TAD clock.</td>
</tr>
<tr>
<td></td>
<td>11 - Internal 250 kHz Oscillator</td>
</tr>
<tr>
<td></td>
<td>10 - POSC/32</td>
</tr>
<tr>
<td></td>
<td>01 - POSC/8</td>
</tr>
<tr>
<td></td>
<td>00 - POSC/2</td>
</tr>
<tr>
<td>5-3</td>
<td>CHS2:CHS0 - Bits used to Select which Analog Input is to be Measured.</td>
</tr>
<tr>
<td></td>
<td>These bits and their operation is Part Number Specific</td>
</tr>
<tr>
<td>2</td>
<td>GO/DONE - Set Bit to Start ADC Conversion.</td>
</tr>
<tr>
<td></td>
<td>Reset by Hardware when ADC Conversion is Complete.</td>
</tr>
<tr>
<td>1</td>
<td>ADIF - Set upon Completion of ADC Conversion and Requests an Interrupt.</td>
</tr>
<tr>
<td>0</td>
<td>ADON - Set to Enable the ADC</td>
</tr>
</tbody>
</table>

The ADC consumes power even when it is not being used and for this reason, if the ADC is not being used “ADON” should be reset.

If the PICmicro® MCU’s ADC is capable of returning a 10-bit result, the data is stored in the two “ADRESH” and “ADRES” registers. When 10-bit ADC results are available, the data can be stored in ADRESH/ADRESL in two different formats. The first is to store the data “right justified” with the most significant six bits of ADRESH loaded with “zero” and the least two significant bits loaded with the two most significant bits of the result. This format is useful if the result is going to be used as a 16-bit number, with all the bits used to calculate an average.
The second 10-bit ADC result format is “left justified” in which the eight most significant bits are stored in “ADRESH”. This format is used when only an 8-bit value is required in the application and the two least significant bits can be “lopped” off or ignored.

To do an analog to digital conversion, the following steps are taken:

1. Write to ADCON1 indicating what are the digital I/O pins and which are the analog I/O pins. At this time, if a 10-bit conversion is going to be done, set the format flag in ADCON1 appropriately.
2. Write to ADCON0, setting ADON, resetting ADIF and GO/DONE and specifying the ADC TAD clock and the pin to be used.
3. Wait for the input signal to stabilize.
4. Set the GO/DONE bit. If this is a high-accuracy measurement, ADIE should be enabled for interrupts and then the PICmicro® MCU put to “sleep”.
5. Poll “GO/DONE” until it is reset (conversion done).
6. Read the result from “ADRES” and optionally “ADRESH”.

To read an analog voltage from the RAO pin of a PIC167C1 running a 4-MHz PICmicro® MCU, the code would be:

```
bsf STATUS, RPO
movlw 0x002
movwf ADCON1 ; AN1/AN0 are Analog Inputs
bcf STATUS, RPO
movlw 0x041 ; Start up the ADC
movwf ADCON0
movlw 5
addlw 0x0FF ; Delay 20 usec for Holding
```
Built-in Hardware Features

```
btfss STATUS, Z ; Capacitor to Stabilize
    goto $ - 2
    bsf ADCON0, GO ; start the ADC conversion
    btfsc ADCON0, GO ; Wait for the ADC Conversion
    goto $ - 1 ; to End
    movf ADRES, w ; Read the ADC result
```

**Built-In Comparators**

In the PIC16C2x, analog voltages can be processed by the use of comparators that indicate when a voltage is greater than another voltage. The inputs “compared” can be switched between different I/O pins as well as ground or a reference voltage that can be generated inside the PICmicro® MCU chip.

Enabling comparators is a very straightforward operation with the only prerequisite being that the pins used for the analog compare must be in “input” mode. Comparator response is virtually instantaneous, which allows “alarm” or other fast responses from changes in the comparator inputs (Fig. 7.33).

There are two comparators in the PIC16C62X controlled by the “CMCON” register, which is defined as:

<table>
<thead>
<tr>
<th>CMCON Bit Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5-4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2-0</td>
</tr>
</tbody>
</table>
The CIS and CM2:CM0 bits work together to select the operation of the comparators.

<table>
<thead>
<tr>
<th>CM</th>
<th>CIS</th>
<th>Comp 1 Input</th>
<th>Comp 2 Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>X</td>
<td>RA0 (1)</td>
<td>RA2 (2)</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>RA0 (1)</td>
<td>RA2 (2)</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>RA3 (3)</td>
<td>RA2 (2)</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>Vref (4)</td>
<td>RA3 (2)</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>Vref (4)</td>
<td>RA3 (2)</td>
</tr>
<tr>
<td>011</td>
<td>X</td>
<td>RA2 (5)</td>
<td>RA3 (2)</td>
</tr>
<tr>
<td>100</td>
<td>X</td>
<td>RA1 (4)</td>
<td>RA2 (2)</td>
</tr>
<tr>
<td>101</td>
<td>X</td>
<td>DON'T CARE</td>
<td>RA2 (2)</td>
</tr>
<tr>
<td>110</td>
<td>X</td>
<td>RA0 (5)</td>
<td>RA2 (2)</td>
</tr>
<tr>
<td>111</td>
<td>X</td>
<td>RA3 (7)</td>
<td>RA2 (2)</td>
</tr>
</tbody>
</table>
Built-in Hardware Features

From these selections, there are some notes.

1. For CM2:CM0 equal to 000, RA3 through RA0 cannot be used for digital I/O.
2. For CM2:CM0 equal to 000, RA2 and RA1 cannot be used for digital I/O.
3. RA3 can be used for digital I/O.
4. RA0 and RA3 can be used for digital I/O.
5. RA3 is a digital output, same as comparator 1 output.
6. RA4 is the open drain output of comparator 2.
7. RA0 and RA3 can be used for digital I/O.
8. RA1 and RA2 can be used for digital I/O.

Upon power up, the comparator CM bits are all reset, which means RA0 to RA3 are in analog input mode. If you want to disable analog input, the CM bits must be set (write 0x007 to CMCOM).

Interrupts can be enabled that will interrupt the processor when one of the comparator’s output changes. This is enabled differently for each PICmicro® MCU with built-in comparators. Like the PORTB change on interrupt, after a comparator change interrupt request has been received, the CMCOM register must be read to reset the interrupt handler.

Along with comparing to external values, the PIC16C62x can also generate a reference voltage (“Vref” in the table above) using its own built-in 4-bit digital-to-analog converter. The digital-to-analog converter circuit is shown in Fig. 7.34.

The Vref control bits are found in the VRCON register and are defined as:
The Vref output is dependent on the state of the “VRR” bit. The Vref voltage output can be expressed mathematically if VRR is set as:

\[ V_{\text{ref}} = V_{\text{dd}} \times (\text{VRCON} \& 0x00F)/24 \]
Or, if it is reset as:

\[ V_{\text{ref}} = V_{\text{dd}} \times \frac{8 + (\text{VRCON} \& 0x00F))}{32} \]

Note that when VRR is set, the maximum voltage of \( V_{\text{ref}} \) is 15/24 of \( V_{\text{dd}} \), or just less than two-thirds \( V_{\text{dd}} \). When VRR is reset, \( V_{\text{ref}} \) can be almost three-quarters of \( V_{\text{dd}} \).

Parallel Slave Port

The PSP is very easy to wire up with separate chip select and read/write pins for enabling the data transfer. The block diagram of the PSP is shown in Fig. 7.35.

A read and write operation waveform is shown in Fig. 7.36.

The minimum access time is one clock (not "instruc-
tion clock") cycle. For a PICmicro® MCU running at 20 MHz, the minimum access time is 50 nsecs.

To enable the parallel slave port, the “PSP mode” bit of the TRISE register must be set. When this bit is set, port D becomes driven from the "_CS", "_RD", and "_WR" bits, which are RE2, RE1, and RE0, respectively. When the PSP mode bit is set, the values in PORTD, PORTE, TRISD, and TRISE are ignored.

When PSP mode is enabled and _CS and _RD are active, PORTD drives out the contents of "OUTREG". When "OUTREG" (which is at PORTD's address) is written to, the "OBF" ("Output Buffer Full") bit of TRISE is set. This feature, along with the input data flags in TRISE is not available in all devices. The PBF bit will become reset automatically, when the byte in the OUTREG is read by the device driving the external parallel bus.

When a byte is written into the parallel slave port (_CS and _WR are active), the value is saved in "INREG" until it is overwritten by a new value. If the optional status registers are available, the “IBF” bit is set when the

---

**Figure 7.36** Parallel Slave Port Operation
INREG is written to and cleared when the byte in INREG is read. If the byte is not read before the next byte is written into “INREG”, the “IBOV” bit, which indicates the overwrite condition is set.

In older PICmicro® MCUs that have PSP port, the “IBF”, “OBF”, and “IBOV” bits are not available in TRISE.

Built-In EEPROM Data Memory Access

The “EECON1”, “EECON2”, “EEADR”, and “EEDATA” are used to control access to the EEPROM. “EEADR” and “EEDATA” are used to provide the address and data interface into the up to 256 byte data EEPROM memory. “EECON” and “EECON2” are used to initiate the type of access as well as indicate that the operation has completed. “EECON2” is a “pseudo-register” that cannot be read from, but is written to with the data, 0x055/0x0AA to indicate the write is valid.

EECON1, contains the following bits for controlling the access:

<table>
<thead>
<tr>
<th>Critical EECON1 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
</tr>
<tr>
<td>EEPCD</td>
</tr>
<tr>
<td>WRERR</td>
</tr>
<tr>
<td>WREN</td>
</tr>
<tr>
<td>WR</td>
</tr>
<tr>
<td>RD</td>
</tr>
</tbody>
</table>
Using these bits, a Read can be initiated as:

```assembly
movf / movlw address/ADDR, w
bcf STATUS, RPO
movwf EEADR
bsf STATUS, RPO
bsf EECON1, ^ 0x08, RD
bcf STATUS, RPO
movf EEDATA, w ; w = EEPROM [address/ADDR]
```

Write operations are similar, but have two important differences. The first is that the operation can take up to ten milliseconds to complete, which means the “WR” bit of EECON1 has to be polled for completion, or in the EEPROM, interrupt request hardware enabled. The second difference as mentioned above, is that a “timed write” has to be implemented to carry out the operation.

```assembly
movlw / movf constant/DATA, w
bcf STATUS, RPO
movwf EEDATA
movlw / movf address/ADDR, w
movwf EEADR
bsf STATUS, RPO
bsf EECON1, ^ 0x080, WREN
bcf INTCON, GIE
movlw 0x055              ] CRITICAL SECTION
movwf EECON2 ^ 0x080     ]
movlw 0x0AA              ]
movwf EECON2 ^ 0x080     ]
bsf EECON1 ^ 0x080, WR ]
bsf INTCON, GIE
btfsc EECON1 ^ 0x080, WR ] Poll for Operation Ended
 goto $ - 1 ]
bcf EECON1 ^ 0x080, WHEN
bcf STATUS, RPO
bsf INTCON, GIE
```
The EEPROM included PIC12CE5xx parts use the most significant bits of the “GPIO” ("general purpose I/O") register and its corresponding “TRIS” register. The PIC12CE5xx's EEPROM interface can be described as shown in the block diagram Fig. 7.37.

In Fig. 7.37, the GPIO bits six and seven do not have "TRIS" control bits. As well, bit six (the 12CEEPROM bit), "SDA") has an open-drain driver. This driver circuit is designed to let both the PICmicro® MCU and the EEPROM drive the data line at different intervals without having to disable the PICmicro® MCUs write of the EEPROM. Information is written to the EEPROM device using the waveform shown in Fig. 7.38.

The “start” and “stop” bits are used to indicate the beginning and end of an operation and can be used halfway through to halt an operation. The start and stop bits are actually invalid cases (data cannot change while one clock is active or “high”).

**Figure 7.37** PIC12CE5xx EEPROM Interface
This operation means that the GPIO port must be accessed carefully; always make sure the SDA and SCL GPIO bits have a “one” in them or else the built-in EEPROM may be accessed incorrectly, causing problems with subsequent reads.

The instruction

```assembly
clr GPIO
```

should never be used in applications that access low-end data EEPROM.

Data is written most significant bit first, which is probably backwards to most applications. Before any transfer, a “control byte” has to be written. The “control byte’s” data is in the format:

```binary
0b01010000R
```

where “R” is the “Read/Write” byte (indicating what is coming next). If the “read/write” bit is set, then a read of
the EEPROM at the current address pointer will take place. If a write is to take place, the “read/write” bit is reset.

After a byte is sent, the SDA line is pulled low to indicate an “acknowledgment” (“ACK” or just “A” in the bit stream representations below). This bit is set low (as an acknowledgment) when the operation has completed successfully. If the acknowledgment bit is high (“NACK”), it does not necessarily mean there was a failure; if it is issued by the EEPROM then it indicates a previous write has not completed. The PICmicro® MCU will issue it to stop the EEPROM from preparing to send additional bytes out of its memory in a multi-byte read.

There are five operations that can be carried out with the EEPROM that is built into the PIC12CE5xx. They are

1. Current Address Set.
2. Current Address Set/Data Byte Write.
3. Data Byte Read at Current Address.
4. Sequential (“multi-byte”) Read at Current Address.
5. Write completion poll.

The EEPROM in the PIC12CE5xx is only 16 bytes in size. Each byte is accessed using a 4-bit address. This address is set using a control byte with the “R” bit reset followed by the address. The bit stream looks like:

idle - Start - 1010000A - 0000addrA - DataByteA - Stop - idle
In the second byte sent, the 0b00000addr pattern indicates that the four “addr” address bits become the address to set the EEPROM’s internal address pointer to for subsequent operations. After the two bytes have been sent, the SCL and SDA lines are returned to “IDLE” for three cycles, using the instruction:

```assembly
movlw 0x0C0
iorwf GPIO, f       ; set SDA /SCL
```

before another operation can complete.

The address data write is similar to the address write, but does not force the two lines into IDLE mode and it passes along a data byte before stopping the transfer:

```
Idle – Start – 10100000A – 0000addrA – DataByteA
- Stop – idle
```

Data bytes can be read singly or sequentially depending on the state of “ACK” from the PICmicro® MCU to the EEPROM after reading a byte. To halt a read, when the last byte to be read has been received, the PICmicro® MCU issues a “NACK” (or “N” in the bitstream listing) to indicate that the operation has completed.

A single byte read looks like:

```
idle – Start – 10100001A – DataByteN – Stop – idle
```

while a 2-byte read looks like:

```
idle – Start – 10100001A – DataByteA – DataByteN
- Stop – idle
```

The last operation is sending dummy “write” control bytes to poll the EEPROM to see whether or not a byte
write has completed (10 msecs are required). If the write has completed, then an “ACK” will be returned else a “NACK” will be returned.

**EPROM Program Memory Access**

To read from the EPROM (or external memory), the following code can be used:

```assembly
movf   SaveAddress + 1, TBLPTRH   ; Setup TBLPTR to the Data
movf   SaveAddress, TBLPTRL      ; being Read
tablrd 0, 0, SaveData            ; Load TBLAT with Memory
tlrd   1, SaveData + 1           ; Contents
       tlrd   0, SaveData
movf   SaveData + 1, WREG        ; High Instruction Byte
movf   SaveData, WREG            ; Low Instruction Byte

; 
; 
```

To write to the built-in EPROM of the PIC17Cxx, the "_MCLR" line will have to be driven to Vpp (13 to 14 volts). When the program memory is being written, all instruction execution in the PIC17Cxx stops. To resume operation after a program memory write, an interrupt, like returning from a TMR0 interrupt request, is executed. Sample code for writing to the PIC17Cxx’s program memory is as follows:

```assembly
org      0x00010
TMR0Int                              ; Timer Interrupt Request
```
270 Chapter 7

; Acknowledge
retfie

: ; Point to the Memory being written to
movfp SaveAddress, TBLPTDL ; Turn on Programming Voltage
movfp SaveAddress + 1, TBLPTRH ; Delay 100 msecs for Programming Voltage to Stabilize

bcf PORTA, 3
movlw HIGH ((100000 / 5) + 256) ; Delay 100 msecs for Programming Voltage to Stabilize
movwf Dlay

movlw LOW ((100000 / 5) + 256)
addlw 0x0FF
btfsc ALUSTA, Z
decfsz Dlay, f
goto $ - 3
movlw HIGH (65536 - 10000) ; Delay 10 msecs for EPROM Write
movwf TMR0H
movlw LOW (65536 - 10000)
movwf TMR0L
bsf TOSTA, TOCS ; Start up the Timer
movlw 1 << TOIE ; Enable Interrupts
movwf INTSTA
bcf CPFSTA, GLINTD
tlw 0, SaveData ; Load Table Pointer with Data
tlw 1, SaveData + 1
tablw 1, 0, SaveData + 1 ; Write the Data In
Flash Program Memory Access

To read to program memory, the following code is used for the 16F87x. Note the two "nops" to allow the operation to complete before the instruction is available for reading:

```
btf  STATUS, RP1
movlw LOW address/ADDR, w
movwf EERADR ^ 0x0100
movlw HIGH address/ADDR, w
movwf EERADH ^ 0x0100
btf  STATUS, RPO
btf  EECON1 ^ 0x0180, EEPGD
btf  EECON1 ^ 0x0180, RD
nfi
nfi
bfi  STATUS, RPO
mvuf EEDATA, w
mvuf ------ ; Store Lo
        ; Byte of
        ; Program
        ; Memory
mvuf EEDATAH, w
mvuf ------ ; Store Hi
        ; Byte of
        ; Program
        ; Memory
btf  STATUS, RP1
```
Writing to program memory is similar to writing to data, but also has the two nops in which the operation takes place. There are no polling or interrupts available for this operation, instead, the processor halts during this operation. Even though the processor has stopped for a program memory write, peripheral function (ADC's, serial I/O, etc.) are still active.

```assembly
bsf STATUS, RP1
movlw /movf LOW address/ADDR, w
movwf EEADR
movlw /movf HIGH address/ADDR, w
movwf EEADRH
movlw /movf LOW Constant/DATA, w
movwf EEDATA
movlw /movf HIGH Constant/DATA, w ; Maximum 0x03F
movwf EEDATAH
bsf STATUS, RPO
bsf EECON1 ^ 0x0180, EEPO
bsf EECON1 ^ 0x0180, WREN
bcf INTCON, GIE          ; Critically timed
movlw 0x055               ; code.
movwf EECON2 ^ 0x0180     ; executes
movwf EECON2 ^ 0x0180, OR
nop                    ; operation
nop
bcf EECON1 ^ 0x0180, WREN
bsf INTCON, GIE
```

**External Parallel Memory**

Parallel memory devices can be connected to the 17Cxx PICmicro® MCU devices to enhance the PICmicro® MCUs program memory space. The interface provided is up to 64k of 16 data bit "words" via a multiplexed address/data bus. The multiplexed bus may seem somewhat difficult to use, but it actually is not; memory devices can be added quite easily and quickly.

There are four memory modes available to the 17Cxx PICmicro® MCUs:
Built-in Hardware Features

PIC17Cxx Memory Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Program memory characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>Internal to the PICmicro® MCU, able to read Configuration Fuses and Read and Write Program Memory</td>
</tr>
<tr>
<td>Protected</td>
<td>Internal to the PICmicro® MCU, able to read Configuration Fuses Program Memory can be read but not Written</td>
</tr>
<tr>
<td>Extended Microcontroller</td>
<td>Program Memory Internal to PICmicro® MCU Accessible. External Memory in Address Space Above Read and Writeable as well. Unable to read Configuration Fuses.</td>
</tr>
<tr>
<td>Microprocessor</td>
<td>No internal Program Memory or Configuration Fuses Accessible. Whole 64k program memory space Accessible outside PICmicro® MCU</td>
</tr>
</tbody>
</table>

These modes can be seen in Fig. 7.39.

An unprogrammed PC17Cxx’s configuration fuses sets the PICmicro® MCU into “microprocessor” mode that cannot access any internal program memory. This allows output devices to be placed into applications, with external program memory providing the application code. This feature allows a way of debugging an application before it is burned into the PICmicro® MCU.

External memory can be read from or written to, using the “TABLRD” and “TABLWT” instructions. In extended microcontrollers and microprocessor modes, the internal program memory can be read using the “TABLRD” instruction in the microcontroller modes. These “Table” instructions use the “Table Pointer” register (“TBLPTRH” for the high 8 bits and “TBLPTRL” for the low 8 bits) to address the operation. During table reads and writes, the “table latch” register (“TABLATH” for the high byte and “TABLATL” for the low byte) is
used to buffer the 16 bits during the transfer because the 17Cxx PICmicro® MCUs processor can only access data 8 bits at a time.

The block diagram for accessing program memory in the 17Cxx family of PICmicro® MCUs is shown in Fig. 7.40. To execute a read or write to program memory, the address in the table pointer has to be first set up. Writing to each of the two 8-bit registers does this. Next, if the operation is a read, the “TABLRD” instruction is executed with a dummy destination to update the table.
The external program memory read is identical to the internal EPROM program memory read.

```
PIC17Cxx External Memory Access

Figure 7.40

Built-in Hardware Features 275

Program Memory

Table Pointer

Address

Table Latch

Data

16 Bits

The external program memory read is identical to the internal EPROM program memory read.

<table>
<thead>
<tr>
<th>PIC17Cxx Program Memory Table Read Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>movlw HIGH_PM_address ; Set up Table Pointer</td>
</tr>
<tr>
<td>movwf TBLPTRH</td>
</tr>
<tr>
<td>movlw LOW_PM_address</td>
</tr>
<tr>
<td>movwf TABLEPTRL</td>
</tr>
<tr>
<td>tablrd 0, 0, WREG                       ; Read High Byte</td>
</tr>
<tr>
<td>tlr 1, WREG</td>
</tr>
<tr>
<td>movwf HIGH Destination</td>
</tr>
<tr>
<td>tablrd 0, 0, WREG                       ; Read Low Byte</td>
</tr>
<tr>
<td>movwf LOW Destination</td>
</tr>
</tbody>
</table>
```
This page intentionally left blank.
Power

Connecting a PICmicro® MCU only requires a 0.01 to 0.1 μF “decoupling” cap across the “Vdd” and “Vss” pins. A typical Power connection is shown in Fig. 8.1. This capacitor should be of low “ESR” type (typically of “tantalum” type).

“Standard” PICmicro® MCUs are designed for anywhere from 4.0 to 6.0 volts of power. Some PICmicro® MCUs have been “qualified” to run from 2.0 to 6.0 volts and are identified for having this capability as being...
“low-voltage” devices. These low-voltage parts are identical to the high-voltage supply parts except that they have been tested at the factory to run with input voltages down to 2.0 volts. Low voltage PICmicro® MCU parts are identified by the addition of the letter “L” before the “C” or “F” in the part number.

Note that the “brown out reset” built into many PICmicro® MCUs is designed to become active at 4.5 volts. This makes the brown out reset incompatible with most low-voltage applications, although there are some
PICmicro® MCUs that have a programmable brown out reset voltage level.

In Fig. 8.2, if Vdd goes below the brown out voltage of the Zener diode, then _MCLR will be pulled low and the PICmicro® MCU will become reset.

The PIC16HV540 has a built-in voltage regulator that allows the PICmicro® MCU to be driven without any external regulators for battery application or poorly regulated power input. The PICmicro® MCU itself is pin and program compatible with the PIC16F54, with PORTA and PORTB having different voltage outputs.

To connect a PIC16HV540 to a battery, the circuit can be as simple as is shown in Fig. 8.3, with “sleep” used for turning the device “off” and putting it in a low-power state.

The device’s block diagram looks like Fig. 8.4.

The voltage regulator can work as either a 5- or 3-volt regulator by setting or resetting, respectively, the “RL

---

**Figure 8.2** “Brown Out” Reset Circuit
Figure 8.3  High-Voltage PICmicro® MCU Connections

Figure 8.4  Actual High-Voltage PICmicro® MCU Circuit
bit of the “option 2” register, which is in the “OPTION/TRIS” address space of the low-end PICmicro® MCU processor. This register is an auxiliary configuration fuses register, which can be modified within an application. The bits of the OPTION2 register are defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Unused</td>
</tr>
<tr>
<td>5</td>
<td>WPC - When set, device will Wake Up On RBO - RB3 changing</td>
</tr>
<tr>
<td>4</td>
<td>SWE - Software Watchdog Timer. If the WDT is not Enabled in the Configuration Fuses, setting this bit will enable it in software</td>
</tr>
<tr>
<td>3</td>
<td>RL - Regulated voltage select bit (Set for 5 Volts, Reset for 3 Volts)</td>
</tr>
<tr>
<td>2</td>
<td>SL - Sleep Voltage Level Setting (if Set, use “RL” Voltage, when Reset, use 3 Volts)</td>
</tr>
<tr>
<td>1</td>
<td>BL - Brown Out Voltage Select. When Set - 3.1 volts for 5 Volt Operation and when Reset - 2.2 Volts for 3 Volt Operation</td>
</tr>
<tr>
<td>0</td>
<td>BE - Brown Out Checking Enabled when Set</td>
</tr>
</tbody>
</table>

OPTION2 is written using the TRIS instruction as:

```
TRIS OPTION2
```

Reset

If the simple reset shown in Fig. 8.5 is used for reset, then the “PWRTE” option should be enabled to allow the PICmicro® MCU’s power input to stabilize before the device starts executing.
Typical PICmicro® MCU output voltages are

\[
\begin{align*}
V_{\text{OL}} (\text{"output low voltage"}) &= 0.6 \, \text{V (max)} \\
V_{\text{OH}} (\text{"output high voltage"}) &= V_{\text{DD}} - 0.7 \, \text{V (min)}
\end{align*}
\]

The input “threshold” voltage, the point at which the input changes from an “I” to an “O” and vice versa, is also dependent on the input power “Vdd” voltage level. The threshold is different for different devices. For a number of different PICmicro® MCU part numbers, this value is specified as being in the range:

\[
0.25 \, V_{\text{DD}} + 0.8 \, \text{V} \geq V_{\text{threshold}} \\
\geq 0.48 \, V_{\text{DD}}
\]

### Digital Logic Interfacing

Parallel busses can be created using PORTB for eight data bits and using other PORT pins for the “_RD” and “_WR” lines as shown in Fig. 8.6. Code to access the Parallel Bus Devices follows.
Writing parallel bus devices is accomplished by the code:

```assembly
bsf STATUS, RPO ; Put PORTB into Input Mode
movlw 0x0FF
movwf TRISB ^ 0x080
bcf STATUS, RPO
bcf PORTA, 0 ; Drop the "_RD" line
call Dlay ; Delay until Data Output Valid
movf PORT B, w ; Read Data from the Port
bsf PORT A, 0 ; "_RD" = 1 (disable "_RD" Line)
```

Figure 8.6 PICmicro® MCU Simulated Parallel IO Port
284 Chapter 8

```assembly
bcf PORTA, 1 ; Enable the "_WR1" Line
movwf PORTB ; output the Data
call Dlay ; Wait Data Receive Valid
bsf PORTA ; "_WR1" = 1.
```

Button Interfacing

The typical button interface circuit is seen in Fig. 8.7.

The first button debouncing macro is inserted in the source code and waits for a Port Pin to reach a set state for a specific amount of time before continuing.

Debounce macro HiLo, Port, Bit
if HiLo == Lo
  btfss Port, Bit ; Is the Button Pressed?
else
  btfsc Port, Bit
endif
  goto $ - 1 ; Yes - Wait for it to be Released
  movlw InitDlay ; Wait for Release to be Debounced
  movwf Dlay ; Have to Delay 20 msecs
  movlw 0
  if HiLo == Lo
    btfss Port, Bit ; If Button Pressed, Wait Again for
    ; it
```

![Figure 8.7 Simple Button Interface](image-url)
else
  btfsc Port, Bit
endif
  goto $ - 6 ; to be Released
ifndef Debug ; Skip Small Loop if "Debug" Defined
  addlw 1 ; Increment the Delay Count
  btfsc STATUS, Z ; Loop If Low Byte (w) Not Equal to Zero
else
  nop ; Match the Number of Instructions
  nop
endif
decfsz Dlay
  goto $ - 5
endm

The "InitDlay" constant is calculated using the formula:

\[
\text{TimeDelay} = \frac{((\text{InitDlay} - 1) \times 256) \times 7}{\left(\frac{\text{Frequency}}{4}\right)}
\]

or

\[
\text{InitDlay} = \frac{((\text{TimeDelay} \times \frac{\text{Frequency}}{4})}{256 \times 7}) + 1
\]

The second button debounce macro works similarly to the Parallax Basic Stamp's PBasic "Button" Function.

Button macro Port, Pin, Down, Delay, Rate, Variable, Target, Address
local ButtonEnd
  incf Variable, w ; Increment the Counter Variable
  if ((Down == 0) && (Target == 0)) || ((Down == 1) && (Target == 1))
    btfsc Port, Pin ; If Low, then Valid Pin
  else
The macro’s parameters are defined as:

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port, Pin</td>
<td>The Button Pin (i.e. &quot;PORTA, 0&quot;)</td>
</tr>
<tr>
<td>Down</td>
<td>The State When the Button is Pressed</td>
</tr>
</tbody>
</table>
Switch Matrix Keypad/Keyboard Interfacing

A switch matrix is simply a two-dimensional matrix of wires, with switches at each vertex. The switch is used to interconnect rows and columns (which are optionally pulled to ground) in the matrix, as can be seen in Fig. 8.8.

![Switch Matrix Diagram]

**Figure 8.8** Switch Matrix with Pull Down Transistors
In this case, by connecting one of the columns to ground, if a switch is closed, the pull down on the row will connect the line to ground. When the row is polled by an I/O pin, a “0” or low voltage will be returned instead of a “1” (which is what will be returned if the switch in the row that is connected to the ground is open).

The PICmicro® MCU is well suited to implementing switch matrix keyboards with PORTB’s internal pull-ups and the ability of the I/O ports to simulate the open-drain pull-downs of the columns as is shown in Fig. 8.9.
Normally, the pins connected to the columns are left in tristate (input) mode. When a column is being scanned, the column pin is output enabled driving a “0” and the four input bits are scanned to see if any are pulled low. In this case, the keyboard can be scanned for any closed switches (buttons pressed) using the code:

```c
int KeyScan() // Scan the Keyboard and
{ // Return when a key is
    int i = 0;
    int key = -1;
    while (key == -1)
    {
        for (i = 0; (i < 4) & ((PORTB & 0x00F) == 0x0F0); i++);
        switch (PORTB & 0x00F) { // Find Key that is
            case 0x00E:            // Pressed
                key = i;
                break;
            case 0x00D:            // Row 1
                case 0x00C:
                    key = 0x04 + i;
                    break;
                case 0x00B:            // Row 2
                    case 0x00A:
                        case 0x009:
                            case 0x008:
                                key = 0x08 + i;
                                break;
                            else // Row 3
                                key = 0x0C + i;
                                break;
                        ///end switch
                    ///end while
                    return key;
                ///end case
            else                   // Row 3
                key = 0x0C + i;
                break;
        } ///end switch
    } // end while
}
```
The “KeyScan” function will only return when a key has been pressed. This routine will not allow keys to be de-bounced or for other code to execute while it is executing. These issues can be resolved by putting the key scan into an interrupt handler, which executes every 5 msecs:

```c
Interrupt KeyScan( ) // 5 msec Interval Keyboard Scan
{
    int i = 0;
    int key = -1
    for (i = 0; i < 4 & ((PORTB & 0x00F) == 0x00F)); i++;
    if (PORTB & 0x00F) != 0x00F) { // Key Pressed
        switch (PORTB & 0x00F) { // Find Key that is Pressed
            case 0x00E: key = i; break;
            case 0x00D: // Row 1
                case 0x00C:
                    key = 0x04 + i;
                    break;
                case 0x00B: // Row 2
                    case 0x00A:
                        case 0x009:
                            case 0x008:
                                key = 0x08 + i;
                                break;
                            else // Row 3
                                key = 0x0C + i;
                                break;
            } // end switch
        if (key == KeySave) {
            keycount = keycount + 1; // Increment Count
            if (keycount == 4) keyvalid = key; // Debounced Key
        } else
            keycount = 0; // No match - Start Again
    }
}```
KeySave = key;  // Save Current key for
next 5 msec
      // Interval
}  // End KeySave

This interrupt handler will set "keyvalid" variable to the row/column combination of the key button (which is known as a "scan code") when the same value comes up four times in a row. For time scan this is the debounce routine for the keypad. If the value doesn't change for four intervals (20 msecs in total), the key is determined to be debounced.

Combining Input and Output

When interfacing the PICmicro® MCU to a driver and receiver (such as a memory with a separate output and input), a resistor can be used to avoid bus contention at any of the pins as is shown in Fig. 8.10.

Buttons can also be put on PICmicro® MCU I/O lines as is shown in Fig. 8.11.

![Figure 8.10 Combining "I/O" on One PICmicro® MCU Pin](image-url)
Simulated “Open Collector”/“Open Drain” I/O

“Open Collector” (“Open Drain”) I/O pins in the PICmicro® MCU are wired as in Fig. 8.12. These pins are available in different devices for different functions. This action can be simulated by using the code listed below that enables the I/O pin output as low if the Carry flag is reset. If the Carry flag is set, then the pin is put into input mode.

```assembly
bcf PORT#, pin ; Make Sure PORTB Pin Bit is "0"
bset STATUS, RPO
btfss STATUS, C ; If Carry Set, Disable Open Collector
goto $ + 4 ; Carry Reset, Enable Open Collector
nop
bset TRIS ^ 0x080, pin
goto $ + 3
```

Figure 8.11 Combining Button Input with Digital I/O
The typical circuit that used to control an LED from a PICmicro® MCU I/O pin is shown in Fig. 8.13. With this circuit, the LED will light when the microcontroller's output pin is set to "0" (ground potential). When the pin is set to input or outputs a '1', the LED will be turned off.

Multisegment LED displays

Seven Segment LED Displays (Fig. 8.14) can be added to a circuit without a lot of software effort. By turning on specific LEDs (each of which lights up a "segment" in...
294 Chapter 8

Figure 8.13 LED Circuit Operation

![Diagram of LED circuit with voltage levels and calculations]

\[ R_i = \frac{(V_{ps} - V_{l1})}{I_{led}} \]
\[ = \frac{(5.0 - 0.7 \text{ Volts})}{20 \text{ mA}} \]
\[ = 4.3 \text{ Volts} / 20 \text{ mA} \]
\[ = 215 \text{ Ohms} \]

Figure 8.14 Organization of a 7-Segment LED Display

![Diagram of 7-segment LED display with labeled segments and common LED pin]
the display), the display can be used to output decimal numbers.

Each one of the LEDs in the display is given an identifier and a single pin of the LED is brought out of the package. The other LED pins are connected together and wired to a common pin. This common LED pin is used to identify the type of Seven Segment Display (as either "Common Cathode" or "Common Anode").

The typical method of wiring multiple Seven Segment LED Displays together is to wire them all in parallel and then control the current flow through the common pin. Because the current is generally too high for a single microcontroller pin, a transistor is used to pass the current to the common power signal. This transistor selects which display is active as shown in Fig. 8.15. In this cir-

---

**Picmicro® MCU Hardware Interfacing** 295

---

**Figure 8.15** Wiring Four 7-Segment LED Displays
cuit, the PICmicro® MCU will shift between the displays showing each digit in a very short "time slice". This is usually done in a Timer Interrupt Handler. The basis for the interrupt handler's code is listed below:

```
Int
- Save Context Registers
- Reset Timer and Interrupt
- LED_Display = 0 ; Turn Off all the LEDs
- LED_Output = Display[ Cur ]
- Cur = (Cur + 1) mod #Displays ; Point to Next Sequence Display
- LED_Display = 1 << Cur ; Display LED for Current Display
- Restore Context Registers
- Return from Interrupt
```

This code will cycle through each of the digits (and displays), with current going through the transistors for each one. To avoid flicker, generally the code should run so that each digit is turned on/off at least 50 times per second. The more digits present, the faster you, the interrupt handler, will have to cycle the interrupt handler (i.e., eight Seven Segment Displays must cycle at least 2,000 digits per second, which is twice as fast as four displays).

**LCD Interfaces**

The most common connector used for the 44780-based LCDs is 14 pins in a row, with pin centers 0.100" apart. The pins are wired as:
The contrast voltage to the display is typically controlled using a potentiometer wired as a voltage divider. This will provide an easily variable voltage between Ground and Vcc, which will be used to specify the contrast (or "darkness") of the characters on the LCD screen. This circuit is shown in Fig. 8.16.

The interface is a parallel bus, allowing simple and fast reading/writing of data to and from the LCD as shown in Fig. 8.17.

This waveform will write an ASCII byte out to the LCD's screen. The ASCII code to be displayed is 8-bits long and is sent to the LCD either 4- or 8-bits at a time. If 4-bit mode is used, two "nybbles" of data (sent high 4-bits and then low 4-bits with an "E" Clock pulse with each nybble) are sent to make up a full 8-bit transfer.

---

**Hitachi 44780 Based LCD Pinout**

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ground</td>
</tr>
<tr>
<td>2</td>
<td>Vcc</td>
</tr>
<tr>
<td>3</td>
<td>Contrast Voltage</td>
</tr>
<tr>
<td>4</td>
<td>&quot;R/S&quot; - Instruction/Register Select</td>
</tr>
<tr>
<td>5</td>
<td>&quot;R/W&quot; - Read/Write LCD Registers</td>
</tr>
<tr>
<td>6</td>
<td>&quot;E&quot; - Clock</td>
</tr>
<tr>
<td>7-14</td>
<td>D0-D7 Data Pins</td>
</tr>
</tbody>
</table>

---

Figure 8.16 LCD Contrast Voltage Circuit
The "E" Clock is used to initiate the data transfer within the LCD.

Sending parallel data as either 4- or 8-bits are the two primary modes of operation. While there are secondary considerations and modes, deciding how to send the data to the LCD is the most critical decision to be made for an LCD interface application.

Eight bit mode is best used when speed is required in an application and at least ten I/O pins are available. Four bit mode requires a minimum of 6 bits. To wire a microcontroller to an LCD in 4-bit mode, just the top 4-bits (DB4-7) are written to.

The "R/S" bit is used to select whether data or an instruction is being transferred between the microcontroller and the LCD. If the bit is set, then the byte at the current LCD "Cursor" position can be read or written. When the bit is reset, either an instruction is being sent to the LCD or the execution status of the last instruction is read back (whether or not it has completed).

The different instructions available for use with the 44780 are shown in the following table:
### Hitachi 44780 Based LCD Commands

<table>
<thead>
<tr>
<th>R/S</th>
<th>R/W</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
<th>Instruction/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Clear Display</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Return Cursor and LCD to Home Position</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>Set Cursor Move Direction</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SC</td>
<td>RL</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>DL</td>
<td>N</td>
<td>F</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>Move Cursor to CGRAM</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>Move Cursor to Display</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>BF</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Poll the &quot;Busy Flag&quot;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Write Hex Character to the Display at the Current Cursor Position</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Read Hex Character at the Current Cursor Position on the Display</td>
</tr>
</tbody>
</table>

The bit descriptions for the different commands are:

- "*" - Not Used/Ignored. This bit can be either "1" or "0"

Set Cursor Move Direction:
- **ID** - Increment the Cursor after Each Byte Written to Display if Set
- **S** - Shift Display when Byte Written to Display
Enable Display/Cursor
   D - Turn Display On(1)/Off(0)
   C - Turn Cursor On(1)/Off(0)
   B - Cursor Blink On(1)/Off(0)

Move Cursor/Shift Display
   SC - Display Shift On(1)/Off(0)
   RL - Direction of Shift Right(1)/Left(0)

Set Interface Length
   DL - Set Data Interface Length 8(1)/4(0)
   N - Number of Display Lines 1(0)/2(1)
   F - Character Font 6x10(1)/5x7(0)

Poll the "Busy Flag"
   BF - This bit is set while the LCD is processing

Move Cursor to CGRAM/Display
   A - Address

Read/Write ASCII to the Display
   H - Data

Reading Data back is best used in applications that require data to be moved back and forth on the LCD (such as in applications which scroll data between lines). The "Busy Flag" can be polled to determine when the last instruction that has been sent has completed processing.

For most applications, there really is no reason to read from the LCD. "R/W" is tied to ground and the software simply waits the maximum amount of time for each instruction to complete. This is 4.1 msecs for clearing the display or moving the cursor/display to the "home position" and 160 usecs for all other commands. As well as making application software simpler, it also frees up a microcontroller pin for other uses.

One area of confusion is how to move to different locations on the display and, as a follow on, how to move to different lines on an LCD display. The following table shows how different LCD displays that use a single


44780 can be set up with the addresses for specific character locations. The LCDs listed are the most popular arrangements available and the "Layout" is given as number of columns by number of lines:

<table>
<thead>
<tr>
<th>LCD</th>
<th>Top</th>
<th>Ninth</th>
<th>Second</th>
<th>Third</th>
<th>Fourth</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Left</td>
<td>Line</td>
<td>Line</td>
<td>Line</td>
<td>Line</td>
<td></td>
</tr>
<tr>
<td>8x1</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 1.</td>
</tr>
<tr>
<td>16x1</td>
<td>0</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 1.</td>
</tr>
<tr>
<td>16x3</td>
<td>0</td>
<td>8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 3.</td>
</tr>
<tr>
<td>8x2</td>
<td>0</td>
<td>N/A</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 1.</td>
</tr>
<tr>
<td>10x2</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 2.</td>
</tr>
<tr>
<td>16x2</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 2.</td>
</tr>
<tr>
<td>20x2</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 2.</td>
</tr>
<tr>
<td>24x2</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 2.</td>
</tr>
<tr>
<td>30x2</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 2.</td>
</tr>
<tr>
<td>32x2</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 2.</td>
</tr>
<tr>
<td>40x2</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 2.</td>
</tr>
<tr>
<td>16x4</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>0x020</td>
<td>0x040</td>
<td>Note 2.</td>
</tr>
<tr>
<td>20x4</td>
<td>0</td>
<td>0x008</td>
<td>0x040</td>
<td>0x020</td>
<td>0x040</td>
<td>Note 2.</td>
</tr>
<tr>
<td>40x4</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Note 4.</td>
</tr>
</tbody>
</table>

Note 1: Single 44780/No Support Chip.
Note 2: 44780 with Support Chip.
Note 3: 44780 with Support Chip. This is quite rare.
Note 4: Two 44780s with Support Chips. Addressing is device specific.

Cursors for the 44780 can be turned on as a simple underscore at any time using the "Enable Display/Cursor" LCD instruction and setting the "C" bit. The "B" ("Block Mode") bit is not recommended as this causes a flashing full character square to be displayed and it really isn’t that attractive.
The LCD can be thought of as a "Teletype" display because in normal operation, after a character has been sent to the LCD, the internal "Cursor" is moved one character to the right. The "Clear Display" and "Return Cursor and LCD to Home Position" instructions are used to reset the Cursor's position to the top right character on the display. An example of moving the cursor is shown in Fig. 8.18.

To move the Cursor, the "Move Cursor to Display" instruction is used. For this instruction, bit 7 of the instruction byte is set with the remaining 7 bits used as the address of the character on the LCD the cursor is to move to. These 7 bits provide 128 addresses, which matches the maximum number of LCD character addresses available. The table above should be used to determine the address of a character offset on a particular line of an LCD display. The LCD Character Set is shown in Fig. 8.19.

![Figure 8.18 Moving an LCD Cursor](image-url)
Figure 8.19  LCD Character Set
Eight programmable characters are available and use codes 0x000 to 0x007. They are programmed by pointing the LCD's "Cursor" to the Character Generator RAM ("CGRAM") Area at eight times the character address. The next 8 bytes written to the RAM are the line information of the programmable character, starting from the top. The "Character Box" is shown in Fig. 8.20.

The user defined character line information is saved in the LCD's "CGRAM" area. This 64 bytes of memory is accessed using the "Move Cursor into CGRAM" instruction in a similar manner to that of moving the cursor to a specific address in the memory with one important difference.

![Offset Diagram](image)

**Figure 8.20** LCD Character "Box"
This difference is that each character starts at eight times its character value. This means that user definable character 0 has its data starting at address 0 of the CGRAM, character 1 starts at address 8, character 2 starts at address 0x010 (16) and so on. To get a specific line within the user definable character, its offset from the top (the top line has an offset of 0) is added to the starting address. In most applications, characters are written to all at one time with character 0 first. In this case, the instruction 0x040 is written to the LCD followed by all the user-defined characters.

Before commands or data can be sent to the LCD module, the module must be initialized. For 8-bit mode, this is done using the following series of operations:

1. Wait more than 15 msecs after power is applied.
2. Write 0x030 to LCD and wait 5 msecs for the instruction to complete.
3. Write 0x030 to LCD and wait 160 usecs for instruction to complete.
4. Write 0x030 AGAIN to LCD and wait 160 usecs or Poll the Busy Flag.
5. Set the Operating Characteristics of the LCD.
   - Write "Set Interface Length"
   - Write 0x010 to turn off the Display
   - Write 0x001 to Clear the Display
   - Write "Set Cursor Move Direction" Setting Cursor Behavior Bits
   - Write "Enable Display/Cursor" & enable Display and Optional Cursor

The first macro is “LCD8”, which provides a basic interface to the LCD with “worst” case start-up delays. To invoke it, the statement

```c
LCD8 DataPort, EPort, EPin, RSPort, RSPin, RWPort, RWPin, Frequency
```
is put in where “DataPort” is the 8-bit I/O port. “EPort” and “EPin” are the “E” clock Definition. “RSPort” and “RSPin” are the “RS” LCD Data Type Input. “RWPort” and “RWPin” are the pins used to poll the LCD for data reply (and are essentially unused). “Frequency” is the PICmicro® MCU operating speed and is used to calculate the delay values. The only variable required for the “LCD8” and “LCD8Poll” macros is the 8-bit variable “Dlay”.

This macro should work with any low-end or mid-range PICmicro® MCUs. Note that the “LCDPORTInit” subroutine cannot be used with low-end PICmicro® MCUs. To initialize the I/O ports the “TRIS” statements will have to be programmed in manually.

```assembly
LCD8 Macro DataPort, EPort, EPin, RSPort, RSPin, RWPort, RWPin, Freq
variable Dlay5Value, Dlay160Value, Dlay160Bit1 = -1, Dlay160Bit2 = -1
variable BitCount = 0
variable Value = 128, Bit = 7

Dlay5Value = ((5007 * (Freq / 1000) / 4000) / 7) + 256
Dlay160Value = (163 * (Freq / 1000) / 4000) / 3

while (Bit >= 0) ; Find the Number of ; Bits and their ; positions in ; “Dlay160Value”
    if ((Dlay160Value & Value) != 0)
        if (Dlay160Bit1 == -1) ; Set the Upper Bit
            Dlay160Bit1 = Bit
        else
            if (Dlay160Bit2 == -1)
                Dlay160Bit2 = Bit
            endif
        endif
        BitCount = BitCount + 1
    endif
endwhile
```

306 Chapter 8
Value = Value >> 1
Bit = Bit - 1
endw
if (BitCount > 2) ; Just Want max two
    Bits
if ((Dlay160Bit1 - 1) == Dlay160Bit2)
    Dlay160Bit1 = Dlay160Bit1 + 1 ; Shift Top up by 1
    Dlay160Bit2 = -1 ; Delete Second
else
    Dlay160Bit2 = Dlay160Bit2 + 1 ; Shift Bottom up by
        1
endif
endif
Dlay5 ; Delay 5 msecs
movlw (Dlay5Value & 0x0FF00) >> 8
movwf Dlay
movlw Dlay5Value & 0x0FF
subwf Dlay, w
xorlw 0x0FF
addwf Dlay, w
btfsc STATUS, Z
decfsz Dlay, f
goto $ - 5
return
LCDPORTInit ; Initialize the I/O
    Ports
    ONLY used by mid-
range
movlw 0x000
movwf DataPort
bcf EPort, EPin
bcf RSPort, RSPin
bcf RWPort, RWPin
bcf STATUS, RP0
bcf EPort, EPin
bcf RSPort, RSPin
bcf RWPort, RWPin
return
LCDIns ; Send the
    Instruction to the
    LCD
movwf DataPort
bcf RSPort, RSPin
if (Freq > 8000000) ; Make Sure Proper
if (Freq < 16000000) ; Delay is In Place
nop
else
goto $ + 1
endif
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper
if (Freq < 16000000) ; Delay is In Place
nop
else
goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
andlw 0x0FC ; Have to Delay 5 msecs?
btfsc STATUS, Z
call Dlay5
return

LCDChar ; Send the Character
movwf DataPort ; to the LCD
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper
if (Freq < 16000000) ; Delay is In Place
nop
else
goto $ + 1
endif
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper
if (Freq < 16000000) ; Delay is In Place

if (Freq < 16000000)
    nop
else
    goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit-2 != -1)
    bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
return

LCDInit                        ; Do the 8 Bit
; Initialization
; Wait 15 msecs
call Dlay5
call Dlay5
call Dlay5
movlw 0x030
call LCDIns ; Send the Reset
; Instruction
call Dlay5
movlw 0x030
call LCDIns
movlw 0x030
call LCDIns
movlw 0x038 ; Set Interface
; Length
call LCDIns
movlw 0x010 ; Turn Off Display
call LCDIns
movlw 0x001 ; Clear Display RAM
call LCDIns
movlw 0x006 ; Set Cursor
; Movement
call LCDIns
movlw 0x00E ; Turn on
; Display/Cursor
call LCDIns
return
endm
The “LCD8Poll” macro is slightly more sophisticated than the “LCD8” macro. Instead of providing “hard-coded” delays in the application, the code “polls” the LCD to see if the Operation is complete before continuing. This is done by putting the “DataPort” into “Input Mode” and then strobing the “E” bit (with RS reset and RW set) and looking at bit 7 of the I/O port. The macro code is

```
LCD8Poll Macro DataPort, EPort, EPin, RSPort, RSPin, 
    RWPort, RWPin, Freq 
variable Dlay5Value, Dlay160Value, Dlay160Bit1 = 
    -1, Dlay160Bit2 = -1 
variable BitCount = 0 
variable Value = 128, Bit = 7 
errorlevel 0, -224 
Dlay5Value = ((5007 * (Freq / 1000) / 4000) / 7) + 256 
Dlay160Value = (163 * (Freq / 1000) / 4000) / 3 
while (Bit >= 0) ; Find the Number of 
    ; Bits and their 
    ; Positions in 
    ; “Dlay160Value” 
    if ((Dlay160Value & Value) != 0) 
        if (Dlay160Bit1 == -1) ; Set the Upper Bit 
            Dlay160Bit1 = Bit 
        else 
            if (Dlay160Bit2 == -1) 
                Dlay160Bit2 = Bit 
            endif 
        endif 
    endif 
    BitCount = BitCount + 1 
endif 
Value = Value >> 1 
Bit = Bit - 1 
endw 
if (BitCount > 2) ; Just Want max two 
    ; Bits 
    if ((Dlay160Bit1 - 1) == Dlay160Bit2) 
        Dlay160Bit1 = Dlay160Bit1 + 1 ; Shift Top up by 1 
        Dlay160Bit2 = -1 ; Delete Second 
else 
```
Dlay160Bit2 = Dlay160Bit2 + 1 ; Shift Bottom up by 1
endif
endif
Dlay5 ; Delay 5 msecs
movlw (Dlay5Value & 0x0FF00) >> 8
movwf Dlay
movlw Dlay5Value & 0x0FF
subwf Dlay, w
xorlw 0x0FF
addwf Dlay, w
btfsc STATUS, Z
decfsz Dlay, f
goto $ - 5
return

LCDPORTInit ; Initialize the I/O Ports
bsf STATUS, RP0 ; ONLY used by mid-range
movlw 0x000
movwf DataPort
bcf EPort, EPin
bcf RSPort, RSPin
bcf RWPort, RWPin
bcf STATUS, RP0
bcf EPort, EPin
bcf RSPort, RSPin
bcf RWPort, RWPin
return

LCDIns ; Send the Instruction to the LCD
movwf Dlay
movlw 0x0FF ; Read the "BF" Flag
tris DataPort
bcf RSPort, RSPin ; Read the Instruction Register
bsf RWPort, RWPin
goto $ + 1
bsf EPort, EPin
nop
movf DataPort, w ; Read the Data Port Value
nop
bcf  EPort, EPin
andlw 0x080
; Is the High Bit
; Set?
bffs  STATUS, Z
goto $ - 7
bcf  RWMPort, RWPin
movlw 0
; Put the DataPort
; Back into Output
; Mode
tris  DataPort
movf  Dlay, w
; Get the Saved
; Character
movwf  DataPort
if (Freq > 8000000)
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf  EPort, EPin
if (Freq > 8000000)
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf  EPort, EPin
return

LCDChar
; Send the Character
; to the LCD
movwf  Dlay
movlw 0xFF
; Read the "BF" Flag
tris  DataPort
bcf  RSPort, RSPin
; Read the
; Instruction
; Register
bsf  RWMPort, RWPin
goto $ + 1
bcf  EPort, EPin
nop
movf DataPort, w        ; Read the Data Port Value
nop
bcf EPort, EPin
andlw 0x080             ; Is the High Bit Set?
btfss STATUS, Z
goto $ - 7
bcf RWPort, RWPin
movlw 0                 ; Put the DataPort Back into Output Mode
tris DataPort
movf Dlay, w            ; Get the Saved Character
movwf DataPort
if (Freq > 8000000)     ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
else
goto $ + 1
endif
endif
bsf EPort, EPin
if (Freq > 8000000)     ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
else
goto $ + 1
endif
bcf EPort, EPin
return
LCDInit                  ; Do the 8 Bit Initialization
call Dlay5
call Dlay5
call Dlay5
movlw 0x030
movwf DataPort
if (Freq > 8000000)     ; Make Sure Proper Delay is In Place

if (Freq < 16000000)
nop
else
goto $ + 1
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper
 ; Delay is In Place
endif
bcf EPort, EPin ; Send the Reset ; Instruction
call Dlay5
if (Freq > 8000000) ; Make Sure Proper
 ; Delay is In Place
endif
if (Freq < 16000000)
nop
else
goto $ + 1
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper
 ; Delay is In Place
endif
if (Freq < 16000000)
nop
else
goto $ + 1
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper
 ; Delay is In Place
endif
if (Freq < 16000000)
nop
else
goto $ + 1
endif
bsf EPort, EPin
bsf Dlay, Dlay160Bit1 ; Send the Reset ; Instruction
if (Dlay160Bit2 != -1) ; Delay 160 usecs
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movlw 0x030
call LCDIns
The LCD should be initialized in 4-bit mode, data is written to the LCD in terms of nybbles. This is done because initially just single nybbles are sent (and not two, which make up a byte and a full instruction). When a byte is sent, the high nybble is sent before the low nybble and the "E" pin is toggled each time a nybble is sent to the LCD. To initialize in 4-bit mode:

1. Wait more than 15 msecs after power is applied.
2. Write 0x03 to LCD and wait 5 msecs for the instruction to complete.
3. Write 0x03 to LCD and wait 160 usecs for instruction to complete.
4. Write 0x03 AGAIN to LCD and wait 160 usecs (or poll the Busy Flag).
5. Set the Operating Characteristics of the LCD.
   - Write 0x02 to the LCD to Enable Four Bit Mode

All following instruction/Data Writes require two nybble writes:

- Write "Set Interface Length"
- Write 0x01/0x00 to turn off the Display
The 4-bit LCD interfacing (the “LCD4” Macro) is modified from the “LCD8” macro. To invoke the macro, the similar statement

```
LCD4 DataPort, DataBit, EPort, EPin, RSPort, RSPin, RWPort, RWPin, Freq
```

is used. The “DataBit” parameter is lowest of the four data bits. It can only be “0” or “4”. The macro requires the “LCDTemp” Variable along with “Dlay”. The Macro is

```
LCD4 Macro DataPort, DataBit, EPort, EPin, RSPort, RSPin, RWPort, RWPin, Freq
variable Dlay5Value, Dlay160Value, Dlay160Bit1 = -1, Dlay160Bit2 = -1
variable BitCount = 0
variable Value = 128, Bit = 7
Dlay5Value = ((5007 * (Freq / 1000) / 4000) / 7) + 256
Dlay160Value = (163 * (Freq / 1000) / 4000) / 3
if ((DataBit != 0) && (DataBit != 4))
    error "Invalid 'DataBit' Specification - Can only be '0' or '4'"
endif
while (Bit >= 0)
    ; Find the Number of
    ; Bits and their
    ; Positions in
    ; "Dlay160Value"
    if ((Dlay160Value & Value) != 0)
        if (Dlay160Bit1 == -1)
            ; Set the Upper Bit
            Dlay160Bit1 = Bit
        else
            ; Set the Lower Bit
            Dlay160Bit2 = Bit
        endif
    endif
    BitCount = BitCount + BitCount + BitCount + BitCount
    Bit = Bit - 1
endwhile
```
if (Dlay160Bit2 == -1)
Dlay160Bit2 = Bit
endif
endif
BitCount = BitCount + 1
Bit = Bit - 1
endw
if (BitCount > 2) ; Just Want max two
; Bits
if ((Dlay160Bit1 - 1) == Dlay160Bit2)
Dlay160Bit1 = Dlay160Bit1 + 1 ; Shift Top up by 1
Dlay160Bit2 = -1 ; Delete Second
else
Dlay160Bit2 = Dlay160Bit2 + 1 ; Shift Bottom up
; by 1
endif
endif
Dlay5 ; Delay 5 msecs
movlw (Dlay5Value & 0x0FF00) >> 8
movwf Dlay
movlw Dlay5Value & 0x0FF
subwf Dlay, w
movlw 0x0FF
addwf Dlay, w
btfsc STATUS, Z
decfsz Dlay, f
goto $ - 5
return
LCDPORTInit ; Initialize the I/O
; Ports
bsf STATUS, RP0 ; ONLY used by mid-
if (DataBit == 0)
movlw 0x0F0
else
movlw 0x00F
endif
movwf DataPort
bcf RPort, EPin
bcf RSPort, RSPin
bcf RWPort, RWPin
bcf STATUS, RP0

PiCmicro® MCU Hardware Interfacing 317
318 Chapter 8

    bcf  EPort, EPin
    bcf  RSPort, RSPin
    bcf  RWPort, RWPin
    return

LCDIns    ; Send the
    ; Instruction to
    ; the LCD
    movwf  LCDTemp
    if (DataBit == 0)
      swapf  LCDTemp, w    ; Most Significant
      andlw  0x00F
    else
      andlw  0x0F0
    endif
    movwf  DataPort
    bcf  RSPort, RSPin
    if (Freq > 8000000)    ; Make Sure Proper
      ; Delay is In Place
    if (Freq < 16000000)
      nop
    else
      goto  $ + 1
    endif
    endif
    bcf  EPort, EPin
    if (Freq > 8000000)    ; Make Sure Proper
      ; Delay is In Place
    if (Freq < 16000000)
      nop
    else
      goto  $ + 1
    endif
    endif
    bcf  EPort, EPin
    if (DataBit == 0)
      movf  LCDTemp, w
      andlw  0x00F
    else
      swapf  LCDTemp, w    ; Least Significant
      andlw  0x0F0
    endif
movf DataPort, w
bcf RSPort, RSPin
if (Freq > 8000000) ; Make Sure Proper
if (Freq < 16000000) ; Delay is In Place
    nop
else
    goto $ + 1
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper
if (Freq < 16000000) ; Delay is In Place
    nop
else
    goto $ + 1
endif
bsf EPort, EPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
    bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movf LCDTemp, w
andlw 0x0FC                ; Have to Delay 5
btfsc STATUS, Z            ; msecs?
call Dlay5
return

LCDChar                        ; Send the Character
movwf LCDTemp                 ; Save the Value
if (DataBit == 0)
    swapf LCDTemp, w         ; Most Significant
else
    andlw 0x00F
    andlw 0x0F0
endif
movwf DataPort
bcf RSPort, RSPin
if (Freq > 8000000)           ; Make Sure Proper
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
endif
bsf EPort, EPin
if (Freq > 8000000)           ; Make Sure Proper
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
endif
bcf EPort, EPin
if (DataBit == 0)
    movf LCDTemp, w
    andlw 0x00F
else
    swapf LCDTemp, w           ; Least Significant
    andlw 0x0F0
endit
movwf DataPort
bsf RSPort, RSPin
if (Freq > 8000000)           ; Make Sure Proper
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
endif
bsf EPort, EPin
if (Freq > 8000000)           ; Make Sure Proper
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
endif
PlCmicro® MCU Hardware Interfacing 321

```assembly
bcf EPort, EPin
bcf Dlay, Dlay160Bit1   ; Delay 160 usecs
if (Dlay160Bit2 != -1)
  bcf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
return

LCDInit
 ; Do the 8 Bit
 ; Initialization
; Wait 15 msecs
; Send the Reset
; Instruction
movlw 0x003
else
movlw 0x030
endif
movwf DataPort
if (Freq > 8000000)
  ; Make Sure Proper
  ; Delay is In Place
if (Freq < 16000000)
  nope
else
  goto $ + 1
endif
endif
bcf EPort, EPin
if (Freq > 8000000)
  ; Make Sure Proper
  ; Delay is In Place
if (Freq < 16000000)
  nope
else
  goto $ + 1
endif
endif
bcf EPort, EPin
bcf EPort, EPin
 ; Send Another Reset
 ; Instruction
if (Freq > 8000000)
  ; Make Sure Proper
  ; Delay is In Place
if (Freq < 16000000)
```
nop
else
    goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
    bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
    goto $ - 1
bsf EPort, EPin ; Send the Third
    Reset Instruction
if (Freq > 8000000) ; Make Sure Proper
    goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
    bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
    goto $ - 1
if (DataBit == 0) ; Send the Data
    movlw 0x002
else
    movlw 0x020
endif
movf DataPort ; Make Sure Proper
if (Freq > 8000000) ; Delay is In Place
    goto $ + 1
endif
if (Freq < 16000000)
    movlw 0x002
else
    movlw 0x020
endif
PlCmicro® MCU Hardware Interfacing  

It is recommended that the I/O pins and the 4-bit "DataPort" are on the same 8-bit I/O Port. The reason for doing this is that when using this code, writes to the "DataPort" will change the output values of other I/O register bits.

The interface requirements to the PlCmicro® MCU can be reduced by using the circuit shown in Fig. 8.21
in which the serial data is combined with the contents of the shift register to produce the “E” strobe at the appropriate interval. This circuit “ANDs” (using the 1K resistor and IN914 diode) the output of the sixth “D-Flip Flop” of the 74LS174 and the “Data” bit from the device writing to the LCD to form the “E” Strobe. This method requires one less pin than a three-wire shift register interface and a few more instructions of code. The two-wire LCD interface circuit is shown in Fig. 8.21.

The 74LS174 can be wired as a shift register (as is shown in the schematic diagram) instead of a serial-in/parallel-out shift register. This circuit should work without any problems with a dedicated serial-in/parallel-out shift register chip, but the timings/clock polarities may be different. When the 74LS174 is used, note that
the data is latched on the rising (from logic “low” to “high”) edge of the clock signal. Figure 8.22 is a timing diagram for the two-wire interface and shows the 74LS174 being cleared, loaded, and then the “E” Strobe when the data is valid and “6Q” and incoming “Data” is high.

Before data can be written to it, loading every latch with zeros clears the shift register. Next, a “1” (to provide the “E” Gate) is written followed by the “R/S” bit and the four data bits. Once the latch is loaded in correctly, the “Data” line is pulsed to Strobe the “E” bit. The biggest difference between the three-wire and two-wire interface is that the shift register has to be cleared before it can be loaded and the two-wire operation re-

**Figure 8.22** 2-Wire LCD Write Waveform
quires more than twice the number of clock cycles to load 4-bits into the LCD.

One note about the LCD’s “E” Strobe is that in some documentation it is specified as “high” level active while in others, it is specified as falling edge active. It seems to be falling edge active, which is why the two-wire LCD interface presented below works even if the line ends up being high at the end of data being shifted in. If the falling edge is used (like in the two-wire interface) then make sure that before the “E” line is output on “0”, there is at least a 450 nsecs delay with no lines changing state.

The two-wire LCD interface macro uses the same parameters as the previous macros. This interface is quite a bit slower than the other ones that I have presented, but uses the fewest PICmicro® MCU I/O pins. The “LCD2” Macro only requires the “Dlay” and “LCDTemp” variables.

```c
LCD2 Macro ClockPort, ClockPin, DataPort, DataPin, Freq

variable Dlay5Value, Dlay160Value, Dlay160Bit1 = -1, Dlay160Bit2 = -1
variable BitCount = 0, i
variable Value = 128, Bit = 7
Dlay5Value = ((5907 * (Freq / 1000) / 4000) / 7) + 256
Dlay160Value = (163 * (Freq / 1000) / 4000) / 3

while (Bit >= 0)              ;  Find the Number of
   Bits and their
   Positions in
   "Dlay160Value"
   if ((Dlay160Value & Value) != 0)
      if (Dlay160Bit1 == -1)        ;  Set the Upper Bit
         Dlay160Bit1 = Bit
      else
         if (Dlay160Bit2 == -1)
            Dlay160Bit2 = Bit
         endif
      endif
```
BitCount = BitCount + 1
endif
Value = Value >> 1
Bit = Bit - 1
endw
if (BitCount > 2) ; Just Want max two
    Bits
if ((Dlay160Bit1 - 1) == Dlay160Bit2)
    Dlay160Bit1 = Dlay160Bit1 + 1 ; Shift Top up by 1
    Dlay160Bit2 = -1 ; Delete Second
else
    Dlay160Bit2 = Dlay160Bit2 + 1 ; Shift Bottom up ; by 1
endif
endif
Dlay5 ; Delay 5 msecs
        movlw (Dlay5Value & 0x0FF00) >> 8
        movwf Dlay
        movlw Dlay5Value & 0x0FF
        subwf Dlay, w
        xorlw 0x0FF
        addwf Dlay, w
        btfsc STATUS, Z
        decfsz Dlay, f
        goto $ - 5
        return
LCDPORTInit ; Initialize the I/O Ports
bsf STATUS, RP0 ; ONLY used by mid-range
bcf ClockPort, ClockPin
bcf DataPort, DataPin
bcf STATUS, RP0
bcf ClockPort, ClockPin
bcf DataPort, DataPin
return
LCDIns ; Send the Instruction to the LCD
        movwf LCDTemp ; Save the Value
        movlw 6 ; Clear the Shift Register

movf Dlay, w
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
decfsz Dlay, f
goto $ - 3
movwf Dlay
movf LCDTemp, w
swapf LCDTemp, f
bsf LCDTemp, 5
bcf LCDTemp, 4
bcf DataPort, DataPin
btfsc LCDTemp, 5
bsf DataPort, DataPin
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin
if (Freq > 8000000)
  movw LCDTemp
  bsf ClockPort, ClockPin
  bcf ClockPort, ClockPin
  decfsz Dlay, f
goto $ - 3
else
  goto $ + 1
endif
endif
bsf DataPort, DataPin
bsf Dlay, 2
bsf Dlay, 1
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
decfsz Dlay, f
goto $ - 3
movwf LCDTemp
bsf Dlay, 2
bsf Dlay, 1
bsf LCDTemp, 5 ; Make LCDTemp Correct for Shifting
bcf LCDTemp, 4 ; This is "RS" Bit
bcf DataPort, DataPin ; Shift Out Each Bit
btfsc LCDTemp, 5 ; 5 is the Current MSB
bsf DataPort, DataPin ; Shift Out the Next Highest Bit
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin ; Latch in the Data
if (Freq > 8000000) ; Make Sure Proper Delay is In Place
    nop
else
goto $ + 1
endif
bsf DataPort, DataPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
    bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
andlw 0x0FC ; Have to Delay 5 msecs?
btfsc STATUS, Z
call Dlay5
return
LCDChar ; Send the Character to the LCD
movwf LCDTemp ; Save the Value Clear the Shift Register
movlw 6
movwf Dlay
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
decfsz Dlay, f
goto $ - 3
movf Dlay,

movf LCDTemp, w

swapf LCDTemp, f

bsf LCDTemp, 5

bsf LCDTemp, 4

bcf DataPort, DataPin

btfsc LCDTemp, 5

bsf DataPort, DataPin

bsf ClockPort, ClockPin

rlf LCDTemp, f

decfsz Dlay, f

goto $ - 7

bsf DataPort, DataPin

if (Freq > 8000000)

else

goto $ + 1
endif

endif

bcf DataPort, DataPin

bsf Dlay, 2

bsf ClockPort, ClockPin

bsf ClockPort, ClockPin

decfsz Dlay, f

goto $ - 3

movf LCDTemp

bsf Dlay, 2

bsf Dlay, 1

bsf LCDTemp, 5

bsf LCDTemp, 4

bcf DataPort, DataPin

w still equals 6

Shift out the

Upper 4 Bits

Make LCDTemp

Correct for

Shifting

This is "RS" Bit

Shift Out Each Bit

S is the Current

MSB

Shift Out the Next

Highest Bit

Shift Out Each Bit

Latch in the Data

Make Sure Proper

Delay is In Place

nopr

else

goto $ + 1
endif

bcf DataPort, DataPin

bsf Dlay, 2

bsf ClockPort, ClockPin

Clear the Shift

Register

goto $ - 3

movf LCDTemp

bsf Dlay, 2

bsf Dlay, 1

bsf LCDTemp, 5

bsf LCDTemp, 4

bcf DataPort, DataPin

Shift out the Low

Nybble

Dlay = 6 for Shift

Out

Dlay = 6 for Shift

Out

Make LCDTemp

Correct for

Shifting

This is "RS" Bit

Shift Out Each Bit
PiCmicro® MCU Hardware Interfacing 331

btffsc LCDTemp, s          ; 5 is the Current
bsf DataPort, DataPin      ; MSB
bsf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin     ; Latch in the Data
if (Freq > 8000000)        ; Make Sure Proper
if (Freq < 16000000)
  nop
else
  goto $ + 1
endif
bcf DataPort, DataPin     ; Delay is In Place
if (Dlay160Bit2 != -1)
  bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
return LCDInit            ; Do the 8 Bit
  ; Initialization
  ; Wait 15 msecs
  ; Port
  ; Initialize the I/O
  ; Save the Value
  ; Clear the Shift
  ; Register
movlw 0x023                ; Initialize the I/O
movlw 6                    ; Clear the Shift
movwf LCDTemp              ; Save the Value
movwf Dlay                 ; Shift Out Each Bit
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
decfsz Dlay, f
goto $ - 3
movwf Dlay
bsf DataPort, DataPin      ; Shift Out Each Bit
btffsc LCDTemp, s          ; 5 is the Current
bcf ClockPort, ClockPin
btfsc LCDTemp, 5           ; 5 is the Current
bcf ClockPort, ClockPin
bsf DataPort, DataPin
bsf DataPort, DataPin ; Shift Out the Next
        ; Highest Bit
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin ; Latch in the Data
if (Freq > 8000000) ; Make Sure Proper
        ; Delay is In Place
    if (Freq < 16000000)
        nop
    else
goto $ + 1
endif
endif
bcf DataPort, DataPin
bcf DataPort, DataPin
if (Freq > 8000000) ; Send another 0x03
    if (Freq < 16000000)
        nop
    else
goto $ + 1
endif
endif
bcf DataPort, DataPin
bcf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
    bcf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
bsf DataPort, DataPin ; Send another 0x03
if (Freq > 8000000) ; to the LCD
    if (Freq < 16000000)
        nop
    else
goto $ + 1
endif
endif
```assembly
bcf DataPort, DataPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
    bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movlw 0x022 ; Initialize the I/O Port
movwf LCDTemp ; Save the Value
movlw 6 ; Clear the Shift Register
movwf Dlay
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
decfsz Dlay, f
goto $ - 3
movwf Dlay
bsf DataPort, DataPin ; Shift Out Each Bit
btfs CLDTemp, 5 ; 5 is the Current MSB
bsf DataPort, DataPin ; Shift Out the Next Highest Bit
bsf ClockPort, ClockPin
bsf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin ; Latch in the Data
if (Freq > 8000000) ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
    else
goto $ + 1
endif
else
goto $ + 1
endif
bcf DataPort, DataPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
    bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movlw 0x028 ; Set Interface Length
call LCDIns
movlw 0x010 ; Turn Off Display
```
```assembly
    ; Clear Display RAM
    call    LCDIns
    movlw   0x001
    call    LCDIns
    movlw   0x006
    call    LCDIns
    movlw   0x00E
    call    LCDIns
    return
endm

I2C Bit Banging “Master” Interface

For the interface code below, make sure there is a 1K to 10K pull up on the SCL and SDA lines.

I2CSetup Macro ClockPort, ClockPin, DataPort, DataPin, Rate, Frequency
variable Dlay, Fraction ; Delay in Instruction Cycles
Dlay = ((Frequency * 110) / (800 * Rate)) / 1000
Fraction = ((Frequency * 110) / (800 * Rate)) - (Dlay * 1000)
if (Fraction > 499)
  Dlay = Dlay + 1
endif
I2CBitSetup ; Setup I2C Lines for Application
bsf     STATUS, RP0
bcf     ClockPort, ClockPin ; Driving Output
bcf     DataPort, DataPin
bcf     STATUS, RP0
bsf     ClockPort, ClockPin ; Everything High Initially
bsf     DataPort, DataPin
DlayMacro Dlay ; Make Sure Lines are High for adequate Period of Time
return
```
I2CStart ; Send a "Start"
   ; Pulse to the I2C Device
   bsf ClockPort, ClockPin
   bsf DataPort, DataPin
   DlayMacro Dlay - 2
   bcf DataPort, DataPin ; Drop the Data Line
   DlayMacro Dlay
   bcf ClockPort, ClockPin ; Drop the Clock Line
   DlayMacro Dlay - 2 ; Wait for the Specified Period
   return ; Exit with Clock = Low, Data = Low

I2CStop ; Pass Stop Bit to I2C Device
   DlayMacro Dlay
   bsf ClockPort, ClockPin ; Clock Bit High
   DlayMacro Dlay
   bsf DataPort, DataPin
   return ; Exit with Clock = High, Data = High

I2CRead ; Read 8 Bits from the Line
   ; Reply with "ACK" in Carry Flag
   ; Put in the Carry Flag
   bsf I2CTemp, 0
   btfsc STATUS, C
   bcf I2CTemp, 0 ; If Carry Set, then Send "Ack" (-ative)
   bsf STATUS, RP0 ; Let the I2C Device Drive the Data Line
   bsf DataPort, DataPin
   movlw 0x010 - 8
   I2CRLoop
   bsf ClockPort, ClockPin ; Bring the Clock Line Up
   DlayMacro (Dlay / 2) - 1
336 Chapter 8

bcf STATUS, C ; Sample the
bsf STATUS, C ; Incoming Data

DlayMacro (Dlay / 2) - 2 ; Shift in the Bit
bcf ClockPort, ClockPin
rlf I2CTemp, f ; Store the Ack of
andlw 0x07F ; Bit 7 of the Data
bsf STATUS, C
iorlw 0x080 ; If High, Set Bit 7
addlw 0x001 ; Finished, Do the
DlayMacro Dlay - 9 ; Next Bit
bsf STATUS, DC ; Put in "Tlow"
goto I2CBLoop
bfc STATUS, RP0 ; Send Ack Bit
bcf DataPort, DataPin
bcf STATUS, RP0
andlw 0x080 ; High or Low?
bftss STATUS, Z
bsf DataPort, DataPin ; Low, Send Ack
DlayMacro Dlay / 18 ; Any Reason to
delay?
bf ClockPort, ClockPin
DlayMacro Dlay
bcf ClockPort, ClockPin
bcf DataPort, DataPin
movf I2CTemp, w ; Get the Received
return ; Byte
I2CSend ; Return with Clock
; = Data = Low

I2CBLoop ; Send the 8 Bits in
movwf I2CTemp ; "w" and Return
movlw 0x010 - 8
I2CBLoop
rlf I2CTemp, f ; Shift up the Data
bcf STATUS, C ; into "C"
goto $ + 4
nop
bcf DataPort, DataPin ; Low Bit
PiCmicro® MCU Hardware Interfacing

```assembly
goto $ + 3
bsf DataPort, DataPin ; High Bit
goto $ + 1
bsf ClockPort, ClockPin ; Strobe Out the
DlayMacro Dlay
bcf ClockPort, ClockPin
DlayMacro Dlay - 12
addlw 1
bsfss STATUS, DC
goto I2CSLoop
DlayMacro 6
bsf STATUS, RP0 ; Now, Get the Ack
bsf DataPort, DataPin
bcf STATUS, RP0
bsf ClockPort, ClockPin
DlayMacro (Dlay / 2) - 1
bcf STATUS, C
bsfss DataPort, DataPin
bsf STATUS, C ; Line Low, "Ack"
DlayMacro (Dlay / 2) - 2
bsf STATUS, RP0
bcf DataPort, DataPin
bcf STATUS, RP0
bcf ClockPort, ClockPin
bcf DataPort, DataPin
return ; Return with Ack in
endm ; Clock = Data = Low
```

The macro is similar to

```
I2CSetup I2CClock, I2CData, Rate, Frequency
```

where

```
Pin Description
I2CClock Port and Pin used for the "SCL" line - Pulled up with 1K to 10K Resistor
I2CData Serial Data - Pulled up with 1K to 10K Resistor
```
Rate  I2C Data Rate specified in kHz (normally 100 or 400)
Frequency PICmicro® MCU’s Clock Frequency

Data is sent to an I2C Device using the format:
idle - Start - CommandWriteA - AddressByteA - Start
- CommandReadA - DataA - DataN - Stop - idle

Using the subroutines in the “I2CSetup” macro, the PICmicro® MCU code for carrying out a 16-bit read would be

```assembly
call I2CStart ; Start the Transfer
movlw CommandWrite ; Send the Address
call I2CSend ; to Read the Sixteen Bit Word
movlw AddressByte
call I2CSend

call I2CStart ; Reset the I2C EEPROM to Read Back
movlw CommandRead ; Send the Read Command
call I2CSend
bsf STATUS, C ; Read the Byte with Ack

call I2CRead
movwf I2CData
bsf STATUS, C ; Read the next byte and stop the Transfer with the Nack

call I2Cread
movwf I2CData + 1

call I2CStop ; Finished with the I2C Operation
```
RS-232 is an older standard with somewhat unusual voltage levels. A “Mark” ("1") is actually −12 volts and a “Space” ("0") is +12 volts. Voltages in the “switching region” (±3 volts) may or may not be read as a “0” or “1” depending on the device. The “Handshaking” lines use the same logic levels as the transmit/receive lines discussed above and are used to interface between devices and control the flow of information between computers. The “Request To Send” ("RTS") and “Clear To Send” ("CTS") lines are used to control data flow between the computer ("DCE") and the modem ("DTE" device). When the PC is ready to send data, it asserts (outputs a “Mark”) on RTS. If the DTE device is capable of receiving data, it will assert the “CTS” line. If the PC is unable to receive data (i.e., the buffer is full or it is processing what it already has), it will de-assert the “RTS” line to notify the DTE device that it cannot receive any additional information.

The “Data Transmitter Ready” ("DTR") and “Data Set Ready” ("DSR") lines are used to establish communications. When the PC is ready to communicate with the DTE device, it asserts “DTR”. If the DTE device is available and ready to accept data, it will assert “DSR” to notify the computer that the link is up and ready for data transmission. If there is a hardware error in the link, then the DTE device will de-assert the DSR line to notify the computer of the problem. Modems if the carrier between the receiver is lost will de-assert the DSR line.

“Data Carrier Detect” ("DCD") is asserted when the modem has connected with another device (i.e., the other device has “picked up the phone”). The “Ring Indicator” ("RI") is used to indicate to a PC whether or
not the phone on the other end of the line is ringing or if it is busy. These lines (along with the other handshaking lines) are very rarely used in PICmicro® MCU applications.

There is a common ground connection between the DCE and DTE devices. This connection is critical for the RS-232 level converters to determine the actual incoming voltages. The ground pin should never be connected to a chassis or shield ground (to avoid large current flows or being shifted, preventing accurate reading of incoming voltage signals).

Most modern RS-232 connections are implemented using a “Three-Wire RS-232” set up as shown in Fig. 8.23. DTR/DSR and RTS/CTS lines are normally shorted.
together at the PICmicro® MCU end. The DCD and RI lines are left unconnected. With the handshaking lines shorted together, data can be sent and received without having to develop software to handle the different handshaking protocols. The 1488/1489 RS-232 Level Converter Circuits is a common method of implementing RS-232 Serial Port Interfaces if +12 and −12 volts is available to the circuit. The “#C” input is a flow control for the gates (normally RS-232 comes in the “#A” pin and is driven as TTL out of “#Y”) and is normally left floating ("unconnected"). The pinout and wiring for these devices in a PC are shown in Fig. 8.24. If only a +5 volt power supply is available, the MAX232 chip can be used to provide the correct RS-232 signal levels. This circuit is wired as shown in Fig. 8.25. Another method for translating RS-232 and TTL/CMOS voltage levels is to use the transmitter’s negative voltage. The circuit in Fig. 8.26 shows how this can be done using a single MOSFET transistor and two resistors.

Figure 8.24 1488/1489 RS-232 Connections
Figure 8.25  MAXIM MAX232 RS-232 Connections

Figure 8.26  RS-232 to External Device
This circuit relies on the RS-232 communications only running in “Half-Duplex” mode (i.e., only one device can transmit at a given time). When the external device wants to transmit to the PC, it sends the data either as a “Mark” (leaving the voltage to be returned to the PC as a negative value) or as a “Space” by turning on the transistor and enabling the positive voltage output to move to the PC’s receivers.

When the PICmicro® MCU transmits a byte to the external device through this circuit, it will receive the packet it is sent because this circuit connects the PICmicro® MCU’s receiving pin (more or less) directly to its transmitting pin. The software running in the PICmicro® MCU (as well as the “host” device) will have to handle this.

Another issue to notice is that data out of the external device will have to be inverted to get the correct transmission voltage levels (i.e., a “0” will output a “1”) to make sure the transistor turns on at the right time (i.e., a positive voltage for a space). Unfortunately, this means that the built-in serial port in the PICmicro® MCU cannot be used with this circuit because the output is “positive” and it cannot invert the data as required by the circuit.

There is a chip, the Dallas Semiconductor DS275, which basically incorporates the above-mentioned circuit (with built-in inverters) into the single package shown in Fig. 8.27. The DS1275 is an earlier version of this chip.

With the availability of low current PICmicro® MCUs, the incoming RS-232 lines can be used to power the application. This can be done using the host’s RS-232 Ports to supply the current needed by the application as shown in Fig. 8.28.

When the DTR and RTS lines are outputting a “Space”, a positive voltage (relative to ground) is available. This
voltage can be regulated and the output used to power the devices attached to the serial port (up to about 5 mA). For extra current, the TX line can also be added into the circuit as well with a “break” being sent from the PC to output a positive voltage.

The 5 mA is enough current to power the Transistor/Resistor type of RS-232 Transmitter and a PICmicro®
MCU running at 4 MHz along with some additional hardware (such as a single LCD).

**RS-485/RS-422**

RS-485/RS-422 are "differential pair" serial communications electrical standards that consist of a balanced driver with positive and negative outputs that are fed into a comparator. The output from the comparator is a "1" or a "0" depending on whether or not the "positive line" is at a higher voltage than the negative. Figure 8.29 shows the normal symbols used to describe a differential pair connection.

To minimize AC transmission line effects, the two wires should be twisted around each other. "Twisted pair" wiring can either be bought commercially or made by simply twisting two wires together, twisted wires have a characteristic impedance of 75 ohms or greater.

A common standard for differential pair communications is "RS-422". This standard, which uses many commercially available chips, provides:

1. Multiple receiver operation.
2. Maximum data rate of 10 mbps.
3. Maximum cable length of 4,000 meters (with a 100 kHz signal).

![Figure 8.29 Differential Pair Serial Data Transmission](image)
Multiple receiver operation, shown in Fig. 8.30, allows signals to be “broadcasted” to multiple devices. The best distance and speed changes with the number of receivers of the differential pair along with its length. The 4,000 m at 100 kHz or 40 m at 10 MHz are examples of this balancing between line length and data rate. For long data lengths a few hundred ohm “terminating resistor” may be required between the positive terminal and negative terminal at the end of the lines to minimize reflections coming from the receiver and affecting other receivers.

RS-485 is very similar to RS-422, except it allows multiple drivers on the same network. The common chip is the “75176”, which has the ability to drive and receive on the lines as shown in Fig. 8.31.

The only issue to be on the lookout for when creating RS-485/RS-422 connections is to keep the cable polarities correct (positive to positive and negative to negative). Reversing the connectors will result in lost signals and misread transmission values.
Asynchronous Serial I/O Software Routines

The first method is a traditional "bit banging" interface that can be used by both the low-end and mid-range PICmicro® MCUs. To set up the serial interfaces, the macro

```
NR2SerialNI Macro TXPort, TXPin, RXPort, RXPin, 
Polarity, Rate, Freq
```

is invoked, where "TXPort" and "TXPin" along with "RXPort" and "RXPin" are used to define the Transmit port and the Receive port, respectively. As I will discuss in the next section, these pairs of pins can be combined into a single define to make the definition easier. The "Polarity" of the signals is defined as "Pos" for "Positive" or positive logic and "Neg" for "Negative" or inverted logic (useful for interfacing to RS-232 directly through a current limiting resistor). "Rate" is the data rate (in bits per second) and "Freq" is the speed at which the processor is executing in Hz.

When the macro is expanded, the bit delay calculations are made internally and the bit banging serial receive and
transmit subroutines are inserted into the application. The macros can be used by either low-end or mid-range PICmicro® MCUs without modification. The macro code is

```
NRXSerialRX Macro TXPort, TXPin, RXPort, RXPin, polarity, rate, frequency
variable BitDelay
BitDelay = frequency / (4 * rate)
SerialRX ; Receive 8-N-1
if (polarity == Pos)
  btfsc RXPort, RXPin ; Wait for a Bit to
  btfss RXPort, RXPin ; Come in
else
  goto $ - 1
endif
DlayMacro BitDelay / 2 ; Wait 1/2 a Bit to
  if (polarity == Pos)
    btfsc RXPort, RXPin ; Confirm Data is
  else
    btfss RXPort, RXPin ; Correct
  endif
goto SerialRX ; If Just a "Glitch", ; Restart Start Bit
movlw 8 ; Poll ; Wait for 8 Bits
SRXLoop
if ((BitDelay - 10) > 770) ; Check to See if
  DlayMacro 770 ; Value is Too Large
  DlayMacro BitDelay - (770 + 10) ; Put in a "Double" ; Delay
else
  DlayMacro BitDelay - 10 ; Wait for the Middle ; of the Next Bit
endif
bcf STATUS, C ; Check the Incoming Data
```
if (Polarity == Pos)
  btfsc RXPort, RXPin
else
  btfss RXPort, RXPin
endif
bsf STATUS, C
rrf NRZTemp, f ; Shift in the Bit
subwf NRZTemp, w ; Decrement and End if
  == 0
xorlw 0x0FF
addwf NRZTemp, w
btfss STATUS, Z
goto SRXLoop
if ((BitDlay - 9) > 770) ; Check to See if
  DlayMacro 770 ; Value is Too Large
DelayMacro BitDlay - (770 + 9)
else
  DlayMacro BitDlay - 9 ; Wait for the Middle
  of the Next Bit
endif
if (Polarity == Pos) ; Is there a Stop Bit?
  btfss RXPort, RXPin
else
  btfsc RXPort, RXPin
endif
  goto SerialRX ; No, Start All Over
movf NRZTemp, w ; Return the Received
return ; Byte
return ; Note - Zero Returned
SerialTX ; in Low-End
movwf NRZTemp ; Save the Byte to
movlw 10 ; Output
bcf STATUS, C ; Start with Sending
  goto SerialRX ; the Start Bit
```assembly
STXLoop
if (Polarity == Pos)        ; Output the Current
    btfsc STATUS, C
else
    btfss STATUS, C
endif
    goto $ + 4             ; 6 Cycles Required
    nop
    bcf TXPort, TXPin     ; Output a "Low"
    goto $ + 3
    bsf TXPort, TXPin     ; Output a "High"
    goto $ + 1
if ((BitDlay - 15) > 770)   ; Check to See if
    DlayMacro 770               ; Value is Too Large
    DlayMacro BitDlay - (770 + 15)
else
    DlayMacro BitDlay - 15      ; Wait for the Middle
    endif
    subwf NRZTemp, w        ; Decrement the Bit
    xorlw 0x0FF
    addwf NRZTemp, w
    btfsc STATUS, Z
    return                    ; Can Return to Caller
    bsf STATUS, C         ; Shift Down the Next
    rrf NRZTemp, f
    goto STXLoop
endif
```

Mid-Range “Bit Banging” NRZ Serial Interface Initialization Code

```
NRZSerialNISetup Macro TXPort, TXPin, Polarity
SerialSetup                  ; Setup the Serial I/O
    bsf STATUS, RP0
    bcf TXPort, TXPin
endm
```
The TMR0 interrupt based asynchronous serial functions shown in the macro below poll the data bit three times each data period to check for the incoming data. This method does not prevent the PICmicro® MCU from carrying out any other tasks.

```
NRZSerialI Macro TXPort, TXPin, RXPort, RXPin, Polarity, Rate, Frequency
variable BitDlay, Prescaler, TMR0Reset
BitDlay = (Frequency / (3 * 4 * Rate)) - 10
TMR0Reset = BitDlay / 2 ; Using TMR0,
            ; Calculate the
            ; Timer Reset Value
Prescaler = 0 ; And the Prescaler
while (TMR0Reset > 0x0FF) ; Find the Proper
    TMR0Reset = TMR0Reset / 2
    Prescaler = Prescaler + 1
endw
if (Prescaler > 7) ; Can't Use TMR0
    error "Bit Delay cannot use TMR0 for Polling Clock"
endif
TMR0Reset = 256 - TMR0Reset ; Get the TMR0 Reset
                        ; Value
goto AfterInt ; Jump to After
org 4 ; Interrupt Handler
movf _w
movf STATUS, w
bcf STATUS, RP0 ; Make Sure in Bank 0
movf _status
```

PICmicro® MCU Hardware Interfacing 351
bcf    INTCON, T0IF    ; Reset the Timer
        ; Interrupt

movlw   TMR0Reset
movwf   TMR0

; First, Check for a Received Character
Int_RX

movlw   0x004             ; Check for Bit?
addwf   RXCount, f
btfss   STATUS, DC        ; DC Not Affected by
                ; "clrf"
                ; Nothing to Check for
                ; (Yet)
                
movf    RXCount, w        ; Everything Read
                ; Through?
xorlw   0x091
btfsc   STATUS, Z
                ; Yes, Check for Stop
                ; Bit

bcf    STATUS, C        ; Read the Current
                ; State
        if (Polarity == Pos)
        btfsc   RXPort, RXPin  ; Sample at 10 Cycles
        else
        btfsc   RXPort, RXPin
        endif

bsf    STATUS, C    ; Start Counting from 4
_RXEnd13

nop

goto    _RXEnd

_end8

goto    $ + 1

352 Chapter 8


PiCmicro® MCU Hardware Interfacing 353

nop
goto _RXEnd13

_RXNo

btfsc RXCount, 0
  goto _RXEnd8

btfsc RXCount, 3
  goto _RXStartCheck

if (Polarity == Pos)
  btfsc RXPort, RXPin
else
  btfss RXPort, RXPin
endif

bcf RXCount, 2
  goto _RXEnd13

_RXStartCheck

if (Polarity == Pos)
  btfsc RXPort, RXPin
else
  btfss RXPort, RXPin
endif

movlw 0x0FF
  bcf RXCount
addlw 1
  movwf RXCount

goto _RXEnd

_RXAtEnd

if (Polarity == Pos)
354 Chapter 8

```assembly
btfs RXPort, RXPin
else
   btfs RXPort, RXPin
endif
bsf RXFlag

clrf RXCount           ; Finished - Reset
; Check - 12
; Cycles

goto $ + 1

goto _RXEnd

_RXEnd

; Next, Check for Transmitting a Character - Intrinsic Dlay 22 Cycles
Int_TX

movlw 0x004             ; Interrupt Transmit
; Increment Value
addwf TXCount, f
btfss STATUS, DC        ; Send the Next Byte?
goto _TXSendDlayCheck

bsf TXCount, 2        ; Want to Increment 3x
; not Four for each
; Bit
bsf STATUS, C
rrf TXByte, f
movf TXPort, w         ; Send Next Bit
andlw 0xOFF ^ (1 << TXPin)
if (Polarity == Pos)
   btfsc STATUS, C
else
   btfs STATUS, C
endif
iorlw 1 << TXPin
movwf TXPort            ; Cycle 12 is the Bit
; Send

goto _TXCompletedGoOn ; TX Takes 14 Cycles

_TXSendDlayCheck       ; Don't Send Bit,
; Check for Start Bit
```
btfss TXCount, 0 ; Bit Zero Set (Byte to Send)?
goto _TXNothingtoCheck

movlw 0x004 ; Setup the Timer to Increment 3x
movwf TXCount

movf TXPort, w ; Output Start Bit
if (Polarity == Pos)
    andlw 0x0FF ^ (1 << TXPin)
else
    iorlw 1 << TXPin
endif
movwf TXPort

goto _TXCompletedGoOn ; TX First Bit Takes 14 Cycles

_TXNothingtoCheck ; Nothing Being Sent?
movf TXCount, w ; Zero (Originally)
xorlw 0x004 ; TXCount?
btfss STATUS, Z
xorlw 0x004 ^ 0x09C
btfsc STATUS, Z
clrf TXCount

_TXCompletedGoOn ; Finished with TX, Do RX

movf _status, w ; Restore the Interrupts
movwf STATUS
swapf _w, f
swapf _w, w
retfie

SerialRX

bcf RXFlag ; Reset the Character available Flag

btfss RXFlag ; Wait for a Character
    ; to be Received
goto $ - 1
movf RXByte, w    ; Return the Character
        ; Read in
        return

SerialTX

movf TXCount, f ; Anything Being Sent?
btfss STATUS, Z ; Wait for the
        ; Previous Send to
        ; End
        goto $ - 2
movwf TXByte    ; Send out the
        ; Character
bsf TXCount, 0  ; Indicate to the
        ; Interrupt Handler
        ; that it can Send
        ; Something
        return

AfterInt        ; Can Return the Value
        ; Setup the
        ; Interrupts/TX
        ; Output
bsf STATUS, RP0
bcf TXPort, TXPin
movlw 0x0D0 + Prescaler
movwf OPTION_REG ^ 0x080; User Prescaler with
        ; TMR0
bcf STATUS, RP0
if (Polarity == Pos)
        bsf TXPort, TXPin    ; Output 'Idle' for
        ; Data Transmit
else
        bcf TXPort, TXPin
endif
movlw TMR0Reset  ; Reset the Timer
movf TMR0
movlw [1 << GIE] + (1 << TOIE)
movwf INTCON      ; Start up the
        ; Interrupts
clr RF0Count      ; Make Sure No Counts
        ; are Starting
clr TXCount
endm
Along with the macro invocation, the following variables will have to be declared for the code to work:

- _w, _status
- RXByte, TXByte
- RXCount, TXCount
- Flags

Along with these variables the “RXFlag” bit will also have to be defined for use by the code to indicate when a valid byte has been received.

**Dallas Semiconductor One-Wire Interface**

Dallas Semiconductor has created a line of peripherals that are very attractive for use with microcontrollers as they only require one line for transferring data. This single-wire protocol is available in a variety of devices, but the most popular are the DS1820 and DS1821 digital thermometers. These devices can be networked together on the same bus (they have a built-in serial number to allow multiple devices to operate on the same bus) and are accurate to within one degree Fahrenheit.

The DS1820 is available in a variety of packages, with the most common being a three-pin “TO-92” package that looks like a plastic transistor package and can be wired to a PICmicro® MCU as shown in Fig. 8.32.

The DS1820 has many features that would be useful in a variety of different applications. These include the ability of sharing the single-wire bus with other devices. A unique serial number is burned into the device that allows it to be written to individually and gives it the ability to be powered by the host device. Data transfers over the “one-Wire” bus are initiated by the Host system (in the application cases, this is the PICmicro® MCU) and
are carried out 8-bits at a time (with the least significant bit transmitted first). Each bit transfer takes at least 60 usec. The “one-Wire” bus is pulled up externally (as is shown in Fig. 8.32) and is pulled down by either the host or the peripheral device to transfer data. If the Bus is pulled down for a very short interval, a “1” is being transmitted. If the Bus is pulled down for more than 15 usecs, then a “0” is being transmitted. The differences in the “1” and “0” bits are shown in Fig. 8.33.

All Data Transfers are initiated by the host system. If it is transmitting data, then it holds down the line for the specified period. If it is receiving data from the DS1820, then the host pulls down the line and releases it and then polls the line to see how long it takes for it to go back up. During Data Transfers, make sure that Interrupts cannot take place (because this would affect how the data is sent or read if the interrupt takes place during the data transfer).

Before each command is set to the DS1820, a “Reset” and “Presence” Pulse is transferred. A “Reset” Pulse con-

Figure 8.32 Example Thermometer Application
sists of the host pulling down the line for 480 usecs to 960 usecs. The DS1820 replies by pulling down the line for roughly 100 usecs (60 to 240 usecs is the specified value). To simplify the software, do not check for the presence pulse (because I knew in the application that I had the thermometer connected to the bus). In another application (where the thermometer can be disconnected), putting a check in for the "Presence" Pulse may be required.

To carry out a temperature "read" of a single DS1820 connected to a microcontroller, the following instruction sequence is used:

1. Send a "Reset" Pulse and delay while the "Presence" Pulse is returned.
2. Send 0x0CC, which is the "Skip ROM" command, which tells the DS1820 to assume that the next command is directed towards it.
3. Send 0x044, which is the Temperature Conversion Initiate instruction. The current temperature will be sampled and stored for later read back.
4. Wait 500+ usecs for the Temperature Conversion to complete.
5. Send a "Reset" Pulse and delay while the "Presence" Pulse is returned.
6. Send 0x0CC, "Skip ROM" Command Again.
7. Send 0x0BE to read the "Scratchpad" RAM that contains the Current Temperature (in degrees Celsius times two).
8. Read the nine bytes of "Scratchpad" RAM.
9. Display the Temperature (if bit 0 of the second byte returned from the "Scratchpad" RAM, the first byte is negated and a "-" is put on the LCD Display) by dividing the first byte by 2 and sending the converted value to the LCD.

The total procedure for doing a temperature measurement takes about 5 msecs. PICmicro® MCU code to access the DS1820 is given in the following listings for a PICmicro® MCU running at 4 MHz:

```assembly
DSReset                  ; Reset the DS1820
    bcf      DS1820        ; Hold the DS1820 Low for 500 usecs to reset
    movlw    125
    addlw    0x0FF         ; Add -1 until Reset is Equal to Zero
    btfss    STATUS, Z
    goto    $ - 2
    bsf      DSTRIS
    movlw    0             ; Wait 1 msec before sending a command
    addlw    0x0FF
    btfss    STATUS, Z
    goto    $ - 2
    bsf      DSTRIS
    bsf      DS1820
    return

DS1820 Byte Send Code
DSSend                   ; Send the Byte in "w" to the DS1820
    movwf    Temp
    movlw    8
    DSSendLoop
    bcf      INTCON, GIE ; Make Sure Operation isn't interrupted
```

360 Chapter 8
**PICmicro® MCU Hardware Interfacing**

```assembly
; Drop the DS1820's
bcf DS1820

; Control Line
rrf Temp, f

; Load Carry with Contents
btsec STATUS, C

; if "1" Sent, Restore
bsf DS1820

; After 4 Cycles
bsf Count, 3

decfsz Count, f

goto $ - 1

; Loop for 24 Cycles
goto $ + 1 ; Add 2 Cycles for a 30

; usec delay
bsf DS1820

; The Line is High
bsf INTCON, GIE

; Restore the Interrupts
bsf Count, 3

; Loop Another 24 Cycles

decfsz Count, f

goto $ - 1

; for Execution Delay
addlw 0x0FF ; Subtract 1 from the Bit

btfss STATUS, Z

; Finished, Return to Caller
goto DSSendLoop

return

DS1820 Byte Read Code

; Receive the Byte from
DSRead

movlw 8 ; the DS1820 and put in

DSReadLoop

; "w"

bcf INTCON, GIE ; Make Sure Operation

; isn't interrupted

bcf DS1820 ; Drop the DS1820's

; Control Line

bsf DSTRIS ; Turn Port into a

; simulated Open Drain

; Output

nop

; What is Being Returned?
bfc STATUS, C

; If Line is high, a "1"
bfc DS1820

; Shift in the Data bit

rrf Temp, f

; Can Interrupt from here

bsf INTCON, GIE

clrf Count

decfsz Count, f

goto $ - 1

bsf DS1820
```
The Process to read from the DS1820 is

1. Reset the DS1820
2. Send 0CCh followed by 044h to begin the Temperature sense and conversion.
3. Wait 480 usecs for the Temperature conversion to complete.
4. Send another reset to the DS1820
5. Send 0CCh and 0BEh to read the temperature
6. Wait 100 usecs before reading the first byte in the DS1820
7. Read the first, or "SP0" byte of the DS1820
8. Wait another 100 usecs before reading the second or "SP1" byte of the DS1820

Reading a Potentiometer Using Parallel I/O Pins

For measuring resistance values without an ADC, a simple RC network can be used with the PICmicro® MCU as is shown in Fig. 8.34. To measure the resistance (assuming the capacitor is of a known value), the PICmicro® MCU first charges the capacitor to 5 volts (or its nominal output) using the I/O pin in “output” mode. Once this is done, the pin changes to “input” mode and waits for the capacitor to discharge through the potentiometer. Looking at this operation on an oscilloscope, the waveform produced by the circuit looks like Fig. 8.35. In Fig. 8.35, the “Charge” cycle and “Discharge” cycle can
Figure 8.34  Reading a Potentiometer Position without an ADC

Figure 8.35  Oscilloscope Picture for ADCLess Operation
clearly be seen. From basic electronic theory, we know that the time required for the capacitor to charge is

\[
time = \frac{R \times C \times \ln(V_{\text{end}} / V_{\text{start}})}{\ln(V_{\text{end}} / V_{\text{start}})}
\]

where the \( V_{\text{start}} \) and \( V_{\text{end}} \) are the starting and ending voltages. Since the capacitor value, the voltages and the time it took for the capacitor to discharge, the formula above can be rearranged to find \( R \):

\[
R = \frac{\text{time} \times \ln(V_{\text{end}} / V_{\text{start}})}{C}
\]

The code used to test the analog I/O uses the following logic:

```c
int PotRead() // Read the Resistance at 
                // the I/O Pin
{
    int i;
    TRIS.Pin = Output; // Set the Output Mode
    Pin = 1;           // Output a "1" to Charge
                        // the Capacitor
    for (i = 0; i < 5usec, i++);
    TRIS.Pin = Input;  // Now, Time How Long it
    TMR0 = 0;          // Takes for
                        // the Capacitor to
                        // Discharge through
                        // the Potentiometer
    while (Pin == 1);
    return TMR0;       // Return the TMR0 Value
                        // for the
                        // Discharge Time
} // end PotRead
```

**Motor Drivers**

A network of switches (transistors) can be used to control turning a motor in either direction. This is known as an “H-Bridge” and is shown in Fig. 8.36.
In this circuit, if all the switches are open, no current will flow and the motor won’t turn. If switches “1” and “4” are closed, the motor will turn in one direction. If switches “2” and “3” are closed, the motor will turn in the other direction. Both switches on one side of the bridge should never be closed at the same time as this will cause the motor power supply to burn out or a fuse to blow because there is a short circuit directly between the motor power and ground.

Controlling a motor’s speed is normally done by “pulsing” the control signals in the form of a Pulse Wave Modulated (“PWM”) signal as shown in Fig. 8.37. This will control the average power delivered to the motors. The higher the ratio of the “Pulse Width” to the “Period”, the more power delivered to the motor.

The frequency of the PWM signal should be greater than 20 kHz to avoid the PWM from producing an audible signal in the motors as the field is turned on and off.
The 293D chip can control two motors (one on each side of a robot), connected to the buffer outputs (pins 3, 6, 11, and 14). Pins 2, 7, 10, and 15 are used to control the voltage level (the “switches” in the H-Bridge diagram above) of the buffer outputs. Pin 1 and Pin 9 are used to control whether or not the buffers are enabled. These can be PWM inputs, which makes control of the motor speed very easy to implement. “Vs” is +5 volts used to power the logic in the chip and “Vss” is the power supplied to the motors and can be anywhere from 4.5 to 36 volts. A maximum of 500 mA can be supplied to the motors. The 293D contains integral shunt diodes. This means that to attach a motor to the 293D, no external shunt diodes are required as shown in Fig. 8.38. In Fig. 8.38, there is an optional “snubber” resistor and capacitor. These two components, wired across the brush contacts of the motor will help reduce electromagnetic emissions and noise “spikes” from the motor. Erratic operation from the microcontroller when the motors are running can be corrected by adding the 0.1 uF capacitor and 5 ohm (2 watt) resistor “snubber” across the motor’s brushes as shown in the circuit above.
There is an issue with using the 293D and 298 motor controller chips which is that they are bipolar devices with a 0.7 volt drop across each driver (for 1.4–1.5 volts for a dual driver circuit as is shown in Fig. 8.38. This drop, with the significant amount of current required for a motor, results in a fairly significant amount of power dissipation within the driver. The 293D is limited to 1 Amp total output and the 298 is limited to 3 Amps. For these circuits to work best, a significant amount of heat sinking is required.

To minimize the problem of heating and power loss Power MOSFET transistors can be used to implement an H-Bridge motor control.

Stepper motors are much simpler to develop control software for than regular DC motors. This is because the motor is turned one step at a time or can turn at a spe-
Specific rate (specified by the speed in which the “Steps” are executed). In terms of the hardware interface, stepper motors are a bit more complex to wire and require more current (meaning that they are less efficient), but these are offset by the advantages in software control.

A “Bipolar” Stepper motor consists of a permanent magnet on the motor’s shaft that has its position specified by a pair of coils (Fig. 8.39). To move the magnet and the shafts, the coils are energized in different patterns to attract the magnet. For the example above, the following sequence would be used to turn the magnet (and the shaft) clockwise.

<table>
<thead>
<tr>
<th>Step</th>
<th>Angle</th>
<th>Coil “A”</th>
<th>Coil “B”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>90</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>180</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>270</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>360/0</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

In this sequence, Coil “A” attracts the North Pole of the magnet to put the magnet in an initial position. Then Coil “B” attracts the South Pole, turning the magnet 90
degrees. This continues on to turn the motor, 90 degrees for each "Step".

The output shaft of a stepper motor is often geared down so that each step causes a very small angular deflection (a couple of degrees at most rather than the 90 degrees in the example above). This provides more torque output from the motor and greater positional control of the output shaft.

R/C Servo Control

Servos designed for use in radio-controlled airplanes, cars, and boats can be easily interfaced to a PICmicro MCU. They are often used for robots and applications where simple mechanical movement is required. This may be surprising to you because a positional servo is considered to be an analog device. The output of an R/C Servo is usually a wheel that can be rotated from zero to 90 degrees. (There are also servos available that can turn from zero to 180 degrees as well as servos with very high torque outputs for special applications). Typically, they only require +5 volts, ground, and an input signal.

An R/C Servo is indeed an analog device, the input is a PWM signal at digital voltage levels. This pulse is between 1.0 and 2.0 msecs long and repeats every 20 msecs (Fig. 8.40).

The length of the PWM Pulse determines the position of the servo’s wheel. A 1.0 msec pulse will cause the wheel to go to zero degrees while a 2.0 msecs pulse will cause the wheel to go to 90 degrees.

For producing a PWM signal using a PICmicro MCU, the TMR0 timer interrupt (set for every 20 msecs),
which outputs a 1.0 to 2.0 msecs PWM signal can be used. The pseudo-code for the interrupt handler is

```c
Interrupt() {                  //  Interrupt Handler
    int    i = 0;
    BitOutput( Servo, 1);        //  Output the Signal
    for (i = 0; i < (1 msec + ServoDlay); i++ );
    BitOutput( Servo, 2);
    for (; i < 2 msec; i++ );    //  Delay full 2 msecs
}  //  End Interrupt Handler
```

Audio Output

The PICmicro® MCU can output audio signals using a circuit like the one shown in Fig. 8.41.

Timing the output signal is generally accomplished by toggling an output pin at a set period within the TMR0 in-
interrupt handler. To generate a 1kHz signal shown in a PICmicro® MCU running a 4MHz, you can use the code below (which does not use the prescaler) for TMR0 and the PICmicro® MCU’s interrupt capability.

```
org     4
int
    movwf   _w                   ;  Save Context
    movwf   _w                   ;  Registers
    bcf     INTCON, TOIF         ; Reset the
    bcf     INTCON, TOIF         ; Interrupt
    movlw   256
    movwf   TMRO                 ; Reset TMR0 for
    movlw   256 - (250 - 4)
    movwf   TMRO                 ; another 500 usecs
    btfsc   SPKR                 ;  Toggle the Speaker
    goto   $ + 2
    bsf     SPKR                 ; Speaker Output
    bsf     SPKR                 ; High
    goto   $ + 2
    bcf     SPKR                 ; Speaker Output Low
    swapf   _w, f                ; Restore Context
    swapf   _w, w                 ; Registers
    retfie
```

Figure 8.41  Circuit for Driving PICmicro® MCU Audio
AC Power Control

“TRIACS” come under the heading of “Thyristors” and are used to switch AC signals on and off. TRIACS do not rectify the AC voltage because they consist of two “Silicon Controlled Rectifiers” (“SCRs”), which allows the AC Current to pass without any “clipping”. A typical circuit for Triacs is shown in Fig. 8.42.

TRIACS do not allow AC current to pass unless their “gates” are biased relative to the two AC contacts. To do this, a PICmicro® MCU output can pull the control to ground. The current required to “close” many of the TRIACS is 25 mA and can be provided by standard PICmicro® MCU outputs easily.

CAUTION: AC voltages and currents can damage components, start fires, burn, or even kill through electrocution. It is recommended that AC control circuits are tested with low-voltage sources before they are used in “mains” voltage circuits.

![Typical TRIAC AC Control Circuit](Figure 8.42)
Hall-Effect Sensors

A “Hall-effect switch” is a device in which if a current passing through a piece of silicon is deflected by a magnetic field, the output changes state as shown in Fig. 8.43.

The Hall-effect switch output can either be “Totem Pole” or “Open Collector” and can drive a PICmicro® MCU input directly. If an “Open Collector” output is used with the Hall-effect switch, then a pull up is required to ensure positive voltages will be received by the PICmicro® MCU when there is no magnetic field in place.

Sony Infrared TV Remote Control

Most (if not all) I/R TV remotes use a “Manchester” encoding scheme in which the data bits are embedded in the packet by varying the lengths of certain data levels. This can be seen in Fig. 8.44, from a theoretical Perspective and in Fig. 8.45, which shows the output from a 40-kHz Infrared Receiver receiving a signal from a “Sony” brand TV remote control. The normal signal
Figure 8.44 I/R TV remote data stream.

Figure 8.45 Scope View of TV I/R Remote Control Input.
coming from an I/R receiver circuit is “High” when nothing is coming (line idle) and then goes “low” with a “leader” signal to indicate that data is coming in. The data consists of a bit “Synch,” which when it completes the bit value is transmitted as the length of time before the next bit “Synch”.

“Sony” brand TV remotes have 12 data bits and a 40-kHz carrier. The timings are as follows (and use a “Base Timing” “T” of 550 usecs).

<table>
<thead>
<tr>
<th>Feature</th>
<th>&quot;T&quot; Timing</th>
<th>Actual Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader</td>
<td>4T</td>
<td>2.20 msec</td>
</tr>
<tr>
<td>Synch</td>
<td>T</td>
<td>0.55 msec</td>
</tr>
<tr>
<td>&quot;0&quot;</td>
<td>T</td>
<td>0.55 msec</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>2T</td>
<td>1.10 msec</td>
</tr>
</tbody>
</table>

To read the incoming signal, the following code can be used in a PICmicro® MCU running at 4 MHz and a single I/R receiver can be used to pass the signal to the PICmicro® MCU:

```
clrf IntCount         ;  Reset the
                    ;  Counters
clrf ReadCount
GetPack            ;  Get the Next
                    ;  Packet Coming In
movlw 0x088        ;  Wait for Port
movwf INTCON       ;  Change Interrupt
```
Loop                           ;  Loop Here for
;  Each Update of
;  the Screen

movlw   150                  ;  Wait for 25 msec
subwf   IntCount, w
btfss   STATUS, Z
goto   Loop                 ;  Has NOT timed out

clrf    INTCON               ;  No more interrupts
;    for a while

movf    ReadCount, w         ;  Get the Read in
;   CRC

clrf    IntCount             ;  Reset for the
;   Next Packet

clrf    ReadCount

call    DispHex              ;  Now, Display the
;   Character

movlw   0x08E                ;  Reset the Cursor

call    WriteINS

goto   GetPack              ;  Wait for the Next
;   I/R Packet

Int                            ;  Interrupt, Check
;   I/R Input

movwf   _w                   ;  Save the Context

swapf   STATUS, w

movwf   _status

movlw   0x020                ;  Just wait for a
;   Timer Interrupt

movwf   INTCON

movlw   256 - 20             ;  Reset the Timer
This code starts sampling the incoming data after the Leader was received and the “1”s and “0”s were treated as the inputs to a Linear Feedback Shift Register. For the Code above, an 8-bit LFSR was used to produce “Cyclical Redundancy Check” (“CRC”) codes. In this
case, the Input wasn’t the high bit of the shift register—instead it is the input from the I/R Receiver.

Using this code, the following CRC codes were generated from the “Sony” I/R TV Remote Control Transmitter:

<table>
<thead>
<tr>
<th>Key</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>0x052</td>
</tr>
<tr>
<td>Vol+</td>
<td>0x05E</td>
</tr>
<tr>
<td>Vol-</td>
<td>0x0BB</td>
</tr>
<tr>
<td>Ch+</td>
<td>0x0DC</td>
</tr>
<tr>
<td>Ch-</td>
<td>0x062</td>
</tr>
<tr>
<td>&quot;0&quot;</td>
<td>0x017</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>0x07A</td>
</tr>
<tr>
<td>&quot;2&quot;</td>
<td>0x08D</td>
</tr>
<tr>
<td>&quot;3&quot;</td>
<td>0x033</td>
</tr>
<tr>
<td>&quot;4&quot;</td>
<td>0x01F</td>
</tr>
<tr>
<td>&quot;5&quot;</td>
<td>0x04E</td>
</tr>
<tr>
<td>&quot;6&quot;</td>
<td>0x072</td>
</tr>
<tr>
<td>&quot;7&quot;</td>
<td>0x0CC</td>
</tr>
<tr>
<td>&quot;8&quot;</td>
<td>0x0B9</td>
</tr>
<tr>
<td>&quot;9&quot;</td>
<td>0x023</td>
</tr>
</tbody>
</table>
“Hex” File Format

The purpose of MPLAB and other assemblers and compilers is to convert PICmicro® MCU application source code into a data format that can be used by a programmer to load the application into a PICmicro® MCU. The most popular format (used by Microchip and most other programmers, including the two presented in this chapter) is the Intel 8-bit hex file format.

When source code is assembled, a hex file ("Example.hex") is generated. This file is in the format:
Each line consists of a starting address and data to be placed starting at this address. The different positions of each line are defined by:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>First (1)</td>
<td>Always &quot;:&quot; to indicate the Start of the Line.</td>
</tr>
<tr>
<td>2-3</td>
<td>Two Times the Number of Instructions on the Line.</td>
</tr>
<tr>
<td>4-7</td>
<td>Two Times the Starting Address for the Instructions on the Line. This is in &quot;Motorola&quot; Format (High Byte followed by Low Byte).</td>
</tr>
<tr>
<td>8-9</td>
<td>The Line Type (00 - Data, 01 - End).</td>
</tr>
<tr>
<td>10-13</td>
<td>The First Instruction to be loaded into the PICmicro® MCU at the Specified Address. This data is in &quot;Intel&quot; Format (Low Byte followed by High Byte).</td>
</tr>
<tr>
<td>:</td>
<td>Additional Instructions to be loaded at Subsequent Addresses. These instructions are also in &quot;Intel&quot; Format.</td>
</tr>
<tr>
<td>End - 4</td>
<td>The Checksum of the Line.</td>
</tr>
<tr>
<td>End - 2</td>
<td>ASCII Carriage Return/Line Feed Characters</td>
</tr>
</tbody>
</table>

The checksum is calculated by summing each byte in a line and subtracting the least significant bits from 0x0100. For the second line in the example hex file above, the checksum is calculated as:
The least significant 8-bits (0x0DC) are subtracted from 0x0100 to get the checksum:

\[
\begin{align*}
0x0100 & - 0x00DC \\
\hline
0x0024 & \\
\end{align*}
\]

This calculated checksum value of 0x024 is the same as the last two bytes of the original line.

**Low-End PICmicro® MCU Programming**

The low-end PICmicro® MCUs use 17 pins for programming and are programmed using a “Parallel” protocol. The pins are defined as:

<table>
<thead>
<tr>
<th>Pins</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA0-RA3</td>
<td>D0-D3 of the Instruction Word</td>
</tr>
<tr>
<td>RB0-RB7</td>
<td>D4-D11 of the Instruction Word</td>
</tr>
</tbody>
</table>
A programmer designed for low-end PICmicro® MCU programming generally looks like Fig. 9.1.

To program a memory location, the following procedure is used:

1. The new word is driven onto RA0-RA3 and RB0-RB7.
2. The “prog” single shot sends a 100 usec programming pulse to the PICmicro® MCU.
3. The data word driver (“driver enable”) is turned off.

### Low-End PICmicro® MCU Programming Pins (Continued)

<table>
<thead>
<tr>
<th>Pins</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0CK1</td>
<td>Program/Verify Clock</td>
</tr>
<tr>
<td>OSC1</td>
<td>Program Counter Input</td>
</tr>
<tr>
<td>_MCLR/Vpp</td>
<td>Programming Power</td>
</tr>
<tr>
<td>V dd</td>
<td>PICmicro® MCU Power</td>
</tr>
<tr>
<td>V ss</td>
<td>Ground (&quot;Gnd&quot;)</td>
</tr>
</tbody>
</table>

Figure 9.1 Low-End PICmicro® MCU Programmer
4. A programming pulse is driven that reads back the word address to confirm the programming was correct. In Fig. 9.1, the read-back latch is loaded on the falling edge of the “on” gate to get the data driven by the PICmicro® MCU.

5. Steps two through four are repeated a maximum of 25 times or until the data stored in the latch are correct.

6. Steps one through four are repeated three times and each time it is required to get the correct data out of the PICmicro® MCU. This is used to ensure the data is programmed in reliably.

7. “OSC1” is pulsed to increment to the next address. This operation also causes the PICmicro® MCU to drive out the data at the current address before incrementing the PICmicro® MCU’s program counter (which happens on the falling edge of OSC1).

In Fig. 9.2, the programming steps one to four listed above are shown along with the latch clock signal. Note

---

**Figure 9.2** Low-End PICmicro® MCU Programming Waveform
that upon power up, the “configuration fuses” are the first address to be accessed, followed by the contents of the program memory.

Just pulsing the “OSC1” pin can be used to implement a “fast verify” as shown in Fig. 9.3. As noted above, each time “OSC1” is pulsed, data at the current address will be output and then increment the PICmicro® MCU’s Program Counter. Figure 9.3 shows the fast verify right from the start with the configuration fuse value output first before the contents of the program memory are output.

The low-end PICmicro® MCU configuration fuses, while Microchip documentation indicates they are at address 0x0FFF, are the first words to be programmed. When programming a low-end PICmicro® MCU, the configuration fuses should be skipped over the first time the PICmicro® MCU is programmed. After doing this, power should be shut off and the PICmicro® MCU put back into programming mode. The reason for programming the configuration fuses last is to make sure the “code protect” bit of the configuration register is not reset (en-

![Diagram of the Fast Verify Waveform]

Figure 9.3 Low-End PICmicro® MCU “Fast Verify” Waveform
able) during program memory programming. If code protection is enabled then data read back will be scrambled during programming, which makes verification of the code impossible.

**Mid-Range Serial Programming**

Serial programming (known as “In Circuit Serial Programming” or “ICSP”) for the low-end (which implements) and mid-range parts consists of pin access:

<table>
<thead>
<tr>
<th>Pin</th>
<th>12C5xx</th>
<th>16C50x</th>
<th>18 Pin</th>
<th>28 Pin</th>
<th>40 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Vpp</td>
<td>_MCLR</td>
<td>_MCLR</td>
<td>_MCLR</td>
<td>_MCLR</td>
<td>_MCLR</td>
</tr>
<tr>
<td>2 Vdd</td>
<td>1 Vdd</td>
<td>1 Vdd</td>
<td>14 Vdd</td>
<td>26 Vdd</td>
<td>11,32</td>
</tr>
<tr>
<td>Vdd</td>
<td>GND</td>
<td>8 Vss</td>
<td>14 Vss</td>
<td>5 Vss</td>
<td>8,21 Vss</td>
</tr>
<tr>
<td>4 DATA</td>
<td>7 GPO</td>
<td>13 RB0</td>
<td>13 RB7</td>
<td>28 RB7</td>
<td>40 RB7</td>
</tr>
</tbody>
</table>

To program and read data, the PICmicro® MCU must be put into “programming mode” by raising the “_MCLR” pin to 13 to 14 volts, and the “data” and “clock” lines pulled low for several milliseconds. Once, the PICmicro® MCU is in programming mode, “Data” can then be shifted in and out using the “Clock” line.

Putting the PICmicro® MCU into programming mode requires the data waveform shown in Fig. 9.4. When _MCLR is driven to Vpp, the internal program counter of the PICmicro® MCU is reset. The PICmicro® MCU’s program counter is used to keep track of the current pro-
Program memory address in the EPROM that is being accessed.

**Mid-Range PICmicro® MCU EPROM ICSP Programming Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Bit Pattern</th>
<th>Data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Data</td>
<td>0b0000010</td>
<td>0, 14 Bits</td>
<td>Load word for programming</td>
</tr>
<tr>
<td>Begin</td>
<td>0b0001000</td>
<td>none</td>
<td>Start Programming</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td>Data, 0</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>0b0001110</td>
<td>none</td>
<td>End Programming</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increment</td>
<td>0b0000110</td>
<td>none</td>
<td>Increment the PICmicro® MCU's Program Counter</td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Data</td>
<td>0b0001000</td>
<td>0, 14 Bits</td>
<td>Read Program Memory at Program Counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data, 0</td>
<td></td>
</tr>
<tr>
<td>Load Config</td>
<td>0b00000000</td>
<td>0x07FF</td>
<td>Set the PICmicro® MCU's</td>
</tr>
</tbody>
</table>
Data is shifted in and out of the PICmicro® MCU using a synchronous protocol. Data is shifted out least significant bit (LSB) first on the falling edge of the clock line. The minimum period for the clock is 200 nsecs with the data bit centered as shown in Fig. 9.5, which is sending an “increment address” command.

When data is to be transferred, the same protocol is used, but a 16-bit transfer (LSB first) follows after one microsecond has passed since the transmission of the command. The 16 bits consist of the instruction word shifted to the left by one. This means the first and last bits of the data transfer are always “zero”.

Before programming can start, the program memory should be checked to make sure it is blank. This is accomplished by simply reading the program memory (“Read Data” command listed above) and comparing the data returned to 0x07FFE. After every compare, the PICmicro® MCU’s program counter is incremented (using the “increment address” command) to the size of the devices program memory. Once the program memory is checked, the PICmicro® MCU’s program counter is “jumped” to 0x02000 (using the “Load Configuration” command).
and then the next eight words are checked for 0x07FFE.

For different mid-range devices, the following table of PICmicro® MCU part numbers can be used to determine the amount of program memory available within them:

<table>
<thead>
<tr>
<th>Device</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16Cx1</td>
<td>1k</td>
</tr>
<tr>
<td>PIC16Cxx0</td>
<td>0.5k</td>
</tr>
<tr>
<td>PIC16Cxx1</td>
<td>1k</td>
</tr>
<tr>
<td>PIC16Cxx2</td>
<td>2k</td>
</tr>
<tr>
<td>PIC16Cx2</td>
<td>2k</td>
</tr>
<tr>
<td>PIC16Cx3</td>
<td>4k</td>
</tr>
<tr>
<td>PIC16Cx4</td>
<td>2k</td>
</tr>
<tr>
<td>PIC16Cx5</td>
<td>4k</td>
</tr>
<tr>
<td>PIC16Cx6</td>
<td>8k</td>
</tr>
<tr>
<td>PIC16Cx7</td>
<td>8k</td>
</tr>
</tbody>
</table>

The process for programming an instruction word in a mid-range EPROM-based PICmicro® MCU is

1. A “Load Data” command with the word value is written to the PICmicro® MCU.
2. A “Begin Programming” command is written to the PICmicro® MCU.
3. Wait 100 msecs.
4. An “End Programming” command is written to the PICmicro® MCU.
5. A “Read Data” command is sent to the PICmicro® MCU and 14 bits (the LSB and MSB of the 16-bit
data transferred are ignored) in the program memory instruction are read back.

6. Steps one through five are repeated up to 25 times until the data read back is correct. Steps one through five are known as a “Programming Cycle”.

7. Steps one through four are repeated three times the number of cycles required to get a valid instruction word read. This is known as “Over-programming”.

8. The PICmicro® MCU’s Program Counter is incremented using the “Increment Address” command.

9. Steps one through eight are repeated for the entire application to the configuration fuses.

10. A “Load Config” command is sent to the PICmicro® MCU to set the Program Counter to 0x02000.

11. The PICmicro® MCU’s configuration fuses are programmed using the “Programming Cycle” detailed in steps one through seven.

The process for programming program memory could be blocked out with the pseudo-code:

```c
ICSPProgran()     // Program to be burned in is in
                 // an array of
                 // addresses and data

int PC = 0;       // PICmicro® MCU’s program counter
int i, j k;
int retval = 0;

for (i = 0; (i < PGMsize) & (retval == 0); i++)
{
    if (PC != address(i)) {
```
if ((address[i] >= 0x02000) && (PC < 0x02000))
{
    LoadConfiguration(0x07FFE);
    PC = 0x02000;
}

for (; PC < address[i]; PC++)
    IncrementAddress();

for (i = 0; (i < 25) && (retvalue != data[i]); i++)
{
    LoadData(ins[i] << 1); // Programming Cycle
    BeginProgramming();
    Delay(100usec);
    EndProgramming();
    Retvalue = ReadData();
}

if (i == 25)
    retvalue = -1; // Programming Error
else
{
    retvalue = 0; // Okay, Repeat Programming
    // Cycle 3x
    for (k = 0; k < (j * 3); k++)
    {
        LoadData(ins[i] << 1);
        BeginProgramming();
        Delay(100usec);
        EndProgramming();
    } // endif
} // endif
} // endif
} // end ICSPProgram

After the program memory has been loaded with the application code, Vpp should be cycled off and on and the PICmicro® MCUs program memory is read out and compared against the expected contents. When this verify is executed, Vpp should be cycled again with Vdd a minimum voltage (4.5 volts) and then repeated again with Vdd at a maximum voltage (5.0 volts) value. This second
verify is used by “Production PICmicro® MCU Programmers”. Programmers not having these minimum/maximum Vdd verify steps are known as “Prototype PICmicro® MCU Programmers”.

The PIC12C50x and PIC16C50x low-end parts are programmed using a similar protocol as the EPROM based mid-range PICmicro® MCUs. The command first enters programming mode (with the data and clock pins pulled low followed by _MCLR driven to +13 volts) and the PICmicro® MCU’s program counter is set to 0xFFF, which is the configuration register address.

The PIC12C50x and PIC16C505 use a 12-bit instruction word. When data is passed to the PICmicro® MCU, the upper 3 bits (instead of the upper one) are “zero” and ignored by the device as it is programmed. The first bit sent is still “0”, with the LED of the instruction word following.

A simple way of calculating the 16 data bits to be programmed into the PIC12C50x and PIC16C505 microcontrollers from the instruction is to save the instruction in a 16 bit variable and shift it “up” (to the “left”) by one bit. The commands available for programming the PIC12C50x and PIC16C505 have a 6-bit header and optional 16-bit instruction or configuration fuse data word.

**PIC12C5xx and PIC16C505 Programming Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load data</td>
<td>000010 +0, data{12}, 000</td>
</tr>
<tr>
<td>Read data</td>
<td>000100 +0, data{12}, 000</td>
</tr>
<tr>
<td>Increment PC</td>
<td>001000</td>
</tr>
</tbody>
</table>
Microchip uses a modified version of the programming algorithm outlined above for the 16F8x “Flash”-based parts. Along with the programming algorithm being much simpler, the actual programming circuit is much easier to implement in hardware. Figure 9.6 shows a typical Flash programming circuit.

Electrically, the programming voltages are basically the same as what is required for the mid-range devices. There is one difference, however, and that is in the voltage and current required for Vpp. For the mid-range parts, up to 50 mA is required for EPROM programming.

Because the 16F8x parts have built-in EPROM data and Flash VPP generator, this circuit will provide the actual voltages and currents to program and engage the data and program memory resulting in micro-Amperes of current required from the programmer in “Programming Mode”.

The same data packet format is used for the 16F8x as was used for the mid-range EPROM parts, but the com-
The data, like in the mid-range EPROM part, is always 16 bits with the first and last bit always equal to zero. Data is always transferred LSB first using the same timings as specified earlier in the chapter for the mid-range parts. When transferring 14 bits of data from the hex file instruction word, it can be multiplied by two, leaving the LSB and MSB reset.

The programming cycle for the PIC 16F8x is as follows:

1. “Load Data for Program Memory” command with Instruction word.
2. Send “Begin Programming” command.
3. Wait 10 msecs.
4. “Read Data from Program Memory” command and verify the contents of the Program Memory.
5. Send “Increment PICmicro® MCU’s Program Counter” command.
6. Steps one through five are the Flash PICmicro® MCU “Programming Cycle”. These steps are repeated for every instruction in the hex file.
7. A “Load Configuration” command is sent to set the Program Counter to point to address 0x02000.
8. Steps one through four are repeated for the Configuration Information.

The table below lists the different commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Bits</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>000000</td>
<td>07FFE In</td>
</tr>
<tr>
<td>Load Data for Program</td>
<td>000010</td>
<td>Word x 2 Going In</td>
</tr>
</tbody>
</table>

The data, like in the mid-range EPROM part, is always 16 bits with the first and last bit always equal to zero. Data is always transferred LSB first using the same timings as specified earlier in the chapter for the mid-range parts. When transferring 14 bits of data from the hex file instruction word, it can be multiplied by two, leaving the LSB and MSB reset.

The programming cycle for the PIC 16F8x is as follows:

1. “Load Data for Program Memory” command with Instruction word.
2. Send “Begin Programming” command.
3. Wait 10 msecs.
4. “Read Data from Program Memory” command and verify the contents of the Program Memory.
5. Send “Increment PICmicro® MCU’s Program Counter” command.
6. Steps one through five are the Flash PICmicro® MCU “Programming Cycle”. These steps are repeated for every instruction in the hex file.
7. A “Load Configuration” command is sent to set the Program Counter to point to address 0x02000.
8. Steps one through four are repeated for the Configuration Information.
To erase the Flash Program Memory, use the Microchip specified erase for code protected devices. This operation will erase all Flash and EEPROM memory in the PICmicro® MCU device, even if code protection is enabled.

1. Hold RB6 and RB7 low and apply Vpp, wait at least 2 msecs.
2. Execute Load Configuration (0b0000000+ 0x07FFE).
3. Increment the PC to the Configuration Register Word (Send 0b0000110 seven times).
4. Send command 0b0000001 to the PICmicro® MCU.
5. Send command 0b0000111 to the PICmicro® MCU.
6. Send “begin programming” (0b0001000) to the PICmicro® MCU.
7. Wait 10 ms.
8. Send command 0b0000001.
9. Send command 0b0000111.

Note that there are two undocumented commands (“0b0000001” and “0b0000111”) in this sequence.

**PIC17Cxx Programming**

The PIC17Cxx is connected to a programmer using the wiring shown in Fig. 9.7. Note that PORTB and PORTC are used for transferring data 16 bits at a time and PORTA is used for the control bits that control the operation of the programmer. The “_MCLR” pin is pulled high to 13 volts as would be expected to put the PICmicro® MCU into “Programming Mode”.

While the programming of the PIC17Cxx is described as being in “parallel”, a special “Boot ROM” routine executes within the PICmicro® MCU and this accepts data from the I/O ports and programs the code into the PICmicro® MCU. To help facilitate this, the “test” line, which is normally tied low, is pulled high during application execution to make sure that the programming functions can be accessed. The clock, which can be any value from 4 MHz to 10 MHz is used to execute the “Boot ROM” code for the programming operations to execute.

To put the PICmicro® MCU into programming mode, the “test” line is made active before _MCLR is pulled to Vpp and then 0x0E1 is driven on PORTB to command the boot code to enter the programmer routine. This sequence is shown in Fig. 9.8. To end programming mode, _MCLR must be pulled to ground 10 msecs or more before power is taken away from the PICmicro® MCU. “Test” should be de-asserted after _MCLR is pulled low.

When programming, the RA0 pin is pulsed high for at
least 10 instruction cycles (10 us for the PICmicro® MCU running at 4 MHz) to load in the instruction address followed by the PICmicro® MCU latching out the data (so that it can be verified). After the data have been verified, RA0 is pulsed high for 100 usecs to program the data. If RA1 is low during the RA0 pulse, then the PICmicro® MCU program counter will be incremented. If it goes high during the pulse, the internal program counter will not be incremented and the instruction word contents can be read back in the next RA1 cycles without having to load in a new address.

The latter operation is preferred and looks like the waveforms shown in Fig. 9.9. This waveform should be repeated until the data is loaded or up to 25 times. Once it is programmed in, then three times the number of programming cycles must be used to “lock” and “overprogram” the data in. This process is similar to that of the other EPROM parts.
Writing to the specified addresses between 0x0FE00 and 0x0FE0F programs and verifies the configuration word. To program ("make 0") one of the configuration bits, write to its register. Reading back the configuration word uses the first three RA1 cycles of Fig. 9.9 at either 0x0FE00 or 0x0FE08. Reading 0x0FE00 will return the low byte of the configuration word in PORTC (0x0FF will be in PORTB) and reading 0x0FE08 will return the high byte in PORTC.

The configuration bits for the PIC17Cxx are defined as:

<table>
<thead>
<tr>
<th>PIC17Cxx Configuration Bits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Bit</td>
</tr>
<tr>
<td>FOSC0</td>
<td>0x0FE00</td>
</tr>
<tr>
<td>FOSC1</td>
<td>0x0FE01</td>
</tr>
<tr>
<td>WDTPS0</td>
<td>0x0FE03</td>
</tr>
<tr>
<td>WDTPS1</td>
<td>0x0FE04</td>
</tr>
<tr>
<td>PM0</td>
<td>0x0FE05</td>
</tr>
<tr>
<td>PM1</td>
<td>0x0FE06</td>
</tr>
<tr>
<td>PM2</td>
<td>0x0FE0F</td>
</tr>
</tbody>
</table>
where the bits are defined as:

<table>
<thead>
<tr>
<th>PIC17Cxx Configuration Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM2:PM0</td>
</tr>
<tr>
<td>111</td>
</tr>
<tr>
<td>110</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>000</td>
</tr>
<tr>
<td>WDTPS1:WDTPS0</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>FOSC1:FOSC0</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>00</td>
</tr>
</tbody>
</table>

Note, configuration bit addresses must be written to in ascending order. Programming the bit in non-register ascending order can result in unpredictable programming of the configuration word as the “Processor” Mode changes to a “Code Protected” mode before the data is loaded in completely.

**PIC17Cxx ICSP Programming**

The capability of a PIC17Cxx application to write to program memory is enabled when the _MCLR is driven by more than 13 volts and a tablwt instruction is executed. When tablwt is executed, the data loaded into the TABLATH and TABLATL registers is programmed into
the memory locations. This instruction keeps executing (it does not complete after two cycles, as it would if the
TBLPRH and TBLPTRL registers were pointing outside the internal EPROM) until it is terminated by an inter-
rupt request or _MCLR reset.

To perform a word write, the following mainline process would be used:

1. Disable TMR0 interrupts.
2. Load TABPTRH and TABPTRL with the address.
3. Load TABLATL or TABLATL with the data to be stored.
4. Enable a 1,000 usec TMR0 delay interrupt (initialize TMR0 and enable TMR0 interrupt).
5. Execute tablwt instruction with the missing half of data.
6. Disable TMR0 interrupts.
7. Read back data—Check for match.
8. If no match—Return error.

The interrupt handler for this process can just be executing a “retfie” instruction. Sample code for writing to
the PIC17Cxx’s program memory is

```
org    0x00010
TMR0Int                       ; Timer Interrupt
; Request
; Acknowledge
retfie
;
movfp   SaveAddress,        ; Point to the
        TBLPTRL            ; Memory being
```
400  Chapter 9

    movfp  SaveAddress + 1,  ; written to
           TBLPTRH

    bcf    PORTA, 3           ; Turn on Programming
           ; Voltage

    movlw  HIGH ((100000 / 5)  ; Delay 100 msecs
           + 256)             ; for
    movwf   Dlay                ; Programming Voltage
                   ; to Stabilize
    movlw  LOW ((100000 / 5)
           + 256)
    addlw   0x0FF
    btfsc   ALUSTA, Z
    decfsz  Dlay, f
    goto $ - 3

    movlw  HIGH (65536 -       ; Delay 10 msecs for
           10000)             ; EPROM Write
    movwf   TMR0H
    movlw  LOW (65536 - 10000)
    movwf   TMR0L
    bsf     T0STA, T0CS         ; Start up the Timer
    movlw   1 << T0IE           ; Enable Interrupts
    movwf   INTSTA
    bcf     CPUSTA, GLINTD
    tlwt    0, SaveData         ; Load Table Pointer
           ; with Data
    tlwt    1, SaveData + 1
    tablw   1, 0, SaveData + 1  ; Write the Data In
    nop
    nop
    clrwf   INTSTA, f            ; Turn Off Interrupts
    bsf     CPUSTA, GLINTD
    movlw   2
    call    SendMSG
    bcf     PORTA, 3
To enable internal programming, _MCLR has to be "switched" from 5 volts (VDD) to 13 volts. The Microchip circuit that is recommended is shown in Fig. 9.10. This circuit will drive the PIC17Cxx's _MCLR pin at 5 volts until "RA2" is pulled low. When RA2 is pulled low, the voltage driven in to _MCLR will become 13 volts (or "Vpp"). The programming current at 13 volts is a minimum of 30 mA.

The “boot code” is a host interface application that reads data and then programs it at the specified address. This code must be burned into the PICmicro® MCU before "ICSP" can execute. The typical “boot code” for a PIC17Cxx PICmicro® MCU would be as follows:

1. Establish communication with programming host.
2. If no communication link is established jump to application code.

![Figure 9.10](Image)

Figure 9.10  PIC17Cxxx in circuit serial programming schematic
3. Enable Vpp (RA2 = 0).
4. Wait for host to send instruction word address.
5. Program in the word.
6. Confirm word is programmed correctly.
7. Loop back to four.

**PIC18Cxx Programming**

Like the PIC17Cxx, the PIC18Cxx has the capability to “self program” using the table “read and write” instructions. In the PIC18Cxx, this capability is not only available within applications, but is used to program an erased device.

To program the PIC18Cxx, instructions are downloaded into the PICmicro® MCU after setting the “_MCLR” pin to Vpp (13 to 14 volts, as in the other EPROM PICmicro® MCUs) with both RB6 and RB7 low. Passing instructions (which contain the program data) to the PICmicro® MCU is accomplished by first sending a 4-bit “Special Instruction” followed by an optional 16-bit instruction. The 4-bit Special Instruction is sent most significant bit first and can either specify that an instruction follows or that it is a “mnemonic” for a “TBLRD” or “TBLWT” instruction as shown in the table below:

<table>
<thead>
<tr>
<th>Special Instruction</th>
<th>Mnemonic</th>
<th>Instruction Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>nop</td>
<td>Shift in Next Instruction</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>TBLRD</td>
<td>Read Table</td>
<td>2</td>
</tr>
<tr>
<td>1001</td>
<td>TBLRD +</td>
<td>Read Table, Increment TRLPTR</td>
<td>2</td>
</tr>
</tbody>
</table>
The data transmission looks like Fig. 9.11. The four “nop” bit code is transmitted first followed by the 16-bit instruction.

If the instruction is a table operation, then the “Special Instruction” code can be used instead of the “nop” to simplify the data transfer. At the end of Fig. 9.11, the bit pattern 0b01101 (“TBLWT *”) is sent to the PICmicro\textsuperscript{®} MCU.

<table>
<thead>
<tr>
<th>Special Instruction</th>
<th>Mnemonic</th>
<th>Instruction Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>TBLRD *-</td>
<td>Read Table, Decrement TBLPTR</td>
<td>2</td>
</tr>
<tr>
<td>1011</td>
<td>TBLRD +*</td>
<td>Increment TBLPTR, Read Table</td>
<td>2</td>
</tr>
<tr>
<td>1100</td>
<td>TBLWT *</td>
<td>Write Table</td>
<td>2</td>
</tr>
<tr>
<td>1101</td>
<td>TBLWT +*</td>
<td>Write Table, Increment TBLPTR</td>
<td>2</td>
</tr>
<tr>
<td>1110</td>
<td>TBLWT *-</td>
<td>Write Table, Decrement TBLPTR</td>
<td>2</td>
</tr>
<tr>
<td>1111</td>
<td>TBLWT +*</td>
<td>Increment TBLPTR, Write Table</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 9.11 Serial Instruction Timing for 1 Cycle 16-Bit Instructions
While the table reads and writes only require 4 bits, to carry out the program operation, there are always 16 bits following the mnemonic (just as if it were a “nop”) for data transfer and this avoids the need for explicitly loading and unloading the table latch registers using instructions. In Fig. 9.12, the “tblwt” instruction (write to table and do not change TBLPTR) is shown.

After the first 20-bit sequence, a second 20-bit sequence is executed to allow the programming operation to complete (this is what is meant by the “2” in the “Cycles” in the table above). The PICmicro® MCU ignores the second sequence of 20 bits and the initial sequence is processed. Reading data from the PICmicro® MCU’s program memory is accomplished in exactly the same way.

To set up a table read or write, first the TBLPTR has to be initialized. This is done using standard “movlw” and “movwf” instruction. For example, to program address
0x012345 with 0x06789, the data sequence is written to the PIC18Cxx:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction/Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>movlw UPPER 0x012345</td>
</tr>
<tr>
<td>nop</td>
<td>movwf TBLPTRU</td>
</tr>
<tr>
<td>nop</td>
<td>movlw HIGH 0x012345</td>
</tr>
<tr>
<td>nop</td>
<td>movwf TBLPTRH</td>
</tr>
<tr>
<td>nop</td>
<td>movlw LOW 0x012345</td>
</tr>
<tr>
<td>nop</td>
<td>movwf TBLPTRL</td>
</tr>
<tr>
<td>tblwt *</td>
<td>0x06789</td>
</tr>
</tbody>
</table>

**Microchip ICSP Programming Connector**

The “ICSP” Programming Connector defined by Microchip uses the pinout shown in the table below:

<table>
<thead>
<tr>
<th>Microchip “ICSP” Pin Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIN</td>
</tr>
<tr>
<td>1.Vpp</td>
</tr>
<tr>
<td>2.Vdd</td>
</tr>
<tr>
<td>3.Vss</td>
</tr>
<tr>
<td>4.DATA</td>
</tr>
<tr>
<td>5.CLOCK</td>
</tr>
</tbody>
</table>

To connect a PICmicro® MCU, which has been put into an application circuit, the following interface shown in Fig. 9.13 should be used.

The PICmicro® MCU must be isolatable from the application circuit. The diode on the “MCLR/Vpp” pin and the “breakable connections” on Vdd, RB7, and RB6 isolate the PICmicro® MCU. These “breaks” are best provided by unsoldered “zero ohm” resistors or uncon-
connected jumpers in the circuit. This has to be done because the ICSP specification will only provide 50 mA for Vdd and has 1K ohm resistors in the data and clock lines to protect the driver circuits.

Third Party/Downloadable Programmers

When considering a PICmicro® MCU programmer, the following questions should be asked:

1. What are the supported PICmicro® MCU devices?
2. What is the interface and how is the application timed?
3. How are the Configuration Fuses programmed?
4. What Operating System does it run under?
Chapter 10

PC Interfaces

Memory Map
Figure 10.1 shows a graphic of the PC’s memory map.

I/O Space Map
Only the lower 10 bits of the I/O space have been specified for the basic PC/AT register operation. Some PC/XT specific registers have been omitted from this list. It is not obvious in the table below, but I/O port addresses 0x0000 to 0x00FF are on the motherboard while the addresses above are on adapter cards.

For some motherboards, registers are accessed at ad-
addresses 0x0400 and above. To avoid problems, make sure that you only specify addresses below 0x0400.

<table>
<thead>
<tr>
<th>Address</th>
<th>Register Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000h</td>
<td>DMA channel 0 address (low addressed first, then high)</td>
</tr>
<tr>
<td>0001h</td>
<td>DMA channel 0 word count (low addressed first, then high)</td>
</tr>
<tr>
<td>0002h</td>
<td>DMA channel 1 address (low addressed first, then high)</td>
</tr>
<tr>
<td>0003h</td>
<td>DMA channel 1 word count (low addressed first, then high)</td>
</tr>
<tr>
<td>0004h</td>
<td>DMA channel 2 address (low addressed first, then high)</td>
</tr>
<tr>
<td>0005h</td>
<td>DMA channel 2 word count (low addressed first, then high)</td>
</tr>
<tr>
<td>0006h</td>
<td>DMA channel 3 address (low addressed first, then high)</td>
</tr>
<tr>
<td>0007h</td>
<td>DMA channel 3 word count (low addressed first, then high)</td>
</tr>
<tr>
<td>0008h</td>
<td>Read - DMA 1 channel 0-3 status register</td>
</tr>
</tbody>
</table>
Bit 7 = Channel 3 Request
Bit 6 = Channel 2 Request
Bit 5 = Channel 1 Request
Bit 4 = Channel 0 Request
Bit 3 = Channel Terminal Count on Channel 3
Bit 2 = Channel Terminal Count on Channel 2
Bit 1 = Channel Terminal Count on Channel 1
Bit 0 = Channel Terminal Count on Channel 0

Write - DMA 1 channel 0-3 command register
Bit 7 = DACK Sense Active High
Bit 6 = DREQ Sense Active High
Bit 5 = Extended Write Selection
Bit 4 = Rotating Priority
Bit 3 = Compressed Timing
Bit 2 = Enable Controller

0009h DMA 1 write request register
000Ah DMA 1 channel 0-3 mask register
Bit 7-3 = Reserved
Bit 2 = Mask bit
Bit 1-0 = Channel Select
- 00 channel 0
- 01 channel 1
- 10 channel 2
- 11 channel 3

000Bh DMA 1 channel 0-3 mode register
Bit 7-6 = Operating Mode
- 00 demand mode
- 01 single mode
- 10 block mode
- 11 cascade mode
Bit 5 = address increment select
Bit 3-2 = Operation
- 00 verify operation
- 01 write to memory
- 10 read from memory
- 11 reserved
Bit 1-0 = Channel Select
- 00 channel 0
- 01 channel 1
- 10 channel 2
- 11 channel 3
410  Chapter 10

000Ch    DMA 1 clear byte pointer flip-flop
000Dh    Read - DMA 1 read temporary register
          Write - DMA 1 master clear
000Eh    DMA 1 clear mask register
000Fh    DMA 1 write mask register
0020h    Interrupt Controller 1 initialization command word
          Bit 7-5 = 0 - only used in 80/85 mode
          Bit 4 = ICW1 Request
          Bit 3 = Interrupt Request Mode
          - 0 Edge triggered mode
          - 1 Level triggered mode
          Bit 2 = Interrupt Vector Size
          - 0 Eight Byte Interrupt Vectors
          - 1 Four Byte Interrupt Vectors
          Bit 1 = Operating Mode
          - 0 Cascade mode
          - 1 Single mode
          Bit 0 = IC4 Requirements
          - 0 not needed
          - 1 needed
0021h    Interrupt Controller 1 Interrupt Mask Register
          bit 7 = 0  enable parallel printer
          bit 6 = 0  enable diskette interrupt
          bit 5 = 0  enable fixed disk interrupt
          bit 4 = 0  enable serial port 1
          bit 3 = 0  enable serial port 2
          bit 2 = 0  enable video interrupt
          bit 1 = 0  enable keyboard, mouse, RTC
          bit 0 = 0  enable timer interrupt
0040h    8254 Timer Counter 0 & Counter Divisor Register
0041h    8254 Timer Counter 1 & Counter Divisor Register
0042h    8254 Timer Counter 2 & Counter Divisor Register
0043h    8254 Timer Mode/Control port
          Bit 7-6 = Counter Select
          - 00 Counter 0
          - 01 Counter 1
          - 10 Counter 2
Bit 5-4 = Counter Read/Write Operation
- 01 Read/Write Low Counter Byte
- 10 Read/Write High Counter Byte
- 11 Read/Write Low, then High Counter Bytes

Bit 3-1 = Counter Mode Select
- 000 mode 0
- 001 mode 1/Programmable One Shot
- x10 mode 2/Rate Generator
- x11 mode 3/Square Wave Generator
- 100 mode 4/Software Triggered Strobe
- 101 mode 5/Hardware Triggered Strobe

Bit 0   = Counter Type
- 0 Binary Counter
- 1 BCD Counter

0060h        Read - Keyboard Controller
Bit 7   = Keyboard Inhibit (Reset)
Bit 6   = CGA (Reset)
Bit 5   = Manufacturing Jumper Install
Bit 4   = Reset if System RAM 512K
Bit 3-0 = Reserved

Write - Keyboard Controller
Bit 7   = Keyboard Data Output
Bit 6   = Keyboard Clock Output
Bit 5   = Input Buffer Full (Reset)
Bit 4   = Output Buffer Empty (Reset)
Bit 3-2 = Reserved
Bit 1   = Address Line 20 Gate
Bit 0   = System Reset

0061h        Read - Keyboard Controller Port B control register
Bit 7   = Parity Check
Bit 6   = Channel Check
Bit 5   = Current Timer 2 Output
Bit 4   = Toggles with each Refresh Request
Bit 3   = Channel Check Status
Bit 2   = Parity Check Status
Bit 1   = Speaker Data Status
Bit 0   = Timer 2 Gate to Speaker Status
412 Chapter 10

Write - 8255 Compatible Port
Bit 7 = Clear Keyboard
Bit 6 = Hold Keyboard Clock Low
Bit 5 = I/O Check Enable
Bit 4 = RAM Parity Check Enable
Bit 3 = Read low/high switches
Bit 2 = Reserved
Bit 1 = Speaker Clock Enable
Bit 0 = Timer 2 Gate to Speaker Enable

0064h Read - Keyboard Controller Status
Bit 7 = Parity Error on Keyboard Transmission
Bit 6 = Receive Timeout
Bit 5 = Transmit Timeout
Bit 4 = Keyboard Inhibit
Bit 3 = Input Register Type
- 1 data in input register is command
- 0 data in input register is data
Bit 2 = System Flag Status
Bit 1 = Input Buffer Status
Bit 0 = Output Buffer Status

Write Keyboard Controller Input Buffer
20 = Read Byte Zero of Internal RAM, this is the last KB command send to 8042
21-3F = Reads the Byte Specified in the Lower 5 Bits of the command in the 8042's internal RAM
60-7F = Writes the Data Byte to the Address Specified in the 5 Lower Bits of the Command

0065h Address Line 20 Gate Control
Bit 2 = A20 gate control
1 = A20 enabled
0 = A20 disabled

0070h CMOS RAM index register port
Bit 7 = NMI Enable
Bit 6-0 = CMOS RAM Index

0071h CMOS RAM data port
00 = Current Second in BCD
01 = Alarm Second in BCD
02 = Current Minute in BCD
03 = Alarm Minute in BCD
04 = Current Hour in BCD
05 = Alarm Hour in BCD
06 = Day of Week in BCD
07 = Day of Month in BCD
08 = Month in BCD
09 = Year in BCD (00-99)
0A = Status Register A
   Bit 7 = Update Progress
   Bit 6-4 = Divider that Identifies the time-based Frequency
   Bit 3-0 = Rate Selection Output
0B = Status Register B
   Bit 7 = Run/Halt Control
   Bit 6 = Periodic Interrupt Enable
   Bit 5 = Alarm Interrupt Enable
   Bit 4 = Update-Ended Interrupt Enable
   Bit 3 = Square Wave Interrupt Enable
   Bit 2 = Calendar Format
   Bit 1 = Hour Mode
   Bit 0 = Daylight Savings time Enable
0C = Status Register C
   Bit 7 = Interrupt Request Flag
   Bit 6 = Periodic Interrupt Flag
   Bit 5 = Alarm Interrupt Flag
   Bit 4 = Update Interrupt Flag
   Bit 3-0 = Reserved
0D = Status Register D
   Bit 7 = Real-Time Clock power

0080h "MFG_PORT" Write Address
0080h DMA Page Register page register (temporary storage)
0081h DMA Channel 2 Page Address
0082h DMA Channel 3 Page Address
0083h DMA Channel 1 Page Address
0084h Extra Page Register
0085h Extra Page Register
0086h Extra Page Register
0087h DMA Channel 0 Page Address
0088h Extra Page Register
0089h DMA Channel 6 Page Address
008Ah DMA Channel 7 Page Address
008Bh DMA Channel 5 Page Address
008Ch Extra Page Register
008Dh Extra Page Register
008Eh Extra Page Register
008Fh DMA Refresh Page Register
00A0h Interrupt Controller 2 Initialization
Command Word
Bit 7-5 = 0 - only used in 80/85 mode
Bit 4 = ICW1 Request
Bit 3 = Interrupt Request Mode
  - 0 Edge triggered mode
  - 1 Level triggered mode
Bit 2 = Interrupt Vector Size
  - 0 Eight Byte Interrupt Vectors
  - 1 Four Byte Interrupt Vectors
Bit 1 = Operating Mode
  - 0 Cascade mode
  - 1 Single mode
Bit 0 = IC4 Requirements
  - 0 not needed
  - 1 needed

00A1h Interrupt Controller 2 Mask Register
Bit 7 = Reserved
Bit 6 = Fixed Disk
Bit 5 = Coprocessor exception
Bit 4 = Mouse Interrupt
Bit 3 = Reserved
Bit 2 = Reserved
Bit 1 = Redirect Cascade
Bit 0 = Real-Time Clock

00D0h Read - DMA Controller 2 Channel 4-7
status register
Bit 7 = Channel 7 Request
Bit 6 = Channel 6 Request
Bit 5 = Channel 5 Request
Bit 4 = Channel 4 Request
Bit 3 = Channel 7 Terminal Count
Bit 2 = Channel 6 Terminal Count
Bit 1 = Channel 5 Terminal Count
Bit 0 = Channel 4 Terminal Count

Write DMA Controller 2 Channel 4-7
command register
Bit 7 = DACK Sense Active High
Bit 6 = DRQ Sense Active High
Bit 5 = Extended Write Selection
Bit 4 = Rotating Priority
Bit 3 = Compressed Timing
Bit 2 = Enable Controller

00D2h DMA Controller 2 Channel 4-7 Write
Request Register
PC Interfaces

00D4h        DMA Controller 2 Channel 4-7 Write
Single Mask Register
Bit 7-3 = Reserved
Bit 2 = Mask bit
Bit 1-0 = Channel Select
- 00 channel 0
- 01 channel 1
- 10 channel 2
- 11 channel 3

00D6h        DMA Controller 2 Channel 4-7 Mode
Register
Bit 7-6 = Operating Mode
- 00 demand mode
- 01 single mode
- 10 block mode
- 11 cascade mode
Bit 5 = address increment select
Bit 3-2 = Operation
- 00 verify operation
- 01 write to memory
- 10 read from memory
- 11 reserved
Bit 1-0 = Channel Select
- 00 channel 0
- 01 channel 1
- 10 channel 2
- 11 channel 3

00D8h        DMA Controller 2 Channel 4-7 Clear Byte
Pointer

00D9h        Read - DMA Controller Channel 4-7 Read
Temporary Register
Write - DMA Controller Channel 4-7
Master Clear

00DAh        DMA Controller 2 Channel 4-7 Clear Mask
Register

00DBh        DMA Controller 2 Channel 4-7 Write Mask
Register

00F0h        Math Coprocessor Clear Busy Latch
00F1h        Math Coprocessor Reset
00F8h        Opcode Transfer Register
00FAh        Opcode Transfer Register
00FCh        Opcode Transfer Register

01F0h        Hard Disk Controller Data Register
01F1h        Hard Disk Controller Error Register
Bit 7 = Failing Drive
Bit 6-3 = Reserved
416  Chapter 10

Bit 2-0 = Status
  - 001 No Error
  - 010 Formatter Device Error
  - 011 Sector Buffer Error
  - 100 ECC Circuitry Error
  - 101 Controlling Microprocessor Error

01F2h  Sector Count
01F3h  Sector Number
01F4h  Cylinder Low
01F5h  Cylinder High
01F6h  Drive/Head
01F7h  Read - Hard Disk Controller Status Register
      bit 7 = Controller Execution Status
      bit 6 = Drive Status
      bit 5 = Write Fault
      bit 4 = Seek Complete
      bit 3 = Sector Buffer Requires Servicing
      bit 2 = Disk Data Read Successfully Corrected
      bit 1 = Index
      bit 0 = Previous Command Ended in Error

Write - Hard Disk Controller Command Register

0201h  Read - Joystick Position and Status
      Bit 7 = Status B Joystick Button 2
      Bit 6 = Status B Joystick Button 1
      Bit 5 = Status A Joystick Button 2
      Bit 4 = Status A Joystick Button 1
      Bit 3 = B joystick Y coordinate
      Bit 2 = B joystick X coordinate
      Bit 1 = A joystick Y coordinate
      Bit 0 = A joystick X coordinate

Write - Fire Joystick’s four one-shots

0220h  SoundBlaster - Left speaker
      Status/Address
      Address:
      01 = Enable waveform control
      02 = Timer #1 data
      03 = Timer #2 data
      04 = Timer control flags
      08 = Speech synthesis mode
20-35 = Amplitude Modulation/Vibrato
40-55 = Level key scaling/Total level
60-75 = Attack/Decay rate
80-95 = Sustain/Release rate
A0-B8 = Octave/Frequency Number
C0-C8 = Feedback/Algorithm
E0-F5 = Waveform Selection

Address:
01 = Enable waveform control
02 = Timer #1 data
03 = Timer #2 data
04 = Timer control flags
08 = Speech synthesis mode
20-35 = Amplitude Modulation/Vibrato
40-55 = Level key scaling/Total level
60-75 = Attack/Decay rate
80-95 = Sustain/Release rate
A0-B8 = Octave/Frequency Number
C0-C8 = Feedback/Algorithm
E0-F5 = Waveform Selection

PC Interfaces

0221h  SoundBlaster - Left speaker Data
0222h  SoundBlaster - Right speaker/Address
0223h  Right speaker - Data port
0278h  LPT2 data port
0279h  LPT2 Status Port
       Bit 7  = Busy
       Bit 6  = Acknowledge
       Bit 5  = Out of Paper
       Bit 4  = Printer Selected
       Bit 3  = Error
       Bit 2  = IRQ Occurred
       Bit 1-0 = Reserved
027Ah  LPT2 Control Port
       Bit 7-6 = Reserved
       Bit 5  = Data Output Control
       Bit 4  = IRQ Enable
       Bit 3  = Select Printer
       Bit 2  = Initialize
       Bit 1  = Line Feed
       Bit 0  = Strobe
02E8h  8514/A Display Status
02E8h  8514/A Horizontal Total
02EAh  8514/A DAC Mask
02EBh  8514/A DAC Read Index
02ECb  8514/A DAC Write Index
418 Chapter 10

02EDh  8514/A DAC Data
02F8h  Serial Port 3 Transmitter/Receiver
        registers/Divisor Latch Low
02F9h  Serial Port 2 Interrupt Enable
        Register/Divisor Latch High
02FAh  Serial Port 2 Interrupt Identification
        Register
02FBh  Serial Port 2 Line Control Register
02FDh  Serial Port 2 Line Status Register
02FFh  Serial Port 2 Scratchpad Register
0300h-031Fh  IBM Prototype Card Addresses
0360h-036Fh  Network Cards
0370h  Secondary Diskette Controller Status A
0371h  Secondary Diskette Controller Status B
0372h  Secondary Diskette Controller Digital
        Output Register
0374h  Read - Secondary Diskette Controller
        Main Status Register
        Secondary Diskette Controller Data Rate
        Select Register
0375h  Secondary Diskette Controller
        Command/Data Register
0377h  Read - Secondary Diskette Controller
        Digital Input Register
        Write - Select Register for Diskette
        Data Transfer Rate
0378h  LPT1 data port
0379h  LPT1 Status Port
       Bit 7 = Busy
       Bit 6 = Acknowledge
       Bit 5 = Out of Paper
       Bit 4 = Printer Selected
       Bit 3 = Error
       Bit 2 = IRQ Occurred
       Bit 1-0 = Reserved
037Ah  LPT1 Control Port
       Bit 7-6 = Reserved
       Bit 5 = Data Output Control
       Bit 4 = IRQ Enable
       Bit 3 = Select Printer
       Bit 2 = Initialize
       Bit 1 = Line Feed
       Bit 0 = Strobe
0380h-038Fh  Secondary SDLC Registers
IBM Cluster adapter
03A0h-03AFh Primary SDLC Registers
03B4h MDA CRT Index Register
03B5h MDA CRT Data Register

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Horizontal Total</td>
</tr>
<tr>
<td>01</td>
<td>Horizontal Displayed</td>
</tr>
<tr>
<td>02</td>
<td>Horizontal Sync Position</td>
</tr>
<tr>
<td>03</td>
<td>Horizontal Sync Pulse Width</td>
</tr>
<tr>
<td>04</td>
<td>Vertical Total</td>
</tr>
<tr>
<td>05</td>
<td>Vertical Displayed</td>
</tr>
<tr>
<td>06</td>
<td>Vertical Sync Position</td>
</tr>
<tr>
<td>07</td>
<td>Vertical Sync Pulse Width</td>
</tr>
<tr>
<td>08</td>
<td>Interlace Mode</td>
</tr>
<tr>
<td>09</td>
<td>Maximum Scan Lines</td>
</tr>
<tr>
<td>0A</td>
<td>Cursor Start</td>
</tr>
<tr>
<td>0B</td>
<td>Cursor End</td>
</tr>
<tr>
<td>0C</td>
<td>Start Address High</td>
</tr>
<tr>
<td>0D</td>
<td>Start Address Low</td>
</tr>
<tr>
<td>0E</td>
<td>Cursor Location High</td>
</tr>
<tr>
<td>0F</td>
<td>Cursor Location Low</td>
</tr>
<tr>
<td>10</td>
<td>Light Pen High</td>
</tr>
<tr>
<td>11</td>
<td>Light Pen Low</td>
</tr>
</tbody>
</table>

03B8h MDA Mode Control Register
- bit 7-6 = Reserved
- bit 5 = Blink Enable
- bit 4 = Reserved
- bit 3 = Video Enable
- bit 2-1 = Reserved
- bit 0 = High Resolution Mode

03B9h EGA Color Select
03BAh Read - EGA CRT Status Register
Write - EGA/VGA feature control register

03Bh Reserved for EGA
03Ch LPT1 Data Port
03Dh LPT1 Status Port
- Bit 7 = Busy
- Bit 6 = Acknowledge
- Bit 5 = Out of Paper
- Bit 4 = Printer Selected
- Bit 3 = Error
- Bit 2 = IRQ Occurred
- Bit 1-0 = Reserved

03Eh LPT 1 Control Port
- Bit 7-5 = Reserved
420 Chapter 10

Bit 4 = IRQ Enable
Bit 3 = Select Printer
Bit 2 = Initialize
Bit 1 = Line Feed
Bit 0 = Strobe
03BFh Hercules Configuration Switch Register
Bit 7-2 = Reserved
Bit 1 = Enable Upper 12K Graphic Buffer
Bit 0 = Disable Graphics Mode
03C0h EGA/VGA ATC Index/Data Register
03C1h VGA Other Attribute Register
03C2h Read - EGA/VGA Input Status 0 Register
Write - VEGA Miscellaneous Output Register
03C4h VGA Sequencer Index Register
03C5h VGA Other Sequencer Index Register
03C6h VGA PEL Mask Register
03C7h VGA PEL Address Read Mode/VGA DAC state register
03C8h VGA PEL Address Write Mode
03C9h VGA PEL Data Register
03CAh VGA Feature Control Register
03CBh VGA Miscellaneous Output Register
03CEh VGA Graphics Address Register
03CFh VGA Other Graphics Register
03D4h CGA CRT Index Register
03D5h CGA CRT (6845) data register
03D8h CGA Mode Control Register
Bit 7-6 = Reserved
Bit 5 = Blink Enable
Bit 4 = 640*200 Graphics Mode Select
Bit 3 = Video Enable
Bit 2 = Monochrome Signal Select
Bit 1 = Text Mode Select
Bit 0 = Text Mode Select
03D9h CGA Palette Register
Bit 7-6 = Reserved
Bit 5 = Active Color Set Select
Bit 4 = Intense Color Select
Bit 3 = Intense Border Select
Bit 2 = Red Border/Background/Foreground Select
Bit 1 = Green Border/Background/Foreground Select
PC Interfaces

Bit 0 - Blue Border/Background/Foreground Select
03DAh  CGA Status Register
       Bit 7-4 = Reserved
       Bit 3 = Vertical Retrace Status
       Bit 2 = Light Pen Status
       Bit 1 = Light Pen Trigger Set
       Bit 0 = Memory Select
03EAh  EGA/VGA Feature Control Register
03EBh  Clear Light Pen Latch
03ECb  Preset Light Pen Latch
03EDh  Serial Port 3 Transmitter/Receiver registers/Divisor Latch Low
03EEh  Serial Port 3 Interrupt Enable Register/Divisor Latch High
03EEh  Serial Port 3 Interrupt Identification Register
03E8h  Serial Port 3 Line Control Register
03E9h  Serial Port 3 Modem Control Register
03EAh  Serial Port 3 Line Status Register
03EBh  Serial Port 3 Scratchpad Register
03FHh  Primary Diskette Controller Status A
03F1h  Primary Diskette Controller Status B
03F2h  Primary Diskette Controller Digital Output Register
03F4h  Read - Primary Diskette Controller Main Status Register
        Primary Diskette Controller Data Rate Select Register
03F5h  Primary Diskette Controller Command/Data Register
03F7h  Read - Primary Diskette Controller Digital Input Register
        Write - Select Register for Diskette Data Transfer Rate
03F8h  Serial Port 3 Transmitter/Receiver registers/Divisor Latch Low
03F9h  Serial Port 1 Interrupt Enable Register/Divisor Latch High
03FAh  Serial Port 1 Interrupt Identification Register
03FBh  Serial Port 1 Line Control Register
03FCb  Serial Port 1 Modem Control Register
03FDh  Serial Port 1 Line Status Register
03F8h  Serial Port 1 Scratchpad Register
## Interrupt Function by Number

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Name and Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Divide by Zero Error</td>
</tr>
<tr>
<td>01h</td>
<td>Single Step</td>
</tr>
<tr>
<td>02h</td>
<td>Nonmaskable</td>
</tr>
<tr>
<td>03h</td>
<td>Breakpoint (Instruction 0x0CC)</td>
</tr>
<tr>
<td>04h</td>
<td>Overflow</td>
</tr>
<tr>
<td>05h</td>
<td>Print Screen</td>
</tr>
<tr>
<td>06h-07h</td>
<td>Reserved</td>
</tr>
<tr>
<td>08h</td>
<td>Time of Day Services</td>
</tr>
<tr>
<td>09h</td>
<td>Keyboard Interrupt</td>
</tr>
<tr>
<td>0Ah</td>
<td>Slaved Second Interrupt Controller</td>
</tr>
<tr>
<td>0Bh</td>
<td>COM1/COM3 Interrupt</td>
</tr>
<tr>
<td>0Ch</td>
<td>COM2/COM3 Interrupt</td>
</tr>
<tr>
<td>0Dh</td>
<td>Hard Disk Interrupt/LPT2 Interrupt</td>
</tr>
<tr>
<td>0Eh</td>
<td>Diskette Interrupt</td>
</tr>
<tr>
<td>0Fh</td>
<td>LPT1 Interrupt</td>
</tr>
<tr>
<td>10h</td>
<td>Video BIOS</td>
</tr>
<tr>
<td>11h</td>
<td>BIOS Equipment Check</td>
</tr>
<tr>
<td>12h</td>
<td>BIOS Memory Size Determine</td>
</tr>
<tr>
<td>13h</td>
<td>Disk I/O BIOS</td>
</tr>
<tr>
<td>14h</td>
<td>Serial Communications BIOS</td>
</tr>
<tr>
<td>15h</td>
<td>BIOS System Services</td>
</tr>
<tr>
<td>16h</td>
<td>Keyboard I/O BIOS</td>
</tr>
<tr>
<td>17h</td>
<td>Printer BIOS</td>
</tr>
<tr>
<td>18h</td>
<td>Resident BASIC Start Vector</td>
</tr>
<tr>
<td>19h</td>
<td>BootStrap Loader</td>
</tr>
<tr>
<td>1Ah</td>
<td>Time of Day BIOS</td>
</tr>
<tr>
<td>1Bh</td>
<td>Keyboard Break Vector</td>
</tr>
<tr>
<td>1Ch</td>
<td>Timer Tick Vector</td>
</tr>
<tr>
<td>1Dh</td>
<td>Table Address of Video Parameters</td>
</tr>
<tr>
<td>1Eh</td>
<td>Table Address of Disk Parameters</td>
</tr>
<tr>
<td>1Fh</td>
<td>Table Address of Graphic Characters</td>
</tr>
</tbody>
</table>
### PC Interfaces

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20h</td>
<td>MS-DOS Program Terminate</td>
</tr>
<tr>
<td>21h</td>
<td>MS-DOS Function APIs Terminate</td>
</tr>
<tr>
<td>22h</td>
<td>MS-DOS Terminate Vector</td>
</tr>
<tr>
<td>23h</td>
<td>MS-DOS &quot;Ctrl-C&quot; Vector</td>
</tr>
<tr>
<td>24h</td>
<td>MS-DOS Error Handler Vector</td>
</tr>
<tr>
<td>25h-26h</td>
<td>MS-DOS Absolute Disk I/O</td>
</tr>
<tr>
<td>27h</td>
<td>MS-DOS Terminate Stay Resident API</td>
</tr>
<tr>
<td>28h-2Eh</td>
<td>MS-DOS Reserved</td>
</tr>
<tr>
<td>2Fh</td>
<td>MS-DOS Multiplex Interrupt</td>
</tr>
<tr>
<td>30h-32h</td>
<td>MS-DOS Reserved Mouse BIOS</td>
</tr>
<tr>
<td>34h-3Fh</td>
<td>MS-DOS Reserved Re vectored Disk I/O BIOS Interrupt 13h</td>
</tr>
<tr>
<td>40h</td>
<td>Re vectored Disk Drive 0 Parameters</td>
</tr>
<tr>
<td>41h</td>
<td>Re vectored EGA BIOS Interrupt 10h Table Address of EGA Parameters</td>
</tr>
<tr>
<td>42h</td>
<td>ROM BIOS Alarm Handler</td>
</tr>
<tr>
<td>44h-4Fh</td>
<td>Reserved PC/AT Alarm BIOS Interrupt</td>
</tr>
<tr>
<td>45h-59h</td>
<td>Reserved NETB IOS Function APIs</td>
</tr>
<tr>
<td>5Ah</td>
<td>NETBIOS Remap of Vector 19h NETBIOS Entry Point</td>
</tr>
<tr>
<td>5Bh</td>
<td>NETBIOS Memory Function APIs</td>
</tr>
<tr>
<td>5Ch</td>
<td>Reserved 67h LIM EMS Memory Function APIs</td>
</tr>
<tr>
<td>68h-6Fh</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
424 Chapter 10

70h RTC Interrupt
71h Slave Interrupt
Controller Redirect
72h IRQ10
73h IRQ11
74h IRQ12
75h IRQ13
76h IRQ14
77h IRQ15
78h-7Fh Not Allocated/
Available for Use
80h-85h Reserved for Cassette
BASIC
86h-F0h Used by BASIC
F1h-FFh Used during PC Boot
as a Temporary Stack
Area. Should NOT be
used for Interrupts
or Variables

ISAT Bus

When the PC was designed, IBM designed the motherboard and specified the ISA slots in such a way that the complexity of the bus was hidden from the user (Fig. 10.2). The read/write cycle on the ISA bus is shown in Fig. 10.3. This waveform is identical for the I/O address space reads and writes.

![ISA Slot Diagram](image-url)

Figure 10.2 Processor/ISA Block Diagram
The 8-bit ISA bus consists of a two-sided 31-pin card edge connector with the pins defined as:

<table>
<thead>
<tr>
<th>Pin</th>
<th>&quot;A&quot; (Connector)</th>
<th>&quot;B&quot; (Solder)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I/O CHK CHK</td>
<td>Ground</td>
</tr>
<tr>
<td>2</td>
<td>D7</td>
<td>Reset</td>
</tr>
<tr>
<td>3</td>
<td>D6</td>
<td>+5 V</td>
</tr>
<tr>
<td>4</td>
<td>D5</td>
<td>IRQ2</td>
</tr>
<tr>
<td>5</td>
<td>D4</td>
<td>+5 V</td>
</tr>
<tr>
<td>6</td>
<td>D3</td>
<td>DRQ2</td>
</tr>
<tr>
<td>7</td>
<td>D2</td>
<td>-12 V</td>
</tr>
<tr>
<td>8</td>
<td>D1</td>
<td>_CARD SLCTD</td>
</tr>
<tr>
<td>9</td>
<td>D0</td>
<td>+12 V</td>
</tr>
<tr>
<td>10</td>
<td>IO CHK RDY</td>
<td>Ground</td>
</tr>
<tr>
<td>11</td>
<td>ABN</td>
<td>_MEMW</td>
</tr>
<tr>
<td>12</td>
<td>A19</td>
<td>_MEMS</td>
</tr>
<tr>
<td>13</td>
<td>A18</td>
<td>_IOW</td>
</tr>
<tr>
<td>14</td>
<td>A17</td>
<td>_IOR</td>
</tr>
</tbody>
</table>

Figure 10.3 ISA Bus Timing

**ISA pinouts**

The 8-bit ISA bus consists of a two-sided 31-pin card edge connector with the pins defined as:
The data and address buses are buffered to the processor. Addresses from 0x00000 to 0x0FFFFF (zero to one megabyte) can be accessed with the 8-bit connector. Memory devices can be located 0x0C0000 to 0x0DFFF, but care must be taken to avoid “contention” with other devices located within this memory space. The ISA Bus Pin Functions are given below:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BALE</td>
<td>“Buffered ALE” and was the term used in the original PC because the ALE line was produced by the 8088’s instruction sequence clock. This pin was buffered to avoid having the ISA bus directly processor driven. Today, this bit is more commonly known as “ALE” and provides essentially the same operation and timing as “BALE”</td>
</tr>
<tr>
<td>I/O CH CHK</td>
<td>Pin was designed for use with parity checked memory. If a byte was read...</td>
</tr>
</tbody>
</table>
that did not match the saved parity, a NMI interrupt request was made of the processor. In Modern Systems, this Pin can be pulled Low (made active) to indicate a system error.

**I/O CH RDY**
Line driven low by an adapter if it needs more time to complete an operation.

**_IOR_/ _IOW**
I/O Register Read and Write Enables

**_MEMR_/ _MEMW**
Indicate the processor is reading and writing to ISA bus memory.

**IRQ3-IRQ7**
Hardware Interrupt Request Lines. When these lines are driven high, the 8259As on the motherboard (which are known as "Programmable Interrupt Controllers" or "PICs") will process the request in a descending order of priority. These lines are driven high to request an interrupt. A PICmicro® MCU can drive these lines, but it should only be active when a "high" is driven onto the interrupt line to allow other devices to share the interrupt pins. If a PICmicro® MCU is used to drive these lines active, then there must be some way for the processor to reset the PICmicro® MCU interrupt request.

**CLOCK**
Runs at four times NTSC "Color Burst" frequency (14.31818 MHz). The 14.31818 MHz clock was distributed to the system to provide clocking for the "MDA" and "CGA" video display cards. This clock can be useful for providing a simple clock for microcontroller and other clocked devices on adapter cards.

**OSC**
Pin is driven at up to 8 MHz. Used to Request a DMA transfer to take place. When the corresponding "_DACK#" pin is driven high, the DMA controller is reading or writing an I/O address of an adapter card. When the DMA controllers have
control of the bus over the processor, the "AEN" pin is active to indicate to other adapters that a DMA operation is in process. When all the DMA data has been transferred, the "T/C" bit is pulsed high to indicate the operation has completed. When the "T/C" bit becomes active, the adapter should request a hardware interrupt to indicate to software that the operation is complete.

_DACK# Active when the DMA Channel is reading/writing the I/O device. "_DACK0" or "_REFRESH" is used with DRAM memory to request a "RAS only refresh" of the system memory.

_MASTER Driven by an adapter when it is requesting to take over the bus and drive its own signals.

Interrupts

Interrupts IRQ3, IRQ4, and IRQ7 are recommended for use in a PC system. Interrupts are driven high and should use the circuit shown in Fig. 10.4 to allow multi-

Figure 10.4 Multiple Interrupt Request Circuit
The PC Interrupt Handler should be defined as:

1. Save the original vector using MS-DOS interrupt 021h AH = 035h API.
2. Set the new vector using MS-DOS interrupt 021h AH = 025h API.
3. Enable the Interrupt Request Mask bit in the 8259.

To enable the interrupt request mask bit in the 8259, the appropriate interrupt mask register bit has to be reset. This register is at the 8259’s “Base Address” plus one. This can be done with the following statement:

```c
outp(IntBase + 1, inp(IntBase + 1) & ((0x0FF ^ (1 << Bit))));
```

To “release” the interrupt vector and the interrupt source at the end of the application, the following steps must be taken:

1. Disable the Interrupt Request mask bit in the 8259.
2. Restore the original vector using MS-DOS Interrupt 021h AH = 025h API.

**Keyboard and Mouse Ports**

The PCs keyboard and mouse ports operate with a synchronous serial data protocol that was first introduced with the original IBM PC. This protocol allows data to be sent from the keyboard in such a way that multiple pressed keys can be recognized within the PC without any key presses being lost. The standard was enhanced with the PC/AT as a bidirectional communication.
method. Three years later, when the PS/2 was introduced, the “mouse” interface also used the keyboard’s protocol, freeing up a serial port or ISA slot which, up to this point, was needed for the mouse interface. The keyboard protocol used in the PC was so successful that IBM used it for all its PC, terminal, and workstation product lines that have been developed from 1981 and it is also used by many other PC vendors.

**Connector specification**

The female 6-pin “Mini-DIN” keyboard connector facing out of the PC is shown in Fig. 10.5. The port can usually supply up to 100 mA over and above the keyboard requirements. The power (+5 VDC) may or may not be fused, so any hardware put on the port must not draw excessive current to prevent damage to the motherboard.

**Keyboard operation with timing diagrams**

Data from the keyboard looks like the waveform shown in Fig. 10.6. The parity bit is “odd”, which is to say the eight data bits plus the parity is an odd number. The data line should not change for at least 5 usecs from the

![Figure 10.5 PC Keyboard Connector Pinout](image)
change of the clock line. The clock line should be high or low for at least 30 usecs (with 40 usecs being typical).

Data that is sent from the system unit to the keyboard is similar, but with the clock inverted. The data changes while the clock is low and is latched in when the clock goes high as is shown in Fig. 10.7. When data is sent from the keyboard, the clock is pulled low and then data is sent with the keyboard accepting data when the clock is pulsed high. The bit timings are the same as data from the keyboard.

These two protocols are used to allow a device wired in parallel to monitor the communication to and from the PC.

Additional devices can be added to the keyboard/mouse connector in parallel as is shown in Fig. 10.8.

**Keyboard scan codes**

In MS-DOS, the Keyboard Codes are normally a combination of the keyboard scan code and appropriate ASCII
code. The table below shows the different codes returned for keystrokes by themselves, and with a “Shift”, “Ctrl”, or “Alt” Modifier.

The table below shows the codes in scan/ASCII configuration for the extended function keyboard characters. The standard function codes are the same except that “F11”, “F12”, and the keypad “Center Key” do not return any codes and for the explicit arrow and explicit “Insert”, “Home”, “Page Up”, “Delete”, “End”, and “Page Down” keys, the 0x0E0 ASCII code is actually 0x000.

All values in the table below are in hex and I have put in the scan codes as they appear on my PC. I have not made allowances for upper and lower case in this table as this is processed by the PC itself. “KP” indicates the Keypad and it, or a single “A” (which indicates Alternate arrow and other keys), followed by “UA”, “DA”, “LA”, or
"RA" indicates an Arrow. "I", "D", "H", "PU", "PD", or "E" with "KP" or "A" indicates the "Insert", "Delete", "Home", "Page Up", "Page Down", or "End" on the Keypad, respectively.

The Keypad numbers, when "Alt" is pressed is used to enter in specific ASCII codes in Decimal. For example, "Alt", "6", "5" will enter in an ASCII "A" character. These keys in the table below are marked "#".

<table>
<thead>
<tr>
<th>PC Keyboard Scan Codes</th>
<th>Codes</th>
<th>Codes</th>
<th>Codes</th>
<th>Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Standard</td>
<td>&quot;Shift&quot;</td>
<td>&quot;Ctrl&quot;</td>
<td>&quot;Alt&quot;</td>
</tr>
<tr>
<td>Esc</td>
<td>01/1B</td>
<td>01/1B</td>
<td>01/1B</td>
<td>01/00</td>
</tr>
<tr>
<td>1</td>
<td>02/31</td>
<td>02/21</td>
<td>--</td>
<td>78/00</td>
</tr>
<tr>
<td>2</td>
<td>03/32</td>
<td>03/40</td>
<td>03/00</td>
<td>79/00</td>
</tr>
<tr>
<td>3</td>
<td>04/33</td>
<td>04/23</td>
<td>--</td>
<td>7A/00</td>
</tr>
<tr>
<td>4</td>
<td>05/34</td>
<td>05/24</td>
<td>--</td>
<td>7B/00</td>
</tr>
<tr>
<td>5</td>
<td>06/35</td>
<td>06/25</td>
<td>--</td>
<td>7C/00</td>
</tr>
<tr>
<td>6</td>
<td>07/36</td>
<td>07/5E</td>
<td>07/1E</td>
<td>7D/00</td>
</tr>
<tr>
<td>7</td>
<td>08/37</td>
<td>08/26</td>
<td>--</td>
<td>7E/00</td>
</tr>
<tr>
<td>8</td>
<td>09/38</td>
<td>09/2A</td>
<td>--</td>
<td>7F/00</td>
</tr>
<tr>
<td>9</td>
<td>0A/39</td>
<td>0A/28</td>
<td>--</td>
<td>80/00</td>
</tr>
<tr>
<td>0</td>
<td>0B/30</td>
<td>0B/29</td>
<td>--</td>
<td>81/00</td>
</tr>
<tr>
<td>-</td>
<td>0C/2D</td>
<td>0C/5F</td>
<td>0C/1F</td>
<td>82/00</td>
</tr>
<tr>
<td>-</td>
<td>0D/3D</td>
<td>0C/2B</td>
<td>--</td>
<td>83/00</td>
</tr>
<tr>
<td>BS</td>
<td>0E/0E</td>
<td>0E/08</td>
<td>0E/7F</td>
<td>0E/00</td>
</tr>
<tr>
<td>Tab</td>
<td>0F/0F</td>
<td>0F/00</td>
<td>94/00</td>
<td>A5/00</td>
</tr>
<tr>
<td>Q</td>
<td>10/71</td>
<td>10/51</td>
<td>10/11</td>
<td>10/00</td>
</tr>
<tr>
<td>W</td>
<td>11/77</td>
<td>11/57</td>
<td>11/17</td>
<td>11/00</td>
</tr>
<tr>
<td>E</td>
<td>12/65</td>
<td>12/45</td>
<td>12/05</td>
<td>12/00</td>
</tr>
<tr>
<td>R</td>
<td>13/72</td>
<td>13/52</td>
<td>13/12</td>
<td>13/00</td>
</tr>
<tr>
<td>T</td>
<td>14/74</td>
<td>14/54</td>
<td>14/14</td>
<td>14/00</td>
</tr>
<tr>
<td>Y</td>
<td>15/79</td>
<td>15/59</td>
<td>15/19</td>
<td>15/00</td>
</tr>
<tr>
<td>U</td>
<td>16/75</td>
<td>16/55</td>
<td>16/15</td>
<td>16/00</td>
</tr>
<tr>
<td>I</td>
<td>17/69</td>
<td>17/49</td>
<td>17/09</td>
<td>17/00</td>
</tr>
<tr>
<td>O</td>
<td>18/6F</td>
<td>18/4F</td>
<td>18/0F</td>
<td>18/00</td>
</tr>
<tr>
<td>P</td>
<td>19/70</td>
<td>19/50</td>
<td>19/10</td>
<td>19/00</td>
</tr>
<tr>
<td>{</td>
<td>1A/5B</td>
<td>1A/7B</td>
<td>1A/1B</td>
<td>1A/00</td>
</tr>
<tr>
<td>}</td>
<td>1B/5D</td>
<td>1B/7D</td>
<td>1B/1D</td>
<td>1B/00</td>
</tr>
</tbody>
</table>
### PC Keyboard Scan Codes (Continued)

<table>
<thead>
<tr>
<th>Key</th>
<th>Scan Code 1</th>
<th>Scan Code 2</th>
<th>Scan Code 3</th>
<th>Scan Code 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter</td>
<td>1C/0D</td>
<td>1C/0D</td>
<td>1C/0A</td>
<td>1C/00</td>
</tr>
<tr>
<td>A</td>
<td>1D/61</td>
<td>1E/41</td>
<td>1E/01</td>
<td>1E/00</td>
</tr>
<tr>
<td>S</td>
<td>1F/73</td>
<td>1F/53</td>
<td>1F/13</td>
<td>1F/00</td>
</tr>
<tr>
<td>D</td>
<td>20/64</td>
<td>20/44</td>
<td>20/04</td>
<td>20/00</td>
</tr>
<tr>
<td>F</td>
<td>21/66</td>
<td>21/46</td>
<td>21/06</td>
<td>21/00</td>
</tr>
<tr>
<td>G</td>
<td>22/67</td>
<td>22/47</td>
<td>22/07</td>
<td>22/00</td>
</tr>
<tr>
<td>H</td>
<td>23/68</td>
<td>23/48</td>
<td>23/08</td>
<td>23/00</td>
</tr>
<tr>
<td>J</td>
<td>24/6A</td>
<td>24/4A</td>
<td>24/0A</td>
<td>24/00</td>
</tr>
<tr>
<td>K</td>
<td>25/6B</td>
<td>25/4B</td>
<td>25/0B</td>
<td>25/00</td>
</tr>
<tr>
<td>L</td>
<td>26/6C</td>
<td>26/4C</td>
<td>26/0C</td>
<td>26/00</td>
</tr>
<tr>
<td>;</td>
<td>27/3B</td>
<td>27/3A</td>
<td>--</td>
<td>27/00</td>
</tr>
<tr>
<td>'</td>
<td>28/27</td>
<td>28/22</td>
<td>--</td>
<td>28/00</td>
</tr>
<tr>
<td>,</td>
<td>29/60</td>
<td>29/7E</td>
<td>--</td>
<td>29/00</td>
</tr>
<tr>
<td>\</td>
<td>2B/5C</td>
<td>2B/7C</td>
<td>2B/1C</td>
<td>2B/00</td>
</tr>
<tr>
<td>Z</td>
<td>2C/7A</td>
<td>2C/5A</td>
<td>2C/1A</td>
<td>2C/00</td>
</tr>
<tr>
<td>X</td>
<td>2D/78</td>
<td>2D/58</td>
<td>2D/18</td>
<td>2D/00</td>
</tr>
<tr>
<td>C</td>
<td>2E/63</td>
<td>2E/43</td>
<td>2E/03</td>
<td>2E/00</td>
</tr>
<tr>
<td>V</td>
<td>2F/76</td>
<td>2F/56</td>
<td>2F/18</td>
<td>2F/00</td>
</tr>
<tr>
<td>B</td>
<td>30/62</td>
<td>30/42</td>
<td>30/02</td>
<td>30/00</td>
</tr>
<tr>
<td>N</td>
<td>31/6E</td>
<td>31/4E</td>
<td>31/0E</td>
<td>31/00</td>
</tr>
<tr>
<td>M</td>
<td>32/6D</td>
<td>32/4D</td>
<td>32/0D</td>
<td>32/00</td>
</tr>
<tr>
<td>,</td>
<td>33/2C</td>
<td>33/3C</td>
<td>--</td>
<td>33/00</td>
</tr>
<tr>
<td>.</td>
<td>34/2E</td>
<td>34/3E</td>
<td>--</td>
<td>34/00</td>
</tr>
<tr>
<td>/</td>
<td>35/2F</td>
<td>35/3F</td>
<td>--</td>
<td>35/00</td>
</tr>
<tr>
<td>KP *</td>
<td>37/2A</td>
<td>37/2A</td>
<td>96/00</td>
<td>37/00</td>
</tr>
<tr>
<td>SPACE</td>
<td>39/20</td>
<td>39/20</td>
<td>39/20</td>
<td>39/20</td>
</tr>
<tr>
<td>F1</td>
<td>3B/00</td>
<td>54/00</td>
<td>5B/00</td>
<td>6A/00</td>
</tr>
<tr>
<td>F2</td>
<td>3C/00</td>
<td>55/00</td>
<td>5F/00</td>
<td>69/00</td>
</tr>
<tr>
<td>F3</td>
<td>3D/00</td>
<td>56/00</td>
<td>60/00</td>
<td>6A/00</td>
</tr>
<tr>
<td>F4</td>
<td>3E/00</td>
<td>57/00</td>
<td>61/00</td>
<td>6B/00</td>
</tr>
<tr>
<td>F5</td>
<td>3F/00</td>
<td>58/00</td>
<td>62/00</td>
<td>6C/00</td>
</tr>
<tr>
<td>F6</td>
<td>40/00</td>
<td>59/00</td>
<td>63/00</td>
<td>6D/00</td>
</tr>
<tr>
<td>F7</td>
<td>41/00</td>
<td>5A/00</td>
<td>64/00</td>
<td>6B/00</td>
</tr>
<tr>
<td>F8</td>
<td>42/00</td>
<td>5B/00</td>
<td>65/00</td>
<td>6F/00</td>
</tr>
<tr>
<td>F9</td>
<td>43/00</td>
<td>5C/00</td>
<td>66/00</td>
<td>70/00</td>
</tr>
<tr>
<td>F10</td>
<td>44/00</td>
<td>5D/00</td>
<td>67/00</td>
<td>71/00</td>
</tr>
<tr>
<td>F11</td>
<td>45/00</td>
<td>5E/00</td>
<td>68/00</td>
<td>85/00</td>
</tr>
<tr>
<td>F12</td>
<td>46/00</td>
<td>5F/00</td>
<td>69/00</td>
<td>8B/00</td>
</tr>
<tr>
<td>KP H</td>
<td>47/00</td>
<td>47/37</td>
<td>77/00</td>
<td>#</td>
</tr>
<tr>
<td>KP UA</td>
<td>48/00</td>
<td>48/38</td>
<td>8D/00</td>
<td>#</td>
</tr>
<tr>
<td>KP PU</td>
<td>49/00</td>
<td>49/39</td>
<td>84/00</td>
<td>#</td>
</tr>
<tr>
<td>KP -</td>
<td>4A/2D</td>
<td>4A/2D</td>
<td>8E/00</td>
<td>4A/00</td>
</tr>
</tbody>
</table>
Keyboard controller commands

The PC itself has a number of commands that it can send to the keyboard that include:

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0ED</td>
<td>Set Indicator LED's. The next character out is the LED status</td>
</tr>
<tr>
<td>0x0EE</td>
<td>Echo - Keyboard Returns 0x0EE</td>
</tr>
<tr>
<td>0x0EF-0x0F2</td>
<td>Ignored by the Keyboard</td>
</tr>
<tr>
<td>0x0F3</td>
<td>Set Typematic rate, next character is the rate</td>
</tr>
</tbody>
</table>
In all these cases (except for the "ignore" and "echo" commands), the keyboard sends back the "Acknowledge" character 0x0FA.

**BIOS interfaces**

When Data is transferred between the PC’s processor and the keyboard controller, the following information is passed as well:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Set when &quot;Insert&quot; State Active</td>
</tr>
<tr>
<td>6</td>
<td>Set when &quot;Caps Lock&quot; Active</td>
</tr>
<tr>
<td>5</td>
<td>Set when &quot;Num Lock&quot; Active</td>
</tr>
<tr>
<td>4</td>
<td>Set when &quot;Scroll Lock&quot; Active</td>
</tr>
<tr>
<td>3</td>
<td>Set when a &quot;Alt&quot; Key Held Down</td>
</tr>
<tr>
<td>2</td>
<td>Set when a &quot;Ctrl&quot; Key Held Down</td>
</tr>
<tr>
<td>1</td>
<td>Set when the Left &quot;Shift&quot; Key Held Down</td>
</tr>
<tr>
<td>0</td>
<td>Set when the Right &quot;Shift&quot; Key Held Down</td>
</tr>
</tbody>
</table>

**Keyboard Flags Byte**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Set when &quot;Insert&quot; State Active</td>
</tr>
<tr>
<td>6</td>
<td>Set when &quot;Caps Lock&quot; Active</td>
</tr>
<tr>
<td>5</td>
<td>Set when &quot;Num Lock&quot; Active</td>
</tr>
<tr>
<td>4</td>
<td>Set when &quot;Scroll Lock&quot; Active</td>
</tr>
<tr>
<td>3</td>
<td>Set when a &quot;Alt&quot; Key Held Down</td>
</tr>
<tr>
<td>2</td>
<td>Set when a &quot;Ctrl&quot; Key Held Down</td>
</tr>
<tr>
<td>1</td>
<td>Set when the Left &quot;Shift&quot; Key Held Down</td>
</tr>
<tr>
<td>0</td>
<td>Set when the Right &quot;Shift&quot; Key Held Down</td>
</tr>
</tbody>
</table>
To access the keyboard BIOS functions, an “int 016h” instruction is executed with the registers set up as defined in the table below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Set when “SysReq” Key Pressed</td>
</tr>
<tr>
<td>6</td>
<td>Set when “Caps Lock” Key Pressed</td>
</tr>
<tr>
<td>5</td>
<td>Set when “Num Lock” Key Pressed</td>
</tr>
<tr>
<td>4</td>
<td>Set when “Scroll Lock” Key Pressed</td>
</tr>
<tr>
<td>3</td>
<td>Set when Right “Alt” Key Pressed</td>
</tr>
<tr>
<td>2</td>
<td>Set when Right “Ctrl” Key Pressed</td>
</tr>
<tr>
<td>1</td>
<td>Set when Left “Alt” Key Pressed</td>
</tr>
<tr>
<td>0</td>
<td>Set when Left “Ctrl” Key Pressed</td>
</tr>
<tr>
<td>Function</td>
<td>Input</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Read</td>
<td>AH = 00h</td>
</tr>
<tr>
<td>Character</td>
<td>AL = ASCII</td>
</tr>
<tr>
<td>Read Status</td>
<td>AH = 01h</td>
</tr>
<tr>
<td>return</td>
<td>AL = ASCII</td>
</tr>
<tr>
<td>Read Flags</td>
<td>AH = 02h</td>
</tr>
<tr>
<td>Set Typematic</td>
<td>AH = 03h</td>
</tr>
<tr>
<td>Rate and</td>
<td>AL = 5</td>
</tr>
<tr>
<td>Delay</td>
<td>BH = Delay</td>
</tr>
<tr>
<td></td>
<td>0 - 250ms,</td>
</tr>
<tr>
<td></td>
<td>1 - 500ms,</td>
</tr>
<tr>
<td></td>
<td>2 - 750ms,</td>
</tr>
<tr>
<td></td>
<td>3 - 1000ms</td>
</tr>
<tr>
<td></td>
<td>BL = Rate</td>
</tr>
<tr>
<td></td>
<td>0 - 30 cps,</td>
</tr>
</tbody>
</table>
### Read Typematic Rate and Delay

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
<th>BL</th>
<th>BH</th>
<th>Delay and Rate Delay set into the Keyboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>03h</td>
<td>6</td>
<td>Rate</td>
<td>Delay</td>
<td></td>
</tr>
</tbody>
</table>

### Keyboard Write

This command writes a new Character into the keyboard buffer (and not to the keyboard or other external Device as the name would imply).

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
<th>BL</th>
<th>BH</th>
<th>Written Successfully</th>
</tr>
</thead>
<tbody>
<tr>
<td>05h</td>
<td>0</td>
<td>ASCII</td>
<td>Scan Code</td>
<td></td>
</tr>
</tbody>
</table>

### Keyboard Functionality Determination

This API returns the capabilities of the keyboard and hardware to change the

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
<th>BL</th>
<th>BH</th>
<th>Bit 3 - Set If can read Delay/Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>09h</td>
<td>Function Code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Input</td>
<td>Output</td>
<td>Comments</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------</td>
<td>-------------------</td>
<td>--------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Bit 2 - Set</td>
<td>Typematic Rate</td>
<td></td>
<td>Bit 2 - Set Bit 2 - Set If can Set Delay/Rate Bit 1 - Set If cannot Set Delay/Rate Bit 0 - Set If return To default Delay/Rate Supported</td>
<td></td>
</tr>
<tr>
<td>Bit 1 - Set</td>
<td></td>
<td></td>
<td>Bit 1 - Set If cannot Set Delay/Rate</td>
<td></td>
</tr>
<tr>
<td>Bit 0 - Set</td>
<td></td>
<td></td>
<td>Bit 0 - Set If return To default Delay/Rate Supported</td>
<td></td>
</tr>
<tr>
<td>Extended</td>
<td>AH = 10h</td>
<td>AH = Scan Code</td>
<td>Return the full keyboard code if Keyboard Buffer has an unread key or wait for a key to return.</td>
<td></td>
</tr>
<tr>
<td>Keyboard Read</td>
<td></td>
<td>AL = ASCII Code</td>
<td>Return the keyboard code if Keyboard Buffer has an unread key or wait for a key to return.</td>
<td></td>
</tr>
<tr>
<td>Extended</td>
<td>AH = 11h</td>
<td>AH = Scan Code</td>
<td>Check the Keyboard buffer and return the next key to process or set the Zero Flag.</td>
<td></td>
</tr>
<tr>
<td>Keyboard Status</td>
<td></td>
<td>AL = ASCII Code</td>
<td>Check the Keyboard buffer and return the next key to process or set the Zero Flag.</td>
<td></td>
</tr>
<tr>
<td>Extended</td>
<td>AH = 12h</td>
<td>AH = Extended</td>
<td>Return the Extended Keyboard Shift/Alt Shift/ Ctrl/Alt Status.</td>
<td></td>
</tr>
<tr>
<td>Shift Status</td>
<td></td>
<td>Keyboard Flags</td>
<td>Return the Set of Flags Byte</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AL = Keyboard Flags Byte</td>
<td>Return the Set of Flags Byte</td>
<td></td>
</tr>
</tbody>
</table>
PC Interfaces  441

Keyboard commands
To simply process a keystroke in “C”, the following “switch” code could be used:

```c
switch ((KeySave = KEYREAD()) & 0x0FF) {// Process the Key
    case 0x000:                           // Special Function Keys
        ;
        break;
    case 0x0E0:                           // Process the Scan Code:
        KeySave = (KeySave >> 8) & 0x0FF;
        ;
        break;
    case 0x00D:                           // Handle "Enter":
        ;
        break;
    ;
    ;
    default:                              // Other, Unneeded Keys
        ;
} // endswitch
```

Serial Port
The PC’s serial port’s design has not changed since the PC was introduced in 1981. Since that time, a 9-pin connector has been specified for the port (in the PC/AT) and the ability to buffer data within the serial port has been added.

Connector pinouts
DB-9 and D-9 pin RS-232 connectors are shown in Fig. 10.9. These connectors are wired as:
442  Chapter 10

8250 block diagram
The “8250” UART is the basis for serial communications within the PC (Fig.10.10).

DB-25 (Male)  

D-9 (Male)  

Figure 10.9  IBM PC DB-25 and D-9 Pin RS-232 Connectors
Serial port base addresses

The Serial Port “Base Addresses” are as follows:

<table>
<thead>
<tr>
<th>PC Serial Port Base Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>COM1</td>
</tr>
<tr>
<td>COM2</td>
</tr>
<tr>
<td>COM3</td>
</tr>
<tr>
<td>COM4</td>
</tr>
</tbody>
</table>

Each “Base Address” is used as an initial offset to eight registers that are used by the Serial Port Controller (The “8250”). The “Interrupt Number” is the interrupt vector requested when an interrupt condition is encountered. Note that “COM4” has conflicting addresses with the 8514/A (“SuperVGA”) Graphics Adapter.
Chapter 10

8250 registers

The 8250 consists of eight registers offset from the “base address”.

<table>
<thead>
<tr>
<th>Base Address Offset</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Transmitter Holding Register</td>
</tr>
<tr>
<td></td>
<td>Receiver Character Buffer</td>
</tr>
<tr>
<td></td>
<td>LSB Divisor Latch</td>
</tr>
<tr>
<td>1</td>
<td>Interrupt Enable Register/MSB</td>
</tr>
<tr>
<td></td>
<td>Divisor Latch</td>
</tr>
<tr>
<td>2</td>
<td>Interrupt Identification Register</td>
</tr>
<tr>
<td>3</td>
<td>Line Control Register</td>
</tr>
<tr>
<td>4</td>
<td>Modem Control Register</td>
</tr>
<tr>
<td>5</td>
<td>Line Status Register</td>
</tr>
<tr>
<td>6</td>
<td>Modem Status Register</td>
</tr>
<tr>
<td>7</td>
<td>Scratchpad Register</td>
</tr>
</tbody>
</table>

Data Speed is specified by loading a 16-bit divisor value into the Rx/Tx Holding Register and Interrupt Enable Register addresses after bit 7 of the Line Control Register is set. The value loaded into the register is multiplied by 16 and divided into 1.8432 MHz to get the actual data rate.

Data Rate = \( \frac{1.8432 \text{ MHz}}{16 \times \text{Divisor}} \)

The divisors for different standard data rates are:

<table>
<thead>
<tr>
<th>Data Rate</th>
<th>Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>110 bps</td>
<td>0x0417</td>
</tr>
<tr>
<td>300 bps</td>
<td>0x0180</td>
</tr>
</tbody>
</table>
After a character is received, it will set a number of conditions (including error conditions) that can only be reset by reading the character in the Receive Holding Register. For this reason it is always a good idea to read the serial port at the start of an application. By reading the port, the status and left over characters are “cleared” out.

Writing to the base address (with no offset added) loads a character into the “Transmit Holding Register”, which will be loaded as soon as the shift out register has completed sending the previous character. Often, when starting transmission, nothing will be in the shift register so the character is loaded immediately into the shift register, freeing up the holding register for the next character.

When any interrupts are enabled in the 8250, they will output an interrupt request (Fig. 10.11). This may not be desirable, so in the PC, some hardware was added to globally mask the interrupt.

“Out2” is controlled within the “Modem Control Register”.

<table>
<thead>
<tr>
<th>Data Rate</th>
<th>Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>600 bps</td>
<td>0x00C0</td>
</tr>
<tr>
<td>1200 bps</td>
<td>0x0060</td>
</tr>
<tr>
<td>2400 bps</td>
<td>0x0030</td>
</tr>
<tr>
<td>9600 bps</td>
<td>0x000C</td>
</tr>
<tr>
<td>19200 bps</td>
<td>0x0006</td>
</tr>
<tr>
<td>115200 bps</td>
<td>0x0001</td>
</tr>
</tbody>
</table>

PC Serial Port Interrupt Enable Register (Base + 1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-7</td>
<td>Unused, normally set to zero.</td>
</tr>
</tbody>
</table>
Chapter 10

PC Serial Port Interrupt Enable Register (Base + 1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>When set an interrupt request on change of state for modem interface lines.</td>
</tr>
<tr>
<td>2</td>
<td>Request interrupt for change in receiver holding register status</td>
</tr>
<tr>
<td>1</td>
<td>Request interrupt if the holding register is empty</td>
</tr>
<tr>
<td>0</td>
<td>Request interrupt for received character</td>
</tr>
</tbody>
</table>

PC Serial Port Interrupt Identification Register (Base + 2)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-7</td>
<td>Unused, Normally set to zero</td>
</tr>
<tr>
<td>1-2</td>
<td>Interrupt ID Bits</td>
</tr>
<tr>
<td>B2 B1</td>
<td>Priority Request Type</td>
</tr>
<tr>
<td>0 0</td>
<td>Lowest Change in Modem Status Lines</td>
</tr>
<tr>
<td>0 1</td>
<td>Third Transmitter Holding Register Empty</td>
</tr>
<tr>
<td>1 0</td>
<td>Second Data Received</td>
</tr>
<tr>
<td>1 1</td>
<td>Highest Receive Line Status Change</td>
</tr>
</tbody>
</table>

Figure 10.11 IBM PC Serial Interrupt Enable Hardware
### PC Serial Port Line Control Register (Base + 3)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>When set, the Transmitter Holding and Interrupt Enable Registers are used for loading the data speed divisor</td>
</tr>
<tr>
<td>6</td>
<td>When set, the 8250 outputs a &quot;Break Conditions&quot; (sending a space) until this bit is reset</td>
</tr>
<tr>
<td>3-5</td>
<td>Parity Type Specification</td>
</tr>
<tr>
<td></td>
<td>B5 B4 B3</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 - No Parity</td>
</tr>
<tr>
<td>0</td>
<td>0 0 1 - Odd Parity</td>
</tr>
<tr>
<td>0</td>
<td>0 1 0 - No Parity</td>
</tr>
<tr>
<td>0</td>
<td>0 1 1 - Even Parity</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 - No Parity</td>
</tr>
<tr>
<td>1</td>
<td>0 1 1 - &quot;Mark&quot; Parity</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0 - No Parity</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 - &quot;Space&quot; Parity</td>
</tr>
<tr>
<td>2</td>
<td>When set, two stop bits are sent in the Packet, otherwise one</td>
</tr>
<tr>
<td>0-1</td>
<td>Number of Data Bits sent in a Packet</td>
</tr>
<tr>
<td></td>
<td>B1 B1</td>
</tr>
<tr>
<td>0</td>
<td>0 0 - 5 Bits</td>
</tr>
<tr>
<td>0</td>
<td>0 1 - 6 Bits</td>
</tr>
<tr>
<td>1</td>
<td>0 0 - 7 Bits</td>
</tr>
<tr>
<td>1</td>
<td>1 1 - 8 Bits</td>
</tr>
</tbody>
</table>

### PC Serial Port Modem Control Register (Base + 4)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-7</td>
<td></td>
<td>Unused, normally set to zero</td>
</tr>
<tr>
<td>4</td>
<td>Loop</td>
<td>When Set, Data from the transmitter is looped internally to the receiver</td>
</tr>
<tr>
<td>3</td>
<td>Out2</td>
<td>When Set, Interrupt Requests from the 8250 are unmasked</td>
</tr>
<tr>
<td>2</td>
<td>Out1</td>
<td>This bit/pin is not controlling any hardware features in the serial port</td>
</tr>
<tr>
<td>1</td>
<td>_RTS</td>
<td>When this bit is Reset, the RTS line is at 'Mark' State</td>
</tr>
<tr>
<td>0</td>
<td>_DTR</td>
<td>When this bit is Reset, the DTR line is at 'Mark' State</td>
</tr>
</tbody>
</table>
### PC Serial Port Line Status Register (Base + 5)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Unused, Normally set to zero</td>
</tr>
<tr>
<td>6</td>
<td>Set when the transmitter shift register is empty</td>
</tr>
<tr>
<td>5</td>
<td>Set when the transmitter holding register is empty</td>
</tr>
<tr>
<td>4</td>
<td>Set when the receive line is held at a space value for longer than the current packet size</td>
</tr>
<tr>
<td>3</td>
<td>This bit is set when the last character had a framing error (ie stop bit set to &quot;Space&quot;)</td>
</tr>
<tr>
<td>2</td>
<td>Set when the last character had a parity error</td>
</tr>
<tr>
<td>1</td>
<td>Set when the latest character has overrun the receiver holding register</td>
</tr>
<tr>
<td>0</td>
<td>Set when a character has been received but not read</td>
</tr>
</tbody>
</table>

### PC Serial Modem Status Register (Base + 6)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>DCD</td>
<td>When Set, an asserted DCD signal is being received</td>
</tr>
<tr>
<td>6</td>
<td>RI</td>
<td>When Set, the modem is detecting a ring on the device it is connected to</td>
</tr>
<tr>
<td>5</td>
<td>DSR</td>
<td>When Set, a DSR &quot;Mark&quot; is being received</td>
</tr>
<tr>
<td>4</td>
<td>CTS</td>
<td>When Set, a CTS &quot;Mark&quot; is being received</td>
</tr>
<tr>
<td>3</td>
<td>DCD</td>
<td>When this bit is set, the DCD line has changed state since the last check</td>
</tr>
<tr>
<td>2</td>
<td>RI</td>
<td>When set, this bit indicates that the Ring Indicator line has changed from a Mark to a Space</td>
</tr>
</tbody>
</table>
Interrupts

To enable Interrupts for COM1/COM3 (at Interrupt 0x00C), the following code is used:

```c
SetInt( 0x0C, SerIntHndlr );     //  Point the
SetInt( 0x0C, SerIntHndlr );     //  Interrupt
SetInt( 0x0C, SerIntHndlr );     //  Handler to
SetInt( 0x0C, SerIntHndlr );     //  the Correct
SetInt( 0x0C, SerIntHndlr );     //  Handler

Dummy = inp( RxHoldingRegister );//  Turn Off any
SetInt( 0x0C, SerIntHndlr );     //  Pending
SetInt( 0x0C, SerIntHndlr );     //  Interrupts
SetInt( 0x0C, SerIntHndlr );     //  in
SetInt( 0x0C, SerIntHndlr );     //  Controller
SetInt( 0x0C, SerIntHndlr );     //  Request
SetInt( 0x0C, SerIntHndlr );     //  Interrupts
SetInt( 0x0C, SerIntHndlr );     //  on TxHolding
SetInt( 0x0C, SerIntHndlr );     //  Register
SetInt( 0x0C, SerIntHndlr );     //  Empty and Rx
SetInt( 0x0C, SerIntHndlr );     //  Holding
SetInt( 0x0C, SerIntHndlr );     //  Register Pull

outp( ModemControlRegister, inp( ModemControlRegister ) | Out2);  // Unmask
outp( InterruptEnableRegister, 0x003 ); // Enable
outp( IntMaskRegister, inp( IntMaskRegister ) & 0x0FB ); // COM1/COM3,
outp( RxHoldingRegister );//  Interrupts in
outp( IntMaskRegister, inp( IntMaskRegister ) & 0x0FB ); // Controller
outp( RxHoldingRegister );//  Request
outp( InterruptEnableRegister, 0x003 ); // Interrupts
outp( IntMaskRegister, inp( IntMaskRegister ) & 0x0FB ); // on TxHolding
outp( RxHoldingRegister );//  Register
outp( RxHoldingRegister );//  Empty and Rx
outp( RxHoldingRegister );//  Holding
outp( RxHoldingRegister );//  Register Pull
outp( ModemControlRegister, inp( ModemControlRegister ) | Out2);  // Unmask
outp( ModemControlRegister, inp( ModemControlRegister ) | Out2);  // Interrupt
outp( ModemControlRegister, inp( ModemControlRegister ) | Out2);  // Requests
```

### PC Serial Modem Status Register (Base + 6) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DSR</td>
<td>When this bit is set, the DSR line has changed state since the last check</td>
</tr>
<tr>
<td>0</td>
<td>CTS</td>
<td>When this bit is set, the CTS line has changed state since the last check</td>
</tr>
</tbody>
</table>
Once an interrupt request is made by the hardware, control is passed to the service routine:

```c
SerIntHndlr:                     //  Serial Interrupt
//  Handler

Assume that the Interrupting COM port is identified

switch ( InterruptIDRegister ) { //  Handle the
//  Interrupt
//  Request
    case 4:                 //  Received Character
        InString[ i++ ] = RxHoldingRegister;
        break;
    case 2:                 //  TxHolding Register
        //  Empty
        TxHoldingRegister = OutString[ j++ ]; //  Send
        the
        Next
        //  Character
        break;
    default:                //  Some other kind of
        //  Interrupt
        Dummy = RxHoldingRegister; //  Clear the
        Receiving Data
} //  endswitch

InterruptControlRegister = EOI; //  Reset the
//  Interrupt
//  Controller
returnFromInterrupt;        //  Return from the
//  Interrupt.
```

**Interrupt 14h—RS-232 communications APIs**

The following APIs are available within the PC—to access and load registers as specified and execute an “int 014h” instruction.
<table>
<thead>
<tr>
<th>Function</th>
<th>Input</th>
<th>Output</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Communications</td>
<td>AH = 00h</td>
<td>AH = Line</td>
<td>Initialize the Serial port. Note, AH = 004h</td>
</tr>
<tr>
<td>Port</td>
<td>AL = Init</td>
<td>Status</td>
<td>Provides Extended Capabilities</td>
</tr>
<tr>
<td></td>
<td>DX = Port</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td>Write Character</td>
<td>AH = 01h</td>
<td>AH = Line</td>
<td>Send the Character when the Modem Handshake allows or Time Out.</td>
</tr>
<tr>
<td></td>
<td>AL = Character</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DX = Port</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td>Read Character</td>
<td>AH = 02h</td>
<td>AH = Line</td>
<td>Wait for the Character to be Received when the Modem Handshake allows or Time Out.</td>
</tr>
<tr>
<td></td>
<td>DX = Port</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AL = Character</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td>Status Request</td>
<td>AH = 03h</td>
<td>AH = Line</td>
<td>Return the Current Serial Port Status.</td>
</tr>
<tr>
<td></td>
<td>DX = Port</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AL = Modem</td>
<td>Status</td>
<td></td>
</tr>
</tbody>
</table>
Extended Port Initialize

AH = 04h  AL = 0 for no Break, 1
AH = Line  AL = Modem
This is a more Status Break
Port Status Port Initialize.
Initialize For Sending

BH = Parity
0 - No Parity
1 - Odd
2 - Even
3 - Odd
Stick Parity
4 - Even
Stick Parity
BL = Stop
Bits
0 - One
1 - Two
CH = Word Length
5 - 5 Bits
2 - 7 Bits
3 - 8 Bits
<table>
<thead>
<tr>
<th>Function</th>
<th>Input</th>
<th>Output</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL = Data Rate</td>
<td>0 - 110 bps</td>
<td>1 - 150 bps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 - 300 bps</td>
<td>3 - 600 bps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 - 1200 bps</td>
<td>5 - 2400 bps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 - 4800 bps</td>
<td>7 - 9600 bps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 - 19200 bps</td>
<td>DX = Port</td>
<td></td>
</tr>
<tr>
<td>DX = Port</td>
<td>AH = 05h</td>
<td>BL = Modem</td>
<td>Return the Contents of the</td>
</tr>
<tr>
<td>Read Modem</td>
<td>AL = 0</td>
<td>Register</td>
<td>Modem Control Register</td>
</tr>
<tr>
<td>Control</td>
<td>DX = Port</td>
<td>Control</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td></td>
<td>Control</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register</td>
<td></td>
</tr>
<tr>
<td>Write to</td>
<td>AH = 05h</td>
<td>AH = Line</td>
<td>Set the Modem</td>
</tr>
<tr>
<td>Modem</td>
<td>AL = 1</td>
<td>Status</td>
<td>Control</td>
</tr>
<tr>
<td>Control</td>
<td>BL = New</td>
<td>AL = Modem</td>
<td>Register to a</td>
</tr>
<tr>
<td>Register</td>
<td>Modem</td>
<td>Status</td>
<td>New State.</td>
</tr>
<tr>
<td>Control</td>
<td>Control</td>
<td>Register</td>
<td></td>
</tr>
<tr>
<td>Control Register</td>
<td>Register</td>
<td>Value</td>
<td></td>
</tr>
</tbody>
</table>
454 Chapter 10

Parallel Port

The parallel port is the first device that most people look to when simple I/O expansion must be implemented in the PC. The parallel port itself is very simple; the design used in the PC/AT consists of just seven TTL chips and provides a simple, byte-wide parallel bidirectional interface into the PC.

Block diagram/connector

PC Parallel Port can be blocked out as shown in Fig. 10.12. The Parallel Port Connector is shown in Fig. 10.13. The Pinout for the Connector is
## DB-25 (Female)

![IBM PC DB-25 Parallel Port Connector](image)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Label</th>
<th>Direction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>_STROBE</td>
<td>Output</td>
<td>Negative Active</td>
</tr>
<tr>
<td>2</td>
<td>D0</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 0</td>
</tr>
<tr>
<td>3</td>
<td>D1</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 1</td>
</tr>
<tr>
<td>4</td>
<td>D2</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 2</td>
</tr>
<tr>
<td>5</td>
<td>D3</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 3</td>
</tr>
<tr>
<td>6</td>
<td>D4</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 4</td>
</tr>
<tr>
<td>7</td>
<td>D5</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 5</td>
</tr>
<tr>
<td>8</td>
<td>D6</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 6</td>
</tr>
<tr>
<td>9</td>
<td>D7</td>
<td>Bi-Directional</td>
<td>Parallel Data Bit 7</td>
</tr>
<tr>
<td>10</td>
<td>_ACK</td>
<td>Input</td>
<td>Pulsed Low When Data Accepted</td>
</tr>
<tr>
<td>11</td>
<td>BUSY</td>
<td>Input</td>
<td>High while Printer cannot accept another Character</td>
</tr>
<tr>
<td>12</td>
<td>NOPAPER</td>
<td>Input</td>
<td>High Indicates that Printer has run out of Paper</td>
</tr>
<tr>
<td>13</td>
<td>SELECTED</td>
<td>Input</td>
<td>High Indicates Printer is Active and Selected</td>
</tr>
</tbody>
</table>
456  Chapter 10

14  AUTOFEED OC/Output Forces Printer to Eject the Current Page when pulled low
15  ERROR Input Low Indicates Printer cannot Print any more Characters
16  INIT OC/Output Low Resets the Printer
17  SELECT OC/Output Low Indicates Printer is about to be Written to
18-25 Ground N/A Signal Ground

Base registers

The installed Parallel Ports can be read at address 0x00040:0x00008 and 0x040:0x0000C. The common Parallel Port addresses are

<table>
<thead>
<tr>
<th>Port</th>
<th>Base Address</th>
<th>Interrupt Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPT1</td>
<td>0x0378/0x03BC</td>
<td>0x00F/0x00D</td>
</tr>
<tr>
<td>LPT2</td>
<td>0x0378</td>
<td>0x00F</td>
</tr>
<tr>
<td>LPT3</td>
<td>0x0278</td>
<td>0x00D</td>
</tr>
</tbody>
</table>

Registers

Printer Port Data Register (Base Offset + 0)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>Data Bits. Normally Output, can be set to Input for Bi-Directional Operation by setting bit 5 of the &quot;Control Register&quot;</td>
</tr>
</tbody>
</table>
### PC Interfaces

#### Printer Port Control Register (Base Offset + 2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>Undefined</td>
</tr>
<tr>
<td>5</td>
<td>Set to put data pins in &quot;Input Mode&quot;</td>
</tr>
<tr>
<td>4</td>
<td>Set to enable Printer Interrupt Requests from &quot;ACK&quot; Pin. Can be Read back</td>
</tr>
<tr>
<td>3</td>
<td>_SELECT. Set to make &quot;_SELECT&quot; Pin Active (Low). Can be Read back</td>
</tr>
<tr>
<td>2</td>
<td>_INIT. Reset to make &quot;_INIT&quot; Pin Active (Low - Initialize Printer). Pin is NOT Inverted. Can be Read back</td>
</tr>
<tr>
<td>1</td>
<td>_AUTOFEED. Set to make &quot;_AUTOFEED&quot; Pin Active (Low) and current page ejected. Can be Read back</td>
</tr>
<tr>
<td>0</td>
<td>_STROBE. Set to make &quot;_STROBE&quot; Pin Active (Low). Can be Read back</td>
</tr>
</tbody>
</table>

#### Printer Port Status Register (Base Offset + 1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>BUSY Pin Data Passed to Parallel Port</td>
</tr>
<tr>
<td>6</td>
<td>_ACK. When Low, &quot;_ACK&quot; is active</td>
</tr>
<tr>
<td>5</td>
<td>NOPAPER. When High, Printer is out of Paper</td>
</tr>
<tr>
<td>4</td>
<td>SELECTED. When High, Printer is responding that it is Selected</td>
</tr>
<tr>
<td>3</td>
<td>_ERROR. When Low, &quot;_ERROR&quot; is active</td>
</tr>
<tr>
<td>2-0</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
458 Chapter 10

Data output waveform

BIOS interfaces

The Printer Status Byte passes back information from the printer port (along with program status information) via the "AH" register during Parallel Port BIOS Calls:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Not Busy</td>
</tr>
<tr>
<td>6</td>
<td>Acknowledge</td>
</tr>
<tr>
<td>5</td>
<td>Out of Paper</td>
</tr>
<tr>
<td>4</td>
<td>Selected</td>
</tr>
<tr>
<td>3</td>
<td>Error</td>
</tr>
<tr>
<td>0</td>
<td>Time-Out</td>
</tr>
</tbody>
</table>

To Enable one of the Parallel Port BIOS Requests, an "int 017h" instruction is executed with the following Register specifications:
<table>
<thead>
<tr>
<th>Function</th>
<th>Input</th>
<th>Output</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Character</td>
<td>AH = 00h</td>
<td>AH = Status</td>
<td>Send the Specified Character. If the Printer is not present or not working, the “Time-Out” Bit will be set.</td>
</tr>
<tr>
<td></td>
<td>AL = Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DX = Printer Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize</td>
<td>AH = 01h</td>
<td>AH = Status</td>
<td>Initialize the Printer Port and Printer connected to it.</td>
</tr>
<tr>
<td>Printer Port</td>
<td>DX = Printer Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Request</td>
<td>AH = 02h</td>
<td>AH = Status</td>
<td>Return the Current Printer Status.</td>
</tr>
<tr>
<td></td>
<td>DX = Printer Number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This page intentionally left blank.
Jumping outside the Current Page

The general case, low-end PICmicro® MCU interpage "goto" is

```
movf STATUS, w    ; Going to Modify the High
      Three Bits
andlw 0x01F        ; of the STATUS Word
iorlw (Label << 4) & 0x0E0
movwf STATUS
movf STATUS, w    ; Going to Modify the High
      Three Bits
andlw 0x01F        ; of the STATUS Word
iorlw (Label << 4) & 0x0E0
movwf STATUS
goto (Label & 0x01FF) | ($ & 0x0E00)
```
462 Chapter 11

The mid-range and PIC17Cxx interpage “goto” is

<table>
<thead>
<tr>
<th>Mid-Range/PIC17Cxx General Case Table Jump Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>movlw  HIGH Label              ; Get the Current 256 Instruction Block</td>
</tr>
<tr>
<td>movwf  PCLATH                   ; Store it so the Next Jump is Correct</td>
</tr>
<tr>
<td>goto (Label &amp; 0x07FF)</td>
</tr>
</tbody>
</table>

The PIC18Cxx interpage “goto” (and “call”) can jump to anywhere within the PICmicro® MCU program memory space. If a “branch always” (“BRA”) instruction is to be used, the PIC8Cxx code would be

<table>
<thead>
<tr>
<th>PIC18Cxx General Case Table Jump Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>movlw  UPPER Label                  ; Get the Current 256 Instruction Block</td>
</tr>
<tr>
<td>movwf  PCLATU                       ; Store it so the Next Jump is Correct</td>
</tr>
<tr>
<td>movlw  HIGH Label</td>
</tr>
<tr>
<td>movwf  PCLATH</td>
</tr>
<tr>
<td>bra (Label &amp; 0x07FF)</td>
</tr>
</tbody>
</table>

If a “call” to a subroutine in another page is implemented, make sure that PCLATH (and PCLATU) is restored upon return from the call.

Tables

The general case low-end PICmicro® MCU table code is
Useful Code “Snippets” 463

Table1 ; Return Table Value for Contents of “w” ; Anywhere in PICmicro® MCU Memory
         movwf Temp ; Save the Table Index
         movf STATUS, w ; Going to Modify the High Three Bits
         andlw 0x01F ; of the STATUS Word
         iorlw HIGH ((TableEntries << 4) & 0x0E0)
         movwf STATUS
         movlw LOW TableEntries ; Instruction Block
         addwf Temp, w ; Compute the Offset within the 256
         movwf PCL ; Write the correct address to the ; Program Counter

TableEntries
dt "Table", 0

Note that in the low-end PICmicro® MCU case, the index to “TableEntries” should never be in the second 256 instructions of a page.
The general case mid-range and PIC17Cxx table code is

Table2 ; Return Table Value for Contents of “w” ; Anywhere in PICmicro® MCU Memory
         movwf Temp ; Save the Table Index
         movlw HIGH TableEntries ; Instruction Block
         movwf PCLATH ; Store it so the Next Jump is Correct
         movf Temp, w ; (“movfp Temp, WREG” in PIC17Cxx)
addlw LOW TableEntries ; Instruction Block
btfsc STATUS, C
incf PCLATH, f        ; If in next, increment PCLATH
movwf PCL              ; Write the correct address to the Program Counter
movwf  TableEntries    ;   Anywhere in PICmicro® MCU Memory
dt  "Table", 0

The PIC18Cxx requires that the index be multiplied by two before PCL is changed and requires both the "PCLATU" and "PCLATH" registers to be updated:

movlw  UPPER
movwf  PCLATU     ; Store it so the Next Jump is Correct
movlw  HIGH TableEntries
movwf  PCLATH
bcf    STATUS, C
rlcf   Temp, w    ; Multiply Index by 2
btfss  STATUS, C  ; If Carry Set, Increment PCLATH/PCLATU

goto  TableSkip1
infsnz PCLATH, f
infsnz PCLATU, f

TableSkip1
addlw  LOW TableEntries
btfss  STATUS, C  ; Increment PCLATH/PCLATU if necessary

goto  TableSkip2
infsnz PCLATH, f
infsnz PCLATU, f
Useful Code “Snippets” 465

```assembly
movwf PCL    ; Write the correct address to
             ;   Program Counter
TableEntries
dt    "Table", 0
```

Conditional Branching

The following table shows the code used for different comparisons and jumping on Specific Conditions. Note that both variable and constant values are included.

<table>
<thead>
<tr>
<th>Jump “if” Condition to Check</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A == B A - B = 0</td>
<td>movf A, w/movlw A</td>
</tr>
<tr>
<td></td>
<td>subwf B, w/sublw B</td>
</tr>
<tr>
<td></td>
<td>btfsc STATUS, Z</td>
</tr>
<tr>
<td></td>
<td>goto Label            ; Jump if Z = 1</td>
</tr>
<tr>
<td>A != B A - B = 0</td>
<td>movf A, w/movlw A</td>
</tr>
<tr>
<td></td>
<td>subwf B, w/sublw B</td>
</tr>
<tr>
<td></td>
<td>btfss STATUS, Z</td>
</tr>
<tr>
<td></td>
<td>goto Label            ; Jump if Z = 0</td>
</tr>
<tr>
<td>A &gt; B B - A &lt; 0</td>
<td>movf B, w/movlw A</td>
</tr>
<tr>
<td></td>
<td>subwf B, w/sublw B</td>
</tr>
<tr>
<td></td>
<td>btfss STATUS, C</td>
</tr>
<tr>
<td></td>
<td>goto Label            ; Jump if C = 0</td>
</tr>
<tr>
<td>A &gt;= B A - B &gt;= 0</td>
<td>movf B, w/movlw B</td>
</tr>
<tr>
<td></td>
<td>subwf A, w/sublw B</td>
</tr>
<tr>
<td></td>
<td>btfsc STATUS, C</td>
</tr>
<tr>
<td></td>
<td>goto Label            ; Jump if C = 1</td>
</tr>
</tbody>
</table>
Time Delays

Here is a simple, generic delay of zero to 777 cycles as a macro.

\[
\text{DlayMacro Macro Cycles ; Delay Macro for Edges}
\]
\[
\text{variable i, TCycles, Value, TFlag}
\]
\[
\text{Value = 1 << 7}
\]
\[
\text{i = 7}
\]
\[
\text{TFlag = 0}
\]
\[
\text{if (TCycles > 5)}
\]
\[
\text{while (i >= 0)}
\]
\[
\text{if ((TFlag == 0) && ((Value * 3) <= TCycles))}
\]
\[
\text{bsf DlayCount, i}
\]
\[
\text{TFlag = 1}
\]
\[
\text{TCycles = TCycles - (Value * 3)}
\]
\[
\text{else}
\]
\[
\text{if ((TFlag != 0) && (((Value * 3) + 1) <= TCycles))}
\]
\[
\text{bsf DlayCount, i}
\]
\[
\text{TCycles = TCycles - ((Value * 3) + 1)}
\]
\[
\text{endif}
\]
\[
\text{endif}
\]
\[
\text{Value = Value >> 1}
\]
\[
\text{i = i - 1}
\]
\[
\text{endw}
\]
\[
\text{if (TCycles > 3)}
\]
\[
\text{Error "Delay Cycles too Large for Macro"
\]
\[
\text{endif}
\]
Below is a 16-bit Delay. Each loop iteration requires five instruction cycles and the delay can be defined as:

\[
\text{Delay} = \frac{\text{InstructionCycleDelay}}{5}
\]

Note that in the variable initialization, 256 is added to the “InstructionCycleDelay” to take into account the loop when the low byte is initially set.

```
        movlw    HIGH ((InstructionCycleDelay / 5) + 256)
        movwf    HiCount
        movlw    LOW ((InstructionCycleDelay / 5) + 256)
        Dlay:
            addlw    0x0FF        ; Decrement the Counter by 1
            btfsc   STATUS, Z
            decfsz   HiCount, f  ; Decrement the High Byte Counter
            goto    Dlay
```

**Negating the Contents of a Register**

Converting the contents of a File Register to its 2’s complement value without affecting "w" is simply accomplished by:

```
decfsz   HiCount, f
        goto    $ - 1
endif
while (TCycles > 1)
        goto    $ + 1
        TCycles = TCycles - 2
endw
if (TCycles == 1)
        nop                  ; Delay the Last Cycle
endif
endm
```
468  Chapter 11

comf  Reg, f           ;  Invert the bits in the
incf  Reg, f           ;  Add One to them to turn
                         into 2's
                         ;  Complement

This code should not be used on any special hardware
control registers.

The "w" register can be negated in the low-end
PICmicro® MCU using the instructions:

    addwf  Reg, w   ;   w = w + Reg
    subwf  Reg, w   ;   w = Reg - w
                         ;   w = Reg - ( w + Reg )
                         ;   w = -w

Any file register can be used for this code because its
contents are never changed.

In mid-range PICmicro® MCUs, the single instruction:

    sublw  0       ;   w = 0 - w

could be used.

Incrementing/Decrementing "w"

The following assembly language code can be used
to increment/decrement "w" in low-end PICmicro®
MCUs that do not have "addlw" and "sublw" instruc-
tions.
"Reg" can be any register that does not change during the execution of the three instructions. For the low-end parts, any file register can be used because there is no danger of them being updated by an interrupt handler.

To increment:

```
xorlw 0x0FF    ; Get 1s Complement of Number
addwf Reg, w   ; w = Reg + (w^0x0FF)
subwf Reg, w   ; w = Reg + ((Reg + (w^0x0FF))^0x0FF)
               ; w = w + 1
```

To decrement, the instructions are rearranged:

```
subwf Reg, w   ; w = Reg + (2^0x0FF) + 1
xorlw 0x0FF    ; Get 1s Complement of Result
addwf Reg, w   ; w = w + 1
```

Rotating a Byte in Place

These two lines will rotate the contents of a file register without losing data in the "Carry Flag". Rotates right and left can be implemented with this snippet. Note that the carry flag is changed.

```
rlf  Register, w   ; Load Carry with the high bit
rlf  Register, f   ; Shift over with high bit going low
```
Copy Bits from One Register to Another

Here is a fast way to save specific bits from one register into another.

```assembly
movf Source, w
xorwf Destination, w
andlw B'xxxxxxxx'   ; Replace "x" with "1" to Copy the Bit
xorwf Destination, f
```

Converting a Nybble to ASCII

The most obvious way of doing this is

```assembly
NybbletoASCII
    addwf PCL, f            ; Add the Contents of the Nybble to PCL/
dt    "0123456789ABCDEF" ; return the ASCII as a Table Offset
```

Another way is

```assembly
NybbletoASCII    ; Convert a Nybble in "w" to ASCII
    addlw 0x036       ; Add '0' + 6 to Value
    btfsc STATUS, DC ; If Digit Carry Set, then 'A' - 'F'
    addlw 7           ; Add Difference Between '9' and 'A'
```
Converting an ASCII Byte to a Hex Nybble

Using the aspect that the high nybble of ASCII “A” to “F” is 16 greater than the high nybble of “0” to “9”, a value is conditionally added to make the result 0x000 to 0x00F.

ASCIItoNybble

```
addlw 0x0C0      ; If "A" to "F", Set the Carry Flag
btfss STATUS, C  ; If Carry Set, then 'A' - 'F'
addlw 7          ; Add Difference Between '9' and 'A'
addlw 9
return            ; Return the ASCII of Digit in "w"
```

Note that ASCII characters other than “0” to “9” and “A” to “F” will result in an incorrect result.

Using T0CKI as an Interrupt Source Pin

The following code will reset TMR0 when rising edge is received.

```
movlw B'11000000' ; First Setup with Instruction Clock option as TMR0 Source
movlw B'11100000' ; Option Setup for T0CKI TMR0 Source
```
472 Chapter 11

 clr f TMR0 ; Set TMR0 to 0x0FF
 decf TMR0, f

 option ; Enable Timer on Outside Interrupt
 ; Edge
 ; NOTE - Executing this Instruction
 ; after "decf" will Load the
 ; Synchronizer with a "1"

 btfs c TMR0, 1 ; Wait for incoming Rising Edge
 goto $ - 1

 ; When Execution Here, the Input has toggled

This code can also be used on a low-end PICmicro® MCU to monitor when an input changes instead of continuously polling the input pin.

Dividing by Three

Here is an algorithm from Andy Warren for dividing a positive value by three; by knowing that "divide by three" can be represented by the series:

\[ \frac{x}{3} = \frac{x}{2} - \frac{x}{4} + \frac{x}{8} - \frac{x}{16} + \frac{x}{32} - \frac{x}{64} \ldots \]

it can be implemented in the PICmicro® MCU as:

Div3: ; Divide Contents of "w" by 3
    movwf Dividend
    clrf Quotient
    Div3_Loop ; Loop Until the Dividend == 0
Useful Code “Snippets” 473

bcf STATUS, C
rrf Dividend, f  ; Dividend /2 (ie “x/2” in Series)
movf Dividend, w  ; Is it Equal to Zero?
btsec STATUS, Z
  goto Div3_Done    ; If it is, then Stop
addwf Quotient     ; Add the Value to the Quotient
rrf Dividend, f  ; Dividend /2 (ie “x/4” in Series)
movf Dividend, w
btsec STATUS, Z
  goto Div3_Done
subwf Quotient, f ; Quotient = Quotient-(Dividend /4)
goto Div3_Loop

Div3_Done
  movf Quotient, w  ; Return the Quotient
  return

Sixteen-Bit Pulse Measurement with 5-Cycle Delay

The code that measures the pulse width for a “high” pulse is

clrf PulseWidth         ; Reset the Timer
clrf PulseWidth + 1
btfss PORTn, Bit        ; Wait for the Pulse to
goto $ - 1
  go high
incfsz PulseWidth, f    ; Increment the Counter
dece PulseWidth + 1, f   ; Loop while Still High
btsec PORTn, Bit
  goto $ - 3


474 Chapter 11

Detect a Change in a Register

This code can be used to detect changes in the I/O ports, timers, or other registers that can be updated externally to the software execution.

Test a Byte within a Range

Code that tests “Num” to be within a specific byte range and jumps to the “in_range” label if true.
Useful Code “Snippets” 475

Convert ASCII to Upper Case

This is a practical application of the previous snippet.

ToUpper:
addlw  255 - 'z'     ; Get the High limit
addlw  'z' - 'a' + 1 ; Add Lower Limit to Set Carry
btfss STATUS, C     ; If Carry Set, then Lower Case
             ; Carry NOT Set, Restore
addlw h'20'         ; Character
addlw 'A'           ; Add 'A' to restore the
return

Swap the Contents of “w” with a Register

Fast method of exchanging “w” with a register without requiring a third “temporary” file register.

xorwf Reg, f   ; w = w, Reg = Reg ^ w
xorwf Reg, w   ; w = w ^ (Reg ^ w), Reg = Reg ^ w
xorwf Reg, f   ; w = Reg, Reg = Reg ^ w ^ Reg
               ; w = Reg, Reg = Reg

Swap the Contents of Two Registers

Here is a fast snippet to swap the contents of two file registers:

movf    X, w
subwf   Y, w     ; W = Y - X
Chapter 11

476

```assembly
addwf X, f     ;  X = X + (Y - X)
subwf Y, f     ;  Y = Y - (Y - X)
```

**Compare and Swap if Y < X**

This snippet is useful for “Bubble” Sort Routines.

```assembly
movf X, w
subwf Y, w     ;  Is Y >= X?
btsc STATUS, C ;  If Carry Set, Yes
goto $ + 2     ;  Don’t Swap
addwf X, f     ;  Else, X = X + (Y - X)
subwf Y, f     ;  Y = Y - (Y - X)
```

**Counting the Number of “1”s in a Byte**

The code below is Dmitry Kirashov's optimization of the classic problem of counting the number of “1”s in a byte in 12 instructions/12 cycles.

```assembly
;  (c) 1998 by Dmitry Kirashov
rrf X, w       ;  "X" Contains Byte
andlw 0x55     ;  -a-c-e-g
subwf X, f     ;  ABCDEFGH
      ;  where AB=a+b, etc.
      ;  the same trick as in example_1
movwf X
andlw 0x33     ;  -CD--GH
addwf X, f     ;  0AB00EF0
rrf X, f       ;  00CD00CH
```
Generating Parity for a Byte

At the end of the routine, bit 0 of “X” will have the “Even” Parity bit of the original number. “Even” Parity means that if all the “1”s in the byte are summed along with Parity Bit, an even number will be produced.

Keeping a Variable within a Range

Sometimes when handling data, you will have to keep integers within a range. The four instructions below will make sure that the variable “Temp” will always be in the range of Zero to “Constant”.

```Assembly
addwf X, f    ; 0AB00EF0
rrf X, f      ; 0CD00GH0
swapf X, w
addwf X, w
andlw 0x0F    ; Bit Count in "w"
swapf X, w
xorwf X, f
rrf X, w
xorwf X, f
btfsc X, 2
incf X, f
```
Swapping Bit Pairs

; (c) 1998 by Dmitry Kirashov

movwf X ; Save the Incoming Byte in a temporary register
andlw 0x055 ; w = X = ABCDEFGH
addwf X, f ; X = ABCDEFGH + 0B0D0F0H
rrf X, f ; X = (ABCDEFGH + 0B0D0F0h) >> 1
addwf X, w ; w = BADCFEHG

Bitwise Operations

Setting a bit by “ANDing” two others together is accomplished by:

bsf Result ; Assume the result is True
btfsc BitA ; If BitA != 1 then result is False
btfsc BitB ; If BitB == 0 then result is False
bcf Result ; Result is False, Reset the Bit

“ORing” two bits together is similar to the “AND” operation, except the result is expected to be false and when either bit is set, the result is true:
Useful Code “Snippets” 479

bcf  Result ; Assume the result is False
btfss BitA ; If BitA != 0 then result is True
btfsc BitB ; If BitB == 0 then result is False
bsf  Result ; Result is True, Set the Bit

There are two ways of implementing the “NOT” operation based on where the input value is relative to the output value. If they are the same (i.e., the operation is to complement a specific bit), the code to be used is simply:

movlw (1 << BitNumber) ; Complement Specific Bit for “NOT”
xorwf BitRegister, f

If the bit is in another register, then the value stored is the complement of it:

bcf  Result ; Assume that the Input Bit is Set
btfss Bit ; - If it is Set, then Result Correct
bsf  Result ; Input Bit Reset, Set the Result

Constant Multiplication

The following macro will insert 8-bit multiplication by a constant code:

multiply macro Register, ; Multiply 8 bit value by a Value variable i = 0, ; constant TValue
480  Chapter 11

TValue = Value           ; Save the Constant Multiplier
movf   Register, w      ; Use "Temporary" as Shifted
movf   Temporary, w     ; Value
clrf   Temporary + 1
clrf   Product + 1
while (i < 8)           ; If LSB Set, Add the Value
  if ((TValue & 1) != 0)
    movf   Temporary + 1, w
    addwf  Product + 1, f
    movf   Temporary, w
    addwf  Product, f
    btfsc  STATUS, C
    incf  Product + 1, f
  endif
  bcf    STATUS, C       ; Shift Up Temporary
  rlf    Temporary, f
  rlf    Temporary + 1, f
  TValue = TValue >> 1    ; Shift down to check the Next
i = i + 1
endw
endm

Constant Division
The following code will return a rounded quotient for a
variable divided by a constant:

divide macro Register, Value ; Divide 8 bit value
variable i = 0, TValue        ; by a constant
TValue = 0x10000 / Value      ; Get the Constant Divider
movf   Register, w
movf   Temporary, w
clrf   Temporary
clrf   Quotient
clrf   Quotient + 1
while (i < 8)
    bcf STATUS, C ; Shift Down the Temporary
    rrf Temporary + 1, f
    rrf Temporary, f
    if ((TValue & 0x08000) != 0); If LSB Set, Add the
        Value
        movf Temporary + 1, w
        addwf Quotient + 1, f
        movf Temporary, w
        addwf Quotient, f
        btfsc STATUS, C
        incf Quotient + 1, f
    endif
    TValue = TValue << 1 ; Shift up to check the
                        Next Bit
    i = i + 1
endw
    movf Quotient + 1, w ; Provide Result Rounding
    btfsc Quotient, 7
    incf Quotient + 1, w
    movwf Quotient
endm
This page intentionally left blank.
Defining 16 Bit Numbers

16-bit numbers can have their addresses declared specifically, as in the example below:

```
RAM      equ    12          ; Start of RAM for the
                     ; PIC16C71
Reg_8    equ    RAM         ; Define the 8 Bit
Reg_16   equ    RAM + 1     ; Define the first 16
Reg2_16  equ    RAM + 3     ; Defined the 2nd 16 Bit
                     ; Register
                     ; Register
```

Copyright 2001 The McGraw-Hill Companies. Click Here for Terms of Use.
or, using the “CBLOCK” Command in MPASM with the number of bytes in the variable specified:

```
CBLOCK 12                  ; Start of RAM for the
Reg_8                      ; 16C71
Reg_16:2                   ; Define the 16 Bit
Reg2_16:2                  ; Register
ENDC
```

Increments and Decrements

Incrementing a 16-bit value in the low-end or mid-range is accomplished by:

```
incf Reg, f                ; Increment the Low byte
btfsc STATUS, Z            ; Do we have Zero
incf Reg + 1, f            ; (Multiple of 256)?
                        ; Increment High byte
                        ; (if necessary)
```

For the PIC17Cxx or a PIC18Cxx, the “infsnz” instruction is used to simplify the 16-bit increment by one instruction:

```
infsnz Reg, f              ; Increment "Reg's" Low
                        ; Byte and Skip
                        ; (Multiple of 256)?
                        ; High Byte Increment
                        ; if Result is Not
                        ; Equal to Zero
incf Reg + 1, f
```

The decrement of a 16-bit value for the PICmicro® MCUs is a four instruction (instruction cycle) process:

```
movf Reg, f                ; Set "Z" if LOW "Reg"
                        ; == 0
btfsc STATUS, Z
decf Reg + 1, f            ; If Low byte is Zero,
                        ; Decrement High
decf Reg, f
```
Addition/Subtraction

Adding a Constant to a value in the low-end and mid-range PICmicro® MCUs, that is,

\[ \text{Reg} = \text{Reg} + 0x01234 \]

is accomplished by:

```
addlw HIGH 0x01234 ; Add the high byte first
addwf Reg + 1, f
addlw LOW 0x01234 ; Add the Low Byte Next
addwf Reg, f
btfsc STATUS, C ; Don’t Inc high byte if carry is set
incf Reg + 1, f
```

In the PIC17Cxx and PIC18Cxx, the “addwf” instructions can be used to simplify the operation:

```
addlw LOW 0x01234 ; Add Low Byte First
addwf Reg, f
addlw HIGH 0x01234 ; Add High Byte Next
addwf Reg + 1, f
```

The corresponding subtraction, that is,

\[ \text{Reg} = \text{Reg} - 0x01234 \]

looks like the following code for the low-end and mid-range PICmicro® MCUs:

```
sublw HIGH 0x01234 ; Subtract the High Byte First
subwf Reg + 1, f
sublw LOW 0x01234 ; Subtract the Low Byte Next
subwf Reg, f
btfss STATUS, C ; Don’t Dec high byte if carry is set
decf Reg + 1, f
```
For the PIC17Cxx and PIC18Cxx, the "subwfb" instruction is used:

```assembly
movlw  LOW 0x01234 ; Subtract the Low Byte
bsf   STATUS, C ; Don't pass any
    "Borrow"
subwfb Reg, f ; Reg = Reg - w - !C
movlw  HIGH 0x01234
subwfb Reg + 1, f ; Reg + 1 = Reg + 1 - w
    ; - !C
```

The "addwfc" and "subwfb" enhancements can be used in all the 16-bit addition and subtraction operations given below. When using these instructions follow the same format of finding the least significant byte's result followed by the most significant byte's result, which is opposite to how the operations are carried out in the low-end and mid-range PICmicro® MCUs.

When adding to and subtracting from a 16-bit variable and storing the result in another variable in the low-end and mid-range PICmicro® MCUs:

```assembly
Destination = Source + 0x05678
```

the assembly code will look like:

```assembly
movlw HIGH 0x05678 ; Add High Byte First
addwf Source + 1, w ; Store Result in
movwf Destination + 1, f ; Destination
movlw LOW 0x05678 ; Add Low Byte Next
addwf Source, w ; Store Result
movwf Destination, f
btfsc STATUS, C ; Is the Carry Flag Set?
incf Destination + 1, f ; Yes, Increment High
```

Addition of a 16-bit variable to another 16-bit variable is similar to that of adding a Constant to a 16-bit variable.
16-Bit Numbers

If the destination is the same as one of the values, for instance:

\[ a = a + b \]

the low-end and mid-range assembly language code looks like:

\[
\begin{align*}
\text{movf} & \quad b + 1, w & \quad ; \text{Add the High Bytes} \\
\text{addwf} & \quad a + 1, f \\
\text{movf} & \quad b, w & \quad ; \text{Add the Low Bytes} \\
\text{addwf} & \quad a, f \\
\text{btfsc} & \quad \text{STATUS}, C & \quad ; \text{Add the Carry to High} \\
\text{incf} & \quad a + 1, f
\end{align*}
\]

If the Destination is different from both values to be added, for instance,

\[ c = a + b \]

the code is changed to save the sums in "w" and then store them in "c":

\[
\begin{align*}
\text{movf} & \quad a + 1, w & \quad ; \text{Add the High Bytes} \\
\text{addwf} & \quad b + 1, w \\
\text{movwf} & \quad c + 1 \\
\text{movf} & \quad a, w & \quad ; \text{Add the Low Bytes} \\
\text{addwf} & \quad b, w \\
\text{movwf} & \quad c \\
\text{btfsc} & \quad \text{STATUS}, C & \quad ; \text{Increment due to Carry} \\
\text{incf} & \quad c + 1
\end{align*}
\]

Subtraction is carried out in the same way, but care must be taken to ensure that the subtracting Register is kept straight. To implement

\[ c = a - b \]

in assembly language, the following code would be used in the low-end and mid-range PICmicro® MCUs:
Bitwise Operations on Constants and Variables

ANDing a 16-bit variable with 0x0A55A would be implemented in this way:

```assembly
andlw HIGH 0x0A55A       ; Get Value for ANDING
andwf Reg + 1, f         ; Do the High Byte
andlw LOW 0x0A55A        ; Get Value for ANDING
andwf Reg, f             ; Do the Low Byte
bcf STATUS, C            ; Clear the Carry Flag
              ; for new bit
rlf Reg, f               ; Shift the Low Byte
rlf Reg + 1, f           ; Shift High Byte with
                         ; Low Carry
```

and to shift right:

```assembly
bcf STATUS, C            ; Clear Carry Flag for
                         ; the New bit
rrf Reg + 1, f           ; Shift down the High Byte
rrf Reg, f               ; Shift Low Byte with
                         ; Valid Carry
```
Comparisons with 16-Bit Variables

At the end of this series of instructions, “w” contains Reg2 – Reg1 and “_2” contains Reg2HI – Reg1HI with the borrow result of Reg2 – Reg1.

There are six basic conditions that you can look for: Equals, Not Equals, Greater Than, Greater Than or Equal To, Less Than, Less Than or Equal To. So, to discover whether or not I have any of these conditions, the following code can be added.

For Equals and Not Equals, the value in “w” is ORed with “_2” to see if the Result is equal to zero.

iorwf _2, w               ; Is the Result == 0?

for Equals add the lines:

btfss STATUS, Z ; Execute following Code
     goto Zero_Skip ; If == 0
     ; Else, Code != 0, Skip

for Not Equals, append:

```assembly
btfsc STATUS, Z ; Execute following if
  ; != 0
goto NotZero_Skip ; Else, Code == 0, Skip
 ; Over
```

If Greater Than (the 16-bit variable is greater than the comparison value), then the result will not be less than Zero. Actually, the same code (just with a different Bit Skip) can be used to test.

For Greater Than:

```assembly
btfsc _2, 7 ; Not Negative, 16 Bit
  ; is Greater
goto NotGreater_Skip ; Else, Skip if Not
 ; Greater than
iorwf _2, w ; Is it Equal to Zero?
btfsc STATUS, z ; No, It is Greater
  ; than
Goto NotGreater_Skip ; Else, if Zero, Not
 ; Greater than
```

Note that just the most significant bit of the 16-bit difference is checked. If this bit is set (= 1), then the 16-bit variable is less than the Comparison. If it is reset (= 0), then it is greater than and you should check to see if the result is not equal to zero (or else it is equal).

For Less Than:
To check for Greater Than or Equal To, the last three lines of the code checking for Greater Than are simply erased. To check for Less Than or Equal To, the three lines from Not Equals are added before the check for less than.

Here is the complete code for compare and skip on Reg1 less than or equal to Reg2:

```assembly
movf Reg2 + 1, w ; Get the High Byte of
    ; the Result
subwf Reg1 + 1, w ; Store in a Temporary
movwf _2          ; Register
movf Reg2, w      ; Get the Low Byte
subwf Reg1, w     ; Decrement High if
btfss STATUS, C   ; Necessary
    decf _2
    iorwf _2, w ; Check for Equal to
    btfs STATUS, Z ; If Not Zero, Jump Over
    goto EqualLess_Skip ; Equals, Jump to the
    codemove
    btfs _2, 7 ; If Number is Negative,
    goto EqualLess_Skip ; execute
    else, Jump Over
```
492 Chapter 12

Multiplication

Here is multiplication that requires a separate byte for counting the iterations through “Loop”:

```
clrf    Product
clrf    Product + 1
movlw   16                  ; Operating on 16 Bits
movwf   BitCount
Loop ; Loop Here for Each Bit
    rrf     Multiplier + 1, f  ; Shift the Multiplier
    rrf     Multiplier, f      ; down
    btfss   STATUS, C          ; If the bit is set, add
    goto   Skip                ; the Multiplicand to
                        ; the “Product”
movf    Multiplicand + 1, w
    addwf   Product + 1, f
movf    Multiplicand, w
    addwf   Product, f
btfsc   STATUS, C
    incf   Product + 1, f    ; Shift up Multiplicand
Skip ; Shift up Multiplicand
    bcf     STATUS, C        ; and
            ; Loop Around
rlf     Multiplicand, f
rlf     Multiplicand + 1, f
decfsz  BitCount
goto   Loop
```

The code given below is the most efficient way of doing a 16-bit multiply with a 32-bit result. It is not immediately obvious, but it is very clever. Rather than use a
16-Bit Numbers  493

32-bit add each time the shifted data is detected, it provides a 16-bit (with valid carry) add and then shifts the data down. This Code does not change “Multiplicand”, but does change “Multiplier”.

Note that in the code, a 32-bit value for “Product” (using a “Product:5” line in the “CBLOCK” variable declare statement) is used.

clrf Product + 2 ; “Product” will be the resulting of the Operation
clrf Product + 3
movlw 16
movwf BitCount
Loop ; Loop here for Each Bit
rrf Multiplier + 1, f ; Shift the Multiplier down by one
rrf Multiplier, f
btfss STATUS, C ; If the bit is set, add the Multiplicand to the Product
goto Skip
clrf Product
movf Multiplicand + 1, w
addwf Product + 1, f
btfsc STATUS, C ; Make Sure the Carry is passed to the Next Byte
incf Product + 4, f
movf Multiplicand, w
addwf Product + 2, f
btfsc STATUS, C
incf Product + 3, f ; Make Sure Carry is Passed with the Shift
goto $ + 2
incf Product + 4, f
Skip ; Shift “Product” Down with
Both of the Multiplication routines shown here will work with positive and negative numbers.

For the PICmicro® MCUs that have built-in eight by eight multipliers (PIC17Cxx and PIC18Cxx), the code for 16-bit multiplication can be accomplished using the code:

```
clf    Product + 2          ; Clear the High-Order Bits
clf    Product + 3          ; Do the "L" Multiplication first
movf   Al, w                ; Save result
mulwf   Bl
movf   PRODL, w             ; Do the "I" Multiplication
movf   Product
mulwf   PRODH, w             ; Save the Most Significant Byte
movf   Al, w
mulwf   Bh
movf   PRODL, w             ; Save the Most Significant Byte First
adcwf   Product + 1, f
movf   PRODH, w
```
Division

The division routine provided here first finds how far the divisor can be shifted up before comparing to the quotient. The “Count” variable in this routine is a 16-bit variable that is used both to count the bits and add to the quotient. “Temp” is an 8-bit temporary Storage Variable. At the end of the division routine, “Dividend” will contain the remainder of the operation.

```assembly
clrf Quotient
clrf Quotient + 1
movlw 1 ; Initialize Count
movwf Count
clrf Count + 1
```
496 Chapter 12

StartLoop ; Find How Large
 ; "Divisor" can
 ; be
 ; If at the "top", then
 ; do
 ; the Division
 btfsc Divisor + 1, 7 ; Shift Count and
goto Loop ; Divisor Up
bcf STATUS, C
rlf Count, f
rlf Count + 1, f
rlf Divisor, f
rlf Divisor + 1, f
goto StartLoop

Loop ; Now, Take Away
 ; "Divisor"
 ; from "Dividend"
 ; If Divisor < Dividend
 ; then
 ; Don’t Take Away
movf Divisor + 1, w
subwf Dividend + 1, w
movwf Temp
movf Divisor, w
subwf Dividend, w
btfss STATUS, C
dcf Temp, f
btfsc Temp, 7 ; If "Temp" Negative
  ; then
  ; Divisor < Dividend
goto Skip
movwf Dividend ; Save the New Dividend
movf Temp, w
movwf Dividend + 1
movf Count, w ; Add Count to the
  ; Quotient
addwf Quotient + 1, f
movf Count, w
addwf Quotient + 1, f ; No Opportunity for
  ; Carry
Skip ; Shift Divisor/Count
  ; Down
This division routine is designed to only handle positive numbers—there is not a general algorithm that handles both positive and negative numbers and passes back both the quotient and remainder with the correct polarity efficiently.

A general form for a division routine (using the algorithm shown above) could be the division of the core of the pseudo-code in a bit-shift analogous algorithm to multiplication that can handle positive and negative numbers.

```
if (Dividend < 0) {
  // Change dividend to positive number
  Dividend = 0 - Dividend;
  dividendneg = 1;
} else
  dividendneg = 0;
if (Divisor < 0) {
  // Repeat with the Divisor
  Divisor = 0 - Divisor;
  divisorneg = 1;
} else
  divisorneg = 0;
Count = 0;  // Going to Count where division starts
```
Quotient = 0;               // Store the Quotient
while (( Divisor & 0x0400 ) != 0) {
    // Find the Start of the Division
    Count = Count + 1;         // Increment the Number of Bits Shifted
    Divisor = Divisor << 1;
}

while (Count != 0) {        // Now, do the Division
    if (Dividend >= Divisor) {  // A subtract can take place
        Quotient = Quotient + 2 ^ Count;
        Dividend = Dividend - Divisor;
    }
    Count = Count - 1;
    Divisor = Divisor >> 1;
}

if (Dividendneg == 1)       // Now, change the values
    if (Divisorneg == 1) {
        Quotient = 0 - Quotient;
        Remainder = 0 - Dividend;
    } else {
        Quotient = 0 - Quotient;
        Remainder = 0 - Dividend;
    } else                        // The Dividend was Positive
        if (Divisorneg == 1) {
            Quotient = 0 - Quotient;
            Remainder = Dividend;
        } else {  // The Dividend was Positive
            Quotient = Quotient;
            Remainder = Dividend;
        }
Chapter 13

PICmicro® MCU Operations Tables

The following information is based on the datasheets available at the time of printing and are meant to be used for providing a basic operating reference. Some data is not complete due to “Advanced” copies of the datasheets. “Idd”, or “intrinsic” current requirements, is the amount of current required for the base PICmicro® MCU to operate and does not include current required for peripheral functions.

I/O Pin Current Capabilities

Current Source/Sink requirements are in milli-Amperes ("mA").
<table>
<thead>
<tr>
<th>Device</th>
<th>Pin Source/Sink</th>
<th>Port Source/Sink</th>
<th>Device Source/Sink</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C5xx</td>
<td>25/25</td>
<td>25/25</td>
<td>100/100</td>
<td>GPIO used for Data</td>
</tr>
<tr>
<td>PIC14C000</td>
<td>25/25</td>
<td>200/200</td>
<td>250/300</td>
<td>I/O Pins Available for LCD Driving</td>
</tr>
<tr>
<td>PIC16C5x</td>
<td>20/25</td>
<td>40/50</td>
<td>50/150</td>
<td></td>
</tr>
<tr>
<td>PIC16C55x</td>
<td>25/25</td>
<td>100/150</td>
<td>100/150</td>
<td></td>
</tr>
<tr>
<td>PIC16C62x</td>
<td>25/25</td>
<td>200/200</td>
<td>250/300</td>
<td></td>
</tr>
<tr>
<td>PIC16C67x</td>
<td>25/25</td>
<td>200/200</td>
<td>250/300</td>
<td></td>
</tr>
<tr>
<td>PIC16F84</td>
<td>20/25</td>
<td>50/80-100/150</td>
<td>100/150</td>
<td>RA2/RA3 able to sink 60 mA</td>
</tr>
<tr>
<td>PIC16F87x</td>
<td>25/25</td>
<td>200/200</td>
<td>250/300</td>
<td></td>
</tr>
<tr>
<td>PIC17C4x</td>
<td>20/35</td>
<td>100/150</td>
<td>200/250</td>
<td></td>
</tr>
<tr>
<td>PIC18Cxx(x)</td>
<td>25/25</td>
<td>200/200</td>
<td>50/300</td>
<td></td>
</tr>
</tbody>
</table>
PICmicro® MCU Operations Tables

RC Oscillator Component Values

The following table and chart outline different Resistor/Capacitor values and current requirements for the low-end PICmicro® MCUs and the PIC16F84 using an RC oscillator. Note that RC oscillator operation can have variances up to 30% according to Microchip documentation and are only recommended for time-insensitive applications.

For the low-end PICmicro® MCUs, Table 13.1 shows different capacitor values and Fig. 13.1 shows current consumption for different operating speeds.

The PIC16F84 is a very common beginning user PICmicro® MCU. Table 13.2 shows different frequencies for different resistor/capacitor combinations and Fig. 13.2 shows the varying Idd current required for different frequencies.

### TABLE 13.1

<table>
<thead>
<tr>
<th>Capacitor (μF)</th>
<th>Resistor (kΩ)</th>
<th>Average Frequency (MHz)</th>
<th>% Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>2.2</td>
<td>4.97 MHz ± 2%</td>
<td>± 21%</td>
</tr>
<tr>
<td></td>
<td>5.6</td>
<td>2.38 MHz ± 2%</td>
<td>± 21%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2.02 MHz ± 2%</td>
<td>± 21%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>252.15 kHz ± 2%</td>
<td>± 21%</td>
</tr>
<tr>
<td>100</td>
<td>2.2</td>
<td>1.02 MHz ± 1%</td>
<td>± 11%</td>
</tr>
<tr>
<td></td>
<td>5.6</td>
<td>1.15 MHz ± 1%</td>
<td>± 11%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>964.94 kHz ± 2%</td>
<td>± 14%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>71.51 kHz ± 2%</td>
<td>± 21%</td>
</tr>
<tr>
<td>300</td>
<td>2.2</td>
<td>500 kHz ± 1%</td>
<td>± 11%</td>
</tr>
<tr>
<td></td>
<td>5.6</td>
<td>484 kHz ± 1%</td>
<td>± 11%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>257.2 kHz ± 1%</td>
<td>± 11%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>26.41 kHz ± 1%</td>
<td>± 11%</td>
</tr>
</tbody>
</table>

The frequencies are measured on DIP packages.

The percentage variation indicated here is part-to-part variation due to normal process distribution. The variation indicated is ±3 standard deviation from average value for VDD = 5 V.
Chapter 13

Figure 13.1

TABLE 13.2

<table>
<thead>
<tr>
<th>Coil</th>
<th>Res</th>
<th>Average Povn @ 25°C</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 nF</td>
<td>33k</td>
<td>4.63 MHz ± 2.2%</td>
</tr>
<tr>
<td>51k</td>
<td>3.64 MHz ± 2.2%</td>
<td></td>
</tr>
<tr>
<td>10k</td>
<td>0.54 MHz ± 1.3%</td>
<td></td>
</tr>
<tr>
<td>1.5k</td>
<td>0.25 MHz ± 1.3%</td>
<td></td>
</tr>
</tbody>
</table>

| 100 nF | 33k | 1.43 MHz ± 2.2% |
| 51k | 1.13 MHz ± 2.2% |
| 10k | 0.35 MHz ± 1.3% |
| 1.5k | 0.15 MHz ± 1.3% |
| 100k | 0.03 MHz ± 1.3% |

| 500 nF | 51k | 0.54 MHz ± 1.3% |
| 51k | 0.45 MHz ± 1.3% |
| 10k | 0.05 MHz ± 1.3% |

*Measured in POP Packages. The percentage variation indicated here is a part to part variation due to normal process variation. The variation indicated is ±1 standard deviation from average value.
PICmicro® MCU Operations Tables

The following table outlines different capacitor values for different “LP” oscillator executing frequencies using a crystal. Note that “LP” mode is active between 0 and 200 KHz only. The Idd (intrinsic) current requirements are quoted for 32.768 kHz and powered by 5 volts and are in micro-Amperes except where noted.

<table>
<thead>
<tr>
<th>Device</th>
<th>32.768 KHz</th>
<th>200 KHz</th>
<th>Idd Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C5xx</td>
<td>15 pF</td>
<td>N/A</td>
<td>10 uA</td>
</tr>
<tr>
<td>PIC16C5x</td>
<td>15 pF</td>
<td>N/A</td>
<td>32 uA</td>
</tr>
<tr>
<td>PIC16C55x</td>
<td>68-100 pF</td>
<td>15-30 pF</td>
<td>32 uA</td>
</tr>
<tr>
<td>PIC16C6x</td>
<td>33-68 pF</td>
<td>15-47 pF</td>
<td>21 uA</td>
</tr>
<tr>
<td>PIC16C62x</td>
<td>68-100 pF</td>
<td>15-30 pF</td>
<td>32 uA</td>
</tr>
<tr>
<td>PIC16F84</td>
<td>68-100 pF</td>
<td>15-33 pF</td>
<td>48 uA</td>
</tr>
<tr>
<td>PIC16F87x</td>
<td>33 pF</td>
<td>15 pF</td>
<td>48 uA</td>
</tr>
<tr>
<td>PIC18Cx</td>
<td>33 pF</td>
<td>15 pF</td>
<td>N/A</td>
</tr>
</tbody>
</table>
504 Chapter 13

XT Oscillator Operating Characteristics

The following table outlines different capacitor values for different "XT" oscillator executing frequencies using a crystal or ceramic resonator. Note that "XT" mode is active between 0 and 4 MHz. The Idd (intrinsic) current requirements are quoted at 4 MHz and are in milli-Amperes except where noted.

<table>
<thead>
<tr>
<th>Device</th>
<th>200 KHz</th>
<th>1 MHz</th>
<th>4 MHz</th>
<th>Idd Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C5xx</td>
<td>47-68 pF</td>
<td>15 pF</td>
<td>15 pF</td>
<td>0.78 mA</td>
</tr>
<tr>
<td>PIC16C5x</td>
<td>15-30 pF</td>
<td>15-30 pF</td>
<td>15 pF</td>
<td>1.8 mA</td>
</tr>
<tr>
<td>PIC16C55x</td>
<td>N/A</td>
<td>15-68 pF</td>
<td>15-68 pF</td>
<td>3.3 mA</td>
</tr>
<tr>
<td>PIC16C6x</td>
<td>33-68 pF</td>
<td>15-68 pF</td>
<td>15-33 pF</td>
<td>1.6 mA</td>
</tr>
<tr>
<td>PIC16C62x</td>
<td>33-68 pF</td>
<td>15-58 pF</td>
<td>15-68 pF</td>
<td>3.3 mA</td>
</tr>
<tr>
<td>PIC16C7x</td>
<td>33-68 pF</td>
<td>15-68 pF</td>
<td>15-33 pF</td>
<td>5 mA</td>
</tr>
<tr>
<td>PIC16F84</td>
<td>68-100 pF</td>
<td>15-33 pF</td>
<td>15-33 pF</td>
<td>4.5 mA</td>
</tr>
<tr>
<td>PIC16F87x</td>
<td>47-68 pF</td>
<td>15 pF</td>
<td>15 pF</td>
<td>2 mA</td>
</tr>
<tr>
<td>PIC17C4x</td>
<td>N/A</td>
<td>33-150 pF</td>
<td>15-68 pF</td>
<td>1.6 mA</td>
</tr>
<tr>
<td>PIC18Cxx(x)</td>
<td>47-68 pF</td>
<td>15 pF</td>
<td>15 pF</td>
<td>N/A</td>
</tr>
</tbody>
</table>

HS Oscillator Operating Characteristics

The following table outlines different capacitor values for different "HS" oscillator executing frequencies using a crystal or ceramic resonator. Note that "HS" mode is active for frequencies greater than 4 MHz. As a rule of thumb, the maximum speed for low-end and mid-range PICmicro® MCU EPROM program memory devices is 20 MHz. For Flash program memory parts, the maximum speed is usually 10 MHz, except where noted. For the PIC17Cxx, the maximum speed is 33 MHz and for the PIC18Cxx the maximum clock speed is 10 MHz. In
the PIC18Cxx, the HS clock can be multiplied by four for an actual internal clock speed of 40 MHz.

Idd (intrinsic) current requirements are taken from the maximum speed and the PICmicro® MCU powered by 5 volts. Capacitor values are in pFs and Idd current is in milli-Amperes except where noted.

<table>
<thead>
<tr>
<th>Device</th>
<th>4 MHz</th>
<th>10 MHz</th>
<th>20 MHz</th>
<th>32 MHz</th>
<th>Idd Curr</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C000</td>
<td>15-68 pF</td>
<td>10-47 pF</td>
<td>10-47 pF</td>
<td>N/A</td>
<td>4 mA</td>
</tr>
<tr>
<td>PIC16C5x</td>
<td>15 pF</td>
<td>15 pF</td>
<td>15 pF</td>
<td>N/A</td>
<td>5 mA</td>
</tr>
<tr>
<td>PIC16C55x</td>
<td>15-30 pF</td>
<td>15-30 pF</td>
<td>15-30 pF</td>
<td>N/A</td>
<td>20 mA</td>
</tr>
<tr>
<td>PIC16C6x</td>
<td>15-47 pF</td>
<td>15-47 pF</td>
<td>15-47 pF</td>
<td>N/A</td>
<td>35 mA</td>
</tr>
<tr>
<td>PIC16C62x</td>
<td>15-47 pF</td>
<td>15-30 pF</td>
<td>15-30 pF</td>
<td>N/A</td>
<td>20 mA</td>
</tr>
<tr>
<td>PIC16C7x</td>
<td>15-47 pF</td>
<td>15-47 pF</td>
<td>15-47 pF</td>
<td>N/A</td>
<td>30 mA</td>
</tr>
<tr>
<td>PIC16F84</td>
<td>15-33 pF</td>
<td>15-33 pF</td>
<td>N/A</td>
<td>N/A</td>
<td>10 mA</td>
</tr>
<tr>
<td>PIC16F87x</td>
<td>15 pF</td>
<td>15-33 pF</td>
<td>15-33 pF</td>
<td>N/A</td>
<td>20 mA</td>
</tr>
<tr>
<td>PIC17C4x</td>
<td>15-68 pF</td>
<td>15-47 pF</td>
<td>15-47 pF</td>
<td>0</td>
<td>40 mA</td>
</tr>
<tr>
<td>PIC18Cxx(x)</td>
<td>15 pF</td>
<td>15-33 pF</td>
<td>15-33 pF</td>
<td>N/A</td>
<td>45 mA</td>
</tr>
</tbody>
</table>
This page intentionally left blank.
Chapter 14

PICmicro® MCU Application Debugging Checklist
<table>
<thead>
<tr>
<th>Problem</th>
<th>Potential Causes</th>
<th>Check</th>
</tr>
</thead>
</table>
| PICmicro® MCU Application does Not Start | 1 No/Bad Power  
   a) Make Sure Vdd is between 4.5 Volts and 5.5 Volts Relative to Vss  
   b) Make Sure Vdd “ripple” is Less than 100 mV |                            |
|                         | 2 No/Bad Reset  
   a) Make Sure “MCLR” is pulled up to 4.5 Volts to 5.5 Volts  
   b) Make Sure Disabled _MCLR Pin is not pulled below Ground |                            |
|                         | 3 Missing/Bad Decoupling Capacitor  
   a) Check for 0.01 uF to 0.1 uF Capacitor Close to PICmicro® MCU’s Vdd Pin |                            |
|                         | 4 Part Orientation  
   a) Check that PICmicro® MCU Part is Installed Correctly  
   b) Make Sure the PICmicro® MCU is NOT getting very hot |                            |
|                         | 5 Oscillator Not Running  
   a) Check both the OSC1 and OSC2 Pins With an Oscilloscope or Logic Probe  
   b) If Internal Oscillator, Check Configuration Fuses For Correct Setting  
   c) Check for Present and Correct Capacitors |                            |
|                         | 6 Device Programming Incorrect  
   a) Check/Verify Device Programming  
   b) Look for I/O Pins being set high or low |                            |
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Watchdog Timer Enabled</td>
</tr>
<tr>
<td>2</td>
<td>Uninitialized Variable/Value Incorrect</td>
</tr>
<tr>
<td>3</td>
<td>Interrupt Handler NOT allowing execution exit From Handler</td>
</tr>
<tr>
<td>4</td>
<td>Variable Address Overlaid onto a Hardware I/O Register</td>
</tr>
<tr>
<td>5</td>
<td>Outputs switching too fast to see</td>
</tr>
<tr>
<td>6</td>
<td>PICmicro® MCU Device Seeks to Reset Itself Unexpectedly</td>
</tr>
<tr>
<td>7</td>
<td>Watchdog Timer Enabled</td>
</tr>
<tr>
<td>8</td>
<td>Uninitialized Variable/Value Incorrect</td>
</tr>
<tr>
<td>9</td>
<td>Interrupt Handler NOT allowing execution exit From Handler</td>
</tr>
<tr>
<td>10</td>
<td>Variable Address Overlaid onto a Hardware I/O Register</td>
</tr>
<tr>
<td>11</td>
<td>Outputs switching too fast to see</td>
</tr>
<tr>
<td>12</td>
<td>PICmicro® MCU Device Seeks to Reset Itself Unexpectedly</td>
</tr>
</tbody>
</table>

- **7a** Check I/O pins for changing between input/output states
- **7b** Check Actual Configuration Fuse Value
- **8a** Check Variable initialization at application start
- **8b** After resetting the simulated PICmicro® MCU, load file registers with a random value (such as 0x05A)
- **9a** Simulate Interrupt Handler and make sure that execution can return to Mainline before next Interrupt Request is Acknowledged
- **9b** Make sure that correct Interrupt Flag ("IF") is Reset in Handler
- **10a** Make sure that the Variable "CBLOCK" statement is in the File Register area of the PICmicro® MCU and not in the Hardware I/O Area
- **11a** Probe the Outputs using a Logic Probe or Oscilloscope
- **12a** Check Configuration Fuse Value
- **12b** Check for I/O pins changing state with Reset
<table>
<thead>
<tr>
<th>Problem</th>
<th>Potential Causes</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 High Internal Current and Inadequate Decoupling</td>
<td>2a Check for Correlation to Load drawn by PICmicro® MCU</td>
<td></td>
</tr>
<tr>
<td>3 Check for a &quot;noisy&quot; Power Supply</td>
<td>3a Check for greater than 100 mV &quot;Ripple&quot; from Power Supply</td>
<td></td>
</tr>
<tr>
<td>4 Execution Jumps Past Application End</td>
<td>4a Check Code for Subroutine past its End</td>
<td></td>
</tr>
<tr>
<td>5 Uninitialized Variable/Value Incorrect</td>
<td>5a Check Variable initialization for missed Variable</td>
<td></td>
</tr>
<tr>
<td>5b Set Variables to Random values (such as 0x05A) before starting simulation to find problem</td>
<td>5c Check Master Access to Peripheral</td>
<td></td>
</tr>
<tr>
<td>6 Incorrect Part Number</td>
<td>6a Check to see if the Part Being Actually Used has hardware registers</td>
<td></td>
</tr>
<tr>
<td>7 Incorrect Part Number</td>
<td>7b Check to see that Part Being used has correct pin programming</td>
<td></td>
</tr>
<tr>
<td>8 Incorrect Power Supply</td>
<td>8a Check for Power Supply &quot;sags&quot; when the Load is drawn</td>
<td></td>
</tr>
<tr>
<td>9 Incorrect Peripheral Hardware Not Active</td>
<td>9a Check Register Access prerequisites</td>
<td></td>
</tr>
<tr>
<td>10 Incorrect Peripheral Hardware Not Active</td>
<td>10a Check TRIS registers</td>
<td></td>
</tr>
<tr>
<td>11 Incorrect Part Number</td>
<td>11a Check Code for Subroutine past its End</td>
<td></td>
</tr>
<tr>
<td>12 Incorrect Part Number</td>
<td>12a Check Variable Initialization &amp; Precondition if the Peripheral was not programmed before starting simulation</td>
<td></td>
</tr>
<tr>
<td>13 Incorrect Power Supply</td>
<td>13a Check for Power Supply &quot;sags&quot; when the Load is drawn</td>
<td></td>
</tr>
</tbody>
</table>

*Debugging Checklist (Continued)*
That Match Source
Hardware Switching 3       3a Check the Hardware using a Logic Probe or an Oscilloscope
too fast to observe
No Output Mode 1 Incorrect TRIS 1a Check Values Saved in Specification
For I/O Pin Execution 1c “Float” Pin (disconnect from Circuit) to see if Pin is Actually in Output Mode with a Logic Probe
1b Check Causes for “No Output” Mode for I/O Pin
511
### Debugging Checklist (Continued)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Potential Causes</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Pin being held by high Current Source/Sink</td>
<td>2a &quot;Float&quot; Pin and see if State Changes are Possible with Pin Disconnected</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2b Look for Shorts to Vcc/Gnd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2c Look for Missing/Incorrect Resistors or Components</td>
<td></td>
</tr>
<tr>
<td>3 Output Changing State too Quickly to be Observed</td>
<td>3a Check output with a Logic Probe or an Oscilloscope</td>
<td></td>
</tr>
<tr>
<td>Pin Changes State Unexpectedly</td>
<td>1 Look for &quot;bcf&quot;, &quot;bsf&quot; or &quot;movf/&quot;movwf&quot; Instruction combinations that May Reset the Pin</td>
<td>1a Check Value written to I/O Port</td>
</tr>
<tr>
<td></td>
<td>1b Check Computed Values that are used to Modify Pin Values</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1c Look for Saved Port Values that are Incorrect or Inappropriate</td>
<td></td>
</tr>
<tr>
<td>2 Look for Hardware that &quot;Backdrives&quot; the Pin</td>
<td>2a &quot;Float&quot; Pin (disconnect from Circuit) to see if state is incorrect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2b Check Output Enable pins of Tri-State Drivers on the Pin's Net</td>
<td></td>
</tr>
<tr>
<td>3 Variable Address Overlayed onto a Hardware I/O Register</td>
<td>3a Make Sure that the Variable &quot;CBLOCK&quot; statement is in the File Register area of the PICmicro MCU and not in the Hardware I/O Area</td>
<td></td>
</tr>
<tr>
<td>Output Timing</td>
<td>1 Delay Calculations</td>
<td>1a Check to see if the Calculations match the Actual Output</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Not as Expected</td>
<td>1b Use the Assembler Calculator to Calculate Delays and match to Developer Values</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2a Check for Enabled Interrupts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2b Put &quot;bcf INTCON, GIE&quot; Before Timed code and &quot;bsf INTCON, GIE&quot; after</td>
<td></td>
</tr>
<tr>
<td>2 Interrupt Handler</td>
<td>2 Interrupt Handler Active during Timed Output</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Interrupt Handler Active</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Make sure Variables are not located in Hardware Register Space</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2a Check actual Register Address from Listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2b Make sure Variables are in memory space &quot;Above&quot; the Hardware Registers for all PICmicro MCU Family Devices the Application Runs on</td>
<td></td>
</tr>
<tr>
<td>3 Check Instruction Timings</td>
<td>3a Note that &quot;goto&quot;, &quot;call&quot;, &quot;return&quot; and PCL Modifications require Two Instruction Cycles</td>
<td></td>
</tr>
<tr>
<td>Register Values</td>
<td>1 Check for Interrupt Handler Active</td>
<td></td>
</tr>
<tr>
<td>Incorrect/Change Unexpectedly</td>
<td>1a Look for Instances in the Interrupt Handler when Register is Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1b Mask Interrupt Handler During Critical Periods Of Register Operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1c Use another Register in the Interrupt Handler and update Mainline’s version as appropriate</td>
<td></td>
</tr>
<tr>
<td>513 Predko Pocket Chapter 14  9/26/01  12:32 PM  Page 513</td>
<td>513</td>
<td></td>
</tr>
</tbody>
</table>
## Debugging Checklist (Continued)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Potential Causes</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED Not Lighting</td>
<td>1 LED Polarity Incorrect</td>
<td>1a Short PICmicro® MCU Pin to Ground to ensure LED Can Light</td>
</tr>
<tr>
<td></td>
<td>2 Check PICmicro® MCU Pin for NOT Changing to Output</td>
<td>2a Check PICmicro® MCU Pin as Specified Above</td>
</tr>
<tr>
<td>3 PWM Active with setting that turns off LED</td>
<td>3a Check PWM Output with an Oscilloscope or Logic Probe</td>
<td>3b Check for PWM Code Active</td>
</tr>
<tr>
<td>4 PICmicro® MCU Not Working</td>
<td>4a Check the PICmicro® MCU as Specified Above</td>
<td></td>
</tr>
<tr>
<td>5 Output Changing too fast to Observe</td>
<td>5a Check the output using an Oscilloscope</td>
<td></td>
</tr>
<tr>
<td>Button: No Response</td>
<td>1 Pin Pullup/Pull Down Incorrect</td>
<td>1a Check the Wiring of the Button to the PICmicro® MCU, Vcc and Gnd</td>
</tr>
<tr>
<td></td>
<td>2 Pin in Output Mode</td>
<td>2a Check to make sure the PICmicro® MCU I/O Pin is in &quot;Input&quot; Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2b Look for inadvertent Changes to</td>
</tr>
<tr>
<td>Issue</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>3 Output Changing</td>
<td>to quickly be observed</td>
<td></td>
</tr>
<tr>
<td>Button: Strange Response</td>
<td>3a Check Pin output using a Logic Probe or an Oscilloscope</td>
<td></td>
</tr>
<tr>
<td>1 Poor Debounce</td>
<td>1a Check for Multiple Button Presses Recognized by Software</td>
<td></td>
</tr>
<tr>
<td>1b Check Voltage Levels</td>
<td>Hardware to ensure Button Press is within 0.2 Volts from Vcc or Gnd</td>
<td></td>
</tr>
<tr>
<td>2 Interrupt Handler</td>
<td>2a Check the Interrupt Handler's operation with the Input Conditions</td>
<td></td>
</tr>
<tr>
<td>Response Incorrect</td>
<td>2b Check Voltage Levels on Hardware to ensure Button Press is within 0.2 Volts from Vcc or Gnd</td>
<td></td>
</tr>
<tr>
<td>LCD: No Output</td>
<td>1 Check Wiring</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1a Check Ground is on Pin 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1b Check Data Pins</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1c Make sure &quot;R/W&quot; line is held low during Writes</td>
<td></td>
</tr>
<tr>
<td>2 Check Contrast</td>
<td>2a Contrast Different for Different LCDs</td>
<td></td>
</tr>
<tr>
<td>3 Check Timing</td>
<td>3a Make sure that LCD &quot;E&quot; Strobes are a minimum of 450 nsecs in width</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3b Make sure signals do not change during &quot;E&quot; Strobes</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 15

PICmicro® MCU
Application Software
Development Tools

Microsoft Compatible Editor
“Ctrl” Key Combinations

<table>
<thead>
<tr>
<th>Keystrokes</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up Arrow</td>
<td>Move Cursor up one Line</td>
</tr>
<tr>
<td>Down Arrow</td>
<td>Move Cursor down one Line</td>
</tr>
<tr>
<td>Left Arrow</td>
<td>Move Cursor left one Character</td>
</tr>
<tr>
<td>Right Arrow</td>
<td>Move Cursor right on Arrow</td>
</tr>
<tr>
<td>Page Up</td>
<td>Move viewed Window Up</td>
</tr>
<tr>
<td>Page Down</td>
<td>Move viewed Window Down</td>
</tr>
<tr>
<td>Ctrl - Left Arrow</td>
<td>Jump to Start of Word</td>
</tr>
</tbody>
</table>
518 Chapter 15

Ctrl - Right Arrow  Jump to Start of next Word
Ctrl - Page Up     Move Cursor to Top of viewed Window
Ctrl - Page Down   Move Cursor to Bottom of viewed Window
Home               Move Cursor to Start of Line
End                 Move Cursor to End of Line
Ctrl - Home        Jump to Start of File
Ctrl - End         Jump to End of File
Shift - Left Arrow Increase the Marked Block by one character to the left
Shift - Right Arrow Increase the Marked Block by one character to the right
Shift - Up Arrow   Increase the Marked Block by one line up
Shift - Down Arrow Increase the Marked Block by one line down
Ctrl Shift - Left Arrow  Increase the Marked Block by one word to the left
Ctrl Shift - Right Arrow  Increase the Marked Block by one word to the Right

MPSIM.INI

A typical MPSIM.INI File for an application is

; MPSIM File for PROG2 - Turning on an LED
;
; Myke Predko - 96.05.20
;
P 84                ; Use a 16C84
SR X                ; Hex Numbers in the Simulator
ZR                  ; Zero the Registers
RE                  ; Reset Elapsed Time and Step Count
DW D                ; Disable the WDT
V W,X,2             ; Display: the "W" Register
AD F3,B,8           ; Status Register
AD F4,X,2           ; PSR Register
AD OPT,X,2          ; Option Register
AD FB,B,8           ; INTCON Register
AD P2,X,3           ; PCL Register
AD FA,X,3           ; PCLATH Register
AD F1,X,2           ; TIM0 Register
MPLAB

MPLAB is a complete "Integrated Development Environment" ("IDE") for all the different PICmicro® MCU architecture families that runs under Microsoft's "Windows" version 3.1x or later operating systems. MPLAB integrates the different operations of developing a PICmicro® MCU application. This is done from a user configurable "desk top" (see Fig. 15.1) with different capabilities built into the program.

MPLAB can integrate the following different functions:

- editor
- assemblers
- compilers
- linkers
- programmers
- emulators

The following files are accessed by MPLAB:

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.asm</td>
<td>Application Source File</td>
</tr>
<tr>
<td>.$$</td>
<td>Backup of the Application Source File</td>
</tr>
<tr>
<td>.cod</td>
<td>&quot;Label Reference&quot; for MPLAB Simulator/Emulator</td>
</tr>
</tbody>
</table>
MPLAB has the capability of displaying specific register and bit contents in the PICmicro® MCU. These windows, such as the one shown in Fig. 15.2 allow you to select the registers to monitor. To define a Watch Window or add more registers to it, the “Register Selection” Window is brought up for you to select the registers you would like to monitor. The “Properties” Window is selected from the “Register Selection” Window (as is shown in Fig. 15.3) to specify the characteristics of the register that is displayed.
The most basic input method is the "Asynchronous Stimulus" window (shown in Fig. 15.4) which consists of a set of buttons that can be programmed to drive any of the simulated PICmicro® MCU pins and can set the button to change the pin by:

Figure 15.2 Sample “Watch Window”

Figure 15.3 MPLAB stopwatch
The “Pulse” option, pulses the input pin to the complemented state and then back to the original state within one instruction cycle. This mode is useful for clocking TMR0 or requesting an external interrupt. Setting the pin “High” or “Low” will drive the set value onto the pin. To change the value of the pin between the two states, you can program two buttons in parallel with each other and each button changes the state. This can also be done with a single “Toggle” button, which changes the input state each time the button is pressed.

Clocks can be input into the simulated PICmicro® MCU by clicking on the “Debug” pull down, “Simulator Stimulus” and “Clock Stimulus…” selections. The clock stimulus dialog box (Fig. 15.5) can input regular clocks into a PICmicro® MCU by selecting the pin and then the “High” and “Low” time of the clock along with whether or not the clock is inverted (which means at reset, the clock will be low rather than high). The clock counts in Fig. 15.5 are in instruction cycles. Clock stim-
stimulus can be used for simple I/O tests, but it is really best suited for putting in repeating inputs that drive clocks or interrupts.

The “Register Stimulus” feature will store a two-digit hex value in a specified register every time a specific address is encountered in the simulated application execution. To load the operating parameters of the Register Stimulus method, the “debug” pull down is clicked, followed by “Simulator Stimulus” and then “Register Stimulus” is “Enabled”. This brings up the small window shown in Fig. 15.6 on which you will select the address of the register to change as well as the address that this happens at. Once the addresses have been specified, the register stimulus file is selected by clicking on “Browse . . .”. The register “Modify” Window (shown in Fig. 15.7) is available
by clicking on the “Window” pull down and then selecting “Modify . . .”. This window can access any register in the simulated device, including “w”, which cannot be directly addressed in the low-end and mid-range devices.
Stimulus (.STI) files

Stimulus Files require a clock “Step” specification along with the pins to be driven. Comments in the file are preceded by a “!” character. The file below is a sample stimulus file (which, by convention, always ends in ".sti"):

```
! Sample Stimulus File
!
Step   MCLR   RB4   ! Define the Bits to be Controlled
   1      1      1   ! Initialize the Bit Values
   500     0      1   ! Wait for the Program and Hardware to be Initialized
   1000    1      1   ! Reset the PICmicro® MCU
   1500    1      0   ! Change the State of the Port Bit
   2000    1      1   ! Restore it for rest of program
```

Stimulus Files are the recommended method to simulate an application and understand what are the potential software problems.
<table>
<thead>
<tr>
<th>Directive</th>
<th>Usage Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>__BADRAM</td>
<td>__BADRAM Start, End</td>
<td>Flag a range of file registers which are unimplemented</td>
</tr>
<tr>
<td>BANKISEL BANKISEL &lt;label&gt;</td>
<td>Update the &quot;IRP&quot; bit of the &quot;STATUS&quot; register before the &quot;FSR&quot; register is used to access a register indirectly. This directive is normally used with linked source files.</td>
<td></td>
</tr>
<tr>
<td>BANKSEL BANKSEL Label</td>
<td>Update the &quot;RPx&quot; bits of the &quot;STATUS&quot; register before accessing a file register directly. This directive is not available for the low-end devices (for these devices, the FSR register should be used to access the specific address indirectly). This directive is also not available for the High-end PICmicro® MCUs which should use the &quot;movlb&quot; instruction.</td>
<td></td>
</tr>
<tr>
<td>CBLOCK CBLOCK Address Var1, Var2 VarA:2</td>
<td>Used to define a starting address for variables or constants which require increasing values. To declare multiple byte variables or constants which increment by more than one, a colon (&quot;:&quot;;) is placed after the label and before the number to increment by. This is shown for &quot;VarA&quot; in the usage example. The &quot;ENDC&quot; directive is required to &quot;turn off&quot; CBLOCK operation.</td>
<td></td>
</tr>
</tbody>
</table>
| CODE CODE [Address] | Used with an object file to define the start of application code in the source file. A "Label" can be
specified before the directive to give a specific label to the object file block of code. If no "Address" is specified, then MPLINK will calculate the appropriate address for the CODE statement and the instructions that follow it.

```
__CONFIG __CONFIG Value
```

This directive is used to set the PICmicro™ MCU’s configuration bits to a specific value. "__CONFIG" automatically sets the correct address for the specific PICmicro™ MCU. The "Value" is made up of constants declared in the PICmicro™ MCU’s *.inc* file.

```
CONSTANT/ CONSTANT Label = Value
=EQU
```

Define a Constant Using one of the three formatting methods shown in usage example. The constant "Value" references to the "Label" and is evaluated when the "Label" is defined. For replacing a Label with a string, use "#DEFINE".

```
DA/DATA/DB DA Value||string
```

Set program memory words with the specified data values. If a "string" is defined, then each byte is put into its own word. The "DM" directive is recommended to be used instead of "DATA" or "DB" because its operation is less ambiguous when it comes to how the data is stored. Note that "DATA"/"DB"/"DM" do not store the data according as part of a "retlw" instruction. For the "retlw" instruction to be included with the data, the "DT" directive must be used. These directives are best suited for use in Serial EEPROM Source Files.
<table>
<thead>
<tr>
<th>Directive</th>
<th>Usage Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>ORG 0x02100</td>
<td>This instruction is used to save initialization data for the PICmicro MCU's built-in Data EEPROM. Note that an &quot;org 0x02100&quot; statement has to precede the &quot;de&quot; directive to ensure that the PICmicro MCU’s program counter will be at the correct address for programming.</td>
</tr>
<tr>
<td>#DEFINE</td>
<td>#DEFINE Label [string]</td>
<td>Specify that any time &quot;Label&quot; is encountered, it is replaced by the string. Note that string is optional and the defined &quot;Label&quot; can be used for conditional assembly. If &quot;Label&quot; is to be replaced by a constant, then one of the &quot;CONSTANT&quot; declarations should be used. This directive is placed in the first column of the source file.</td>
</tr>
<tr>
<td>DT</td>
<td>DT Value [.Value...] [&quot;string&quot;]</td>
<td>Place the &quot;Value&quot; in a &quot;retlw&quot; statement. If DT's parameter is part of a &quot;string&quot;, then each byte of the string is given its own &quot;retlw&quot; statement. This directive is used for implementing read-only tables in the PICmicro MCU.</td>
</tr>
<tr>
<td>DW</td>
<td>DW Value[.Value...]</td>
<td>Reserve program memory for the specified &quot;Value&quot;. This value will be placed in a full program memory word.</td>
</tr>
<tr>
<td>ELSE</td>
<td></td>
<td>Used in conjunction with &quot;IF&quot;, &quot;IFDEF&quot; or &quot;IFNDEF&quot; to provide an alternative path for conditional assembly. Look at these directives for examples of how &quot;ELSE&quot; is used.</td>
</tr>
<tr>
<td>Directive</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>END END</td>
<td>End the program block. This directive is required at the end of all application source files.</td>
<td></td>
</tr>
<tr>
<td>ENDC</td>
<td>Used to end the &quot;CBLOCK&quot; Label constant value saving and updating. See &quot;CBLOCK&quot; for an example of how this directive is used.</td>
<td></td>
</tr>
<tr>
<td>ENDIF</td>
<td>Used to end an &quot;if&quot; statement conditional code block. See &quot;IF&quot;, &quot;IFDEF&quot; or &quot;IFNDEF&quot; for an example of how this directive is used.</td>
<td></td>
</tr>
<tr>
<td>ENDM</td>
<td>Used to end the &quot;MACRO&quot; Definition. See &quot;CBLOCK&quot; for an example of how this directive is used.</td>
<td></td>
</tr>
<tr>
<td>ENDW</td>
<td>Used to end the block of code repeated by the &quot;WHILE&quot; Conditional Loop instruction. See &quot;WHILE&quot; for an example of how this directive is used.</td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>Force an &quot;ERROR&quot; into the code with the &quot;string&quot; message inserted into the Listing/Error Files.</td>
<td></td>
</tr>
<tr>
<td>ERRORLEVEL</td>
<td>Change the assembler's response to the specific &quot;Error&quot; (&quot;2&quot;), &quot;Warning&quot; (&quot;1&quot;) or Message (&quot;0&quot;) Number (&quot;#&quot;). Specifying &quot;-&quot; before the Number will cause any occurrences of the Error, Warning or Message to be ignored by the assembler and not reported. Specifying &quot;+&quot; before the Number will cause any occurrences of the Error, Warning or Message to be output by the Assembler.</td>
<td></td>
</tr>
</tbody>
</table>
| EXITM     | For use within a MACRO to force the stopping of the MACRO expansion. Using this directive is not
<table>
<thead>
<tr>
<th>Directive</th>
<th>Usage Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPAND</td>
<td>EXPAND</td>
<td>Enable printing MACRO Expansions in the listing file after they have been disabled by the &quot;NOEXPAND&quot; directive. Printing of MACRO Expansions is the default in MPLAB.</td>
</tr>
<tr>
<td>EXTERN</td>
<td>EXTERN Label</td>
<td>Make a program memory Label in an object file available to other object files.</td>
</tr>
<tr>
<td>FILL</td>
<td>FILL Value, Count</td>
<td>Put in &quot;Value&quot; for &quot;Count&quot; words. If &quot;Value&quot; is surrounded by parenthesis, then an instruction can be put in (ie &quot;(goto 0)&quot;). In earlier versions of MPLAB, &quot;Fill&quot; did not have a &quot;Count&quot; parameter and replaced any program memory address that does not have an instruction assigned to it or areas that is not reserved (using &quot;RES&quot;) with the &quot;Value&quot;.</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>GLOBAL Label</td>
<td>Specify a Label within an object file that can be accessed by other object files. &quot;GLOBAL&quot; is different from &quot;EXTERN&quot; as it can only be put into the source after the label is defined.</td>
</tr>
<tr>
<td>IDATA</td>
<td>IDATA [Address]</td>
<td>Used to specify a data area within an object file. If no &quot;Address&quot; is specified, then the assembler recommends printing MACRO Expansions in the case where the MACRO's execution is in error and should not continue until the error has been fixed. Using &quot;EXITM&quot; in the body of the MACRO could result in &quot;Phase Errors&quot; which can be very hard to find.</td>
</tr>
</tbody>
</table>

Recommended except in the case where the MACRO's execution is in error and should not continue until the error has been fixed. Using "EXITM" in the body of the MACRO could result in "Phase Errors" which can be very hard to find.
calculates the address. A Label can be used with
IDATA for referencing it.

__IDLOCS__

__IDLOCS__ Value

Set the four ID Locations of the PICmicro® MCU with the
four nybbles of "Value". This directive is not
available for the 17Cx devices.

IF

IF Parm1 COND Parm2

; "True" Code
ELSE
; "False" Code
ENDIF

IFDEF

IFDEF Label

; "True" Code
ELSE
; "False" Code
ENDIF

IFNDEF

IFDEF Label

; "True" Code
ELSE
; "False" Code
ENDIF

INCLUDE

INCLUDE FileName.Ext

LIST

LIST option[, ...]

Define the assembler options for the source file. The
available options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b=nnn</td>
<td>8</td>
<td>Set tab spaces.</td>
</tr>
<tr>
<td>c=nnn</td>
<td>132</td>
<td>Set column width.</td>
</tr>
<tr>
<td>Directive Usage Example</td>
<td>Comments</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td><strong>Option</strong></td>
<td><strong>Default</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>f=format</td>
<td>INHX8M</td>
<td>Set the hex file output.</td>
</tr>
<tr>
<td>free</td>
<td>FIXED</td>
<td>Use free-format parser.</td>
</tr>
<tr>
<td>fixed</td>
<td>FIXED</td>
<td>Use fixed-format parser.</td>
</tr>
<tr>
<td>mm=ON</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>n=nnn</td>
<td>60</td>
<td>Set lines per page.</td>
</tr>
<tr>
<td>p-type</td>
<td>None</td>
<td>Set PICmicro® MCU type.</td>
</tr>
<tr>
<td>r=radix</td>
<td>HEX</td>
<td>Set default radix (HEX, DEC, or OCT available)</td>
</tr>
<tr>
<td>st=ON</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>t=ON</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>w=0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>x=ON</td>
<td>OFF</td>
<td>ON</td>
</tr>
</tbody>
</table>

**LOCAL** Fillup MACRO Size
Local i
i = 0

Define a Variable that is local to a MACRO and cannot be accessed outside of the MACRO.
WHILE (i < Size)
   DW 0x015AA
   i = i + 1
ENDWENDM

MACRO Label MACRO          Define a block of code that will replace the "Label"
                      [Parm[, ...]] every time it is encountered. The optional Parameters
                      beq Parm, 0 will replace the parameters in the Macro itself.
ENDM

__MAXRAM __MAXRAM End         Define the last File Register Address in a PICmicro®
MCU that can be used.

MESSG MESSG "string"          Cause "String" to be inserted into the source file at
                             the "MESSG" statement. No errors or warnings are
                             generated for this instruction.

NOEXPAND NOEXPAND             Turn off Macro expansion in the Listing File.

NOLIST NOLIST                 Turn off Source Code Listing Output in Listing File.

ORG ORG Address               Set the Starting Address for the following code to be
                             placed at.

PAGE PAGE                    Insert a Page Break before the "PAGE" directive.

PAGESEL PAGESEL Label goto Label Insert the Instruction Page of a Label before jumping
                             to that Label or calling the subroutine at it.
<table>
<thead>
<tr>
<th>Directive</th>
<th>Usage Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCESSOR</td>
<td>PROCESSOR type</td>
<td>This directive is available for commonality with earlier Microchip PICmicro® MCU assemblers. The Processor option of the &quot;LIST&quot; directive should be used instead.</td>
</tr>
<tr>
<td>RADIX</td>
<td>RADIX Radix</td>
<td>This directive is available for commonality with earlier Microchip PICmicro® MCU assemblers. Available options are &quot;HEX&quot;, &quot;DEC&quot; and &quot;OCT&quot;. The default radix should be selected in the &quot;LIST&quot; directive instead.</td>
</tr>
<tr>
<td>RES</td>
<td>RES MemorySize</td>
<td>Reserve a block of program memory in an object file for use by another. A label may be placed before the RES directive to save what the value is.</td>
</tr>
<tr>
<td>SET</td>
<td>Label SET Value</td>
<td>&quot;SET&quot; is similar to the &quot;CONSTANT&quot;, &quot;EQU&quot; and &quot;=&quot; directives, except that the &quot;Label&quot; can be changed later in the code with another &quot;SET&quot; directive statement.</td>
</tr>
<tr>
<td>SPACE</td>
<td>SPACE Value</td>
<td>Insert a set number of blank lines into a listing file.</td>
</tr>
<tr>
<td>SUBTITLE</td>
<td>SUBTITLE &quot;string&quot;</td>
<td>Insert &quot;string&quot; on the line following the &quot;TITLE&quot; string on each page of a listing file.</td>
</tr>
<tr>
<td>TITLE</td>
<td>TITLE &quot;string&quot;</td>
<td>Insert &quot;string&quot; on the top line on each page of a listing file.</td>
</tr>
<tr>
<td>UDATA</td>
<td>UDATA [Address]</td>
<td>Declare the beginning of an uninitialized data section. &quot;RES&quot; labels should follow to mark variables in the uninitialized data space. This command is designed for Serial EEPROMs.</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Label1 RES 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Label2 RES 2</td>
<td></td>
</tr>
<tr>
<td>Label1 RES 1</td>
<td>UDATA_ACS [Address]</td>
<td>Declare the beginning of an uninitialized data section in a 18Cxx PICmicro® MCU. &quot;RES&quot; labels should follow to mark variables in the uninitialized data space.</td>
</tr>
<tr>
<td></td>
<td>Label2 RES 2</td>
<td></td>
</tr>
<tr>
<td>Label1 RES 1</td>
<td>UDATA_OVR [Address]</td>
<td>Declare the beginning of an uninitialized data section that can be overwritten by other files (as an &quot;Overlay&quot;). &quot;RES&quot; labels should follow to mark variables in the uninitialized data space. This command is designed for Serial EEPROMs.</td>
</tr>
<tr>
<td></td>
<td>Label2 RES 2</td>
<td></td>
</tr>
<tr>
<td>Label1 RES 1</td>
<td>UDATA_SHR [Address]</td>
<td>Declare the beginning of data memory that is &quot;shared&quot; across all the register banks.</td>
</tr>
<tr>
<td></td>
<td>Label2 RES 1</td>
<td></td>
</tr>
<tr>
<td>#UNDEFINE #UNDEFINE Label</td>
<td>Delete a Label that was &quot;#DEFINED&quot;.</td>
<td></td>
</tr>
<tr>
<td>VARIABLE</td>
<td>VARIABLE Label [= Value]</td>
<td>Declare an assembly-time variable that can be updated within the code using a simple assignment statement.</td>
</tr>
<tr>
<td>WHILE</td>
<td>WHILE Parm1 COND Parm2 ; while &quot;True&quot; ENDW</td>
<td>Execute code within the &quot;WHILE&quot;/&quot;ENDW&quot; directives while the &quot;Parm1 COND Parm2&quot; test is true. Note that in the listing file, the code will appear as if the code within the &quot;WHILE&quot;/&quot;ENDW&quot; directives was repeated a number of times.</td>
</tr>
</tbody>
</table>
Chapter 15

Standard Declaration
and Include (".inc") Files

The Microchip developed standard “include” files are recommended to be included into source code rather than using developer supplied register definitions. There is an ".inc" file for every PICmicro® MCU part number in the format:

`p:1>PICmicro® MCU<.inc`

where “PICmicro® MCU” is the PICmicro® MCU part number.

Linking—Linked Applications

Before creating the linked application object, the source files to be linked together have to be created. Once the files are created, "links" are created to addresses that have to be accessed between the different files. It is important to remember that variables as well as instruction addresses have to be common.

The following two example source code files show how addresses are linked together. The first is the "mainline":

```c
TITLE - Test3 - Jump to Test3A
;
; Example Application using the MPLAB Linker
;
; Myke Predko
; 2000.02.02
;
; Hardware Notes:
; PIC16F84 running in a Simulator
;```
LIST  R=DEC ; list directive to
define processor
#include "p16F84.inc" ; processor specific
variable
definitions
__CONFIG _CP_OFF & _WDT_ON & _PWRTS_ON & _XT_OSC
EXTERN TEST3A ; Specify Mainline
Location
GLOBAL TEST3AStart
GLOBAL flag ; Variable passed to
Linked File

;***** VARIABLE DEFINITIONS (examples)
; example of using Uninitialized Data Section
INT_VAR    UDATA      0x0C
w_temp    RES     1 ; variable used for
context saving
status_temp    RES     1 ; variable used for
context saving
flag    RES     2 ; temporary variable
[shared locations
- G_DATA]

;***************************************************
RESET_VECTOR CODE     0x000 ; processor reset
vector
goto     start ; go to beginning of
program
INT_VECTOR CODE 0x004 ; interrupt vector
location
movwf w_temp
PROG CODE 0x005
movf STATUS, w
movwf status_temp
; isr code can go here or be located as a call
subroutine elsewhere
movf status_temp, w ; Restore
Context
Registers
movwf STATUS
swapf w_temp, f
The second is the file that is linked to the "mainline":

```c
TITLE "Test3A - Actually Execute the Code"
;
; Example Application using the MPLAB Linker
;
; Myke Predko
; 2000.02.02
;
; Hardware Notes:
; PIC16F84 running in a Simulator
;
EXTERN flag               ; External Values
EXTERN TEST3AStart        ; Linked into Code

ing list r=dec
#include "p16f84.inc"

TEST3ACODE CODE
TEST3A
GLOBAL TEST3A              ; Address to Pass to
                             ; Linked File

banksel flag               ; example
clrf flag                  ; example

; remaining code goes here
movlw 77
movwf flag
movlw 0x001
subwf flag, f
btsw STATUS, Z

goto $ - 2
```
Application Code Template

The following file should be used as a mid-range PICmicro® MCU application source code template.

title "FileName - One Line Description"
#define _version "x.xx"

; Update History:
; 
; Application Description/Comments
; 
; Author
; 
; Hardware Notes:
; 
LIST R=DEC ; Device Specification
INCLUDE "p16cxx.inc" ; Include Files/Registers

; Variable Register Declarations
; 
; Macros

__CONFIG _CP_OFF & _XT_OSC & _PWRT_ON & _WDT_OFF & _BODEN_OFF

org    0
Mainline

org  4 ; Interrupt Handler at Address 4
Int

MainLine_Code

; Subroutines

end
The BASIC Language

BASIC variables do not have to be declared except in specialized cases. The variable name itself follows normal conventions of a letter or "_" character as the first character, followed by alphanumeric characters and "_" for variable names. Variable (and Address "Label") names may be case sensitive, depending on the version.

To Specify Data Types, a "Suffix" character is added to the end of the Variable name:

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>String Data</td>
</tr>
<tr>
<td>%</td>
<td>Integer</td>
</tr>
<tr>
<td>&amp;</td>
<td>Long Integer (32 Bits) - Microsoft BASIC Extension</td>
</tr>
<tr>
<td>!</td>
<td>Single Precision (32 Bits) - Microsoft BASIC Extension</td>
</tr>
<tr>
<td>#</td>
<td>Double Precision (64 Bits) - Microsoft BASIC Extension</td>
</tr>
</tbody>
</table>

The following table lists the different BASIC functions:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>Starting Array Element</td>
</tr>
<tr>
<td>DATA</td>
<td>Data Block Header</td>
</tr>
<tr>
<td>DIM</td>
<td>Dimension Array Declaration</td>
</tr>
<tr>
<td>OPTION</td>
<td>Starting Array Element</td>
</tr>
<tr>
<td>LET</td>
<td>Assignment Statement (Not Mandatory)</td>
</tr>
<tr>
<td>RANDOMIZE</td>
<td>Reset Random Number &quot;Seed&quot;</td>
</tr>
<tr>
<td>INPUT</td>
<td>Get Terminal Input</td>
</tr>
<tr>
<td>?</td>
<td>Output to a Terminal</td>
</tr>
<tr>
<td>READ</td>
<td>Get &quot;Data&quot; Information</td>
</tr>
<tr>
<td>GOTO</td>
<td>Jump to Line Number/Label</td>
</tr>
<tr>
<td>GOSUB</td>
<td>Call Subroutine at Line</td>
</tr>
<tr>
<td>RETURN</td>
<td>Return to Caller from Subroutine</td>
</tr>
</tbody>
</table>
**PICmicro® MCU Application Software**  541

IF Condition [THEN] Statement
Conditionally Execute the "Statement"

FOR Variable = Init TO Last [STEP Inc] ... NEXT
[Variable] Loop Specified Number of Times

ON Event GOTO
On an Event, Jump to Line Number/Label

RESTORE
Restore the "DATA" Pointer

STOP
Stop Program Execution

REM
Comment - Everything to the Right is Ignored

ABS
Get Absolute Value of a Number

SGN
Return the Sign of a Number

COS
Return Cosine of an Angle (input usually in Radians)

SIN
Return Sine of an Angle (input usually in Radians)

TAN
Return Tangent of an Angle (input usually in Radians)

ATN
Return the Arc Tangent of a Ratio

INT
Convert Real Number to Integer

SQR
Return the Square Root of a Number

EXP
Return the Power of e for the input

LOG
Return the Natural Logarithm for the Input

END
Return a Random Number

TAB
Set Tab Columns on Printer

For assignment and "if" statements, the following operators are available in BASIC:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>&quot;</td>
<td>Start/End of Text String</td>
</tr>
<tr>
<td>,</td>
<td>Separator</td>
</tr>
</tbody>
</table>
542  Chapter 15

;            Print Concatenation
$            String Variable Identifier
=            Assignment/Equals To Test
<            Less than
<=           Less than or Equals To
>            Greater than
>=           Greater than or Equals To
<>           Not Equals

BASIC's Order of Operations is quite standard for programming languages:

<table>
<thead>
<tr>
<th>Operators</th>
<th>Functions</th>
<th>Priority</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>= &lt;&gt; &lt; &lt;= &gt;&gt;=</td>
<td>Expression Evaluation</td>
<td>Highest</td>
<td>Conditional Tests</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* /</td>
<td>Multiplication/Division</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ -</td>
<td>Lowest</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addition/Subtraction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microsoft BASIC Enhancements

The following functions are available in Microsoft versions of BASIC for the PC as well as some BASICS for the PICmicro® MCU:

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>AND Logical Results</td>
</tr>
<tr>
<td>OR</td>
<td>OR Logical Results</td>
</tr>
<tr>
<td>XOR</td>
<td>XOR Logical Results</td>
</tr>
<tr>
<td>EQV</td>
<td>Test Equivalence of Logical Results</td>
</tr>
<tr>
<td>IMP</td>
<td>Test Implication of Logical Results</td>
</tr>
<tr>
<td>MOD</td>
<td>Get the Modulus (remainder) of an Integer Division</td>
</tr>
<tr>
<td>FIX</td>
<td>Convert a Floating Point Number to Integer</td>
</tr>
<tr>
<td>DEFSTR Variable</td>
<td>Define the Variable as a String (instead of the &quot;DIM&quot; Statement)</td>
</tr>
<tr>
<td>DEFINT Variable</td>
<td>Define the Variable as an Integer (instead of the &quot;DIM&quot; Statement)</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DEFLNG Variable</td>
<td>Define the Variable as a &quot;long&quot; Integer (instead of the &quot;DIM&quot; Statement)</td>
</tr>
<tr>
<td>DEFSNG Variable</td>
<td>Define the Variable as a Single Precision Floating Point Number (instead of the &quot;DIM&quot; Statement)</td>
</tr>
<tr>
<td>DEFPDBL Variable</td>
<td>Define the Variable as a Double Precision Floating Point Number (without using the &quot;DIM&quot; Statement)</td>
</tr>
<tr>
<td>REDIM Variable</td>
<td>Redefine a Variable</td>
</tr>
<tr>
<td>ERASE</td>
<td>Erase an Array Variable from Memory</td>
</tr>
<tr>
<td>LBOUND</td>
<td>Return the First Index of an Array Variable</td>
</tr>
<tr>
<td>UBOUND</td>
<td>Return the Last Index of an Array Variable</td>
</tr>
<tr>
<td>CONST Variable = Value</td>
<td>Define a Constant Value</td>
</tr>
<tr>
<td>DECLARE Function</td>
<td>Declare a Subroutine/Function Prototype at Program Start</td>
</tr>
<tr>
<td>DEF FNFunction( Arg[, Arg...])</td>
<td>Define a Function (&quot;FNFunction&quot;) that returns a Value. If a Single Line, then &quot;END DEF&quot; is not required</td>
</tr>
<tr>
<td>END DEF</td>
<td>End the Function Definition</td>
</tr>
<tr>
<td>FUNCTION Function( Arg[, Arg...])</td>
<td>Define a Function. Same Operation, Different Syntax as &quot;DEF FNFunction&quot;</td>
</tr>
<tr>
<td>END FUNCTION</td>
<td>End a Function Declaration</td>
</tr>
<tr>
<td>SUB Subroutine( Arg[, Arg...])</td>
<td>Define a &quot;Subroutine&quot; which does not return a Value. If a Single Line, then &quot;END DEF&quot; is not required</td>
</tr>
<tr>
<td>END SUB</td>
<td>End the Subroutine Definition</td>
</tr>
<tr>
<td>DATA Value[, Value...]</td>
<td>Specify File Data</td>
</tr>
</tbody>
</table>
Chapter 15

READ Variable[, Variable...]
Read from the "Data" File
Data

IF Condition THEN Statements ELSE Statements END IF
Perform a Structured
If/Else/Endif
ELSEIF
Perform a Condition
Test/Structured
If/Else/Endif instead of
simply "Else"

ON ERROR GOTO Label
On Error Condition, Jump to
Handler
RESUME [Label]
Executed at the End of an
Error Handler. Can either
return to current location,
0 (Start of Application) or
a specific label
ERR
Return the Current Error
Number
ERL
Return the Line the Error
Occurred at
ERROR #
Execute an Application-
Specific Error (Number "#")
DO WHILE Condition Statements LOOP
Execute "Statements" while
"Condition" is True
DO Statements LOOP WHILE Condition
Execute "Statements" while
"Condition" is True
DO Statements LOOP UNTIL Condition
Execute "Statements" until
"Condition" is True
EXIT
Exit Executing "FOR", "WHILE" and "UNTIL" Loops
without executing Check
SELECT Variable
Execute based on "Value"
"CASE" Statements used to
Test the Value and Execute
Conditionally
CASE Value
Execute within a "SELECT"
Statement if the "Variable"
Equals "Value". "CASE ELSE"
is the Default Case
END SELECT
End the "SELECT" Statement
LINE INPUT
Get Formatted Input from the
User
**PICmicro® MCU Application Software**

- **INPUTS( # )**
  - Get the Specified Number
  - of Characters from
  - the User

- **INKEY$**
  - Check Keyboard and Return
  - Pending Characters or Zero

- **ASC**
  - Convert the Character into
  - an Integer ASCII

- **Code**
  - Convert the Integer ASCII
  - Code into a Character

- **CHR$**
  - Convert the Integer ASCII
  - Code into a Character

- **VAR**
  - Convert the String into an
  - Integer Number

- **STR$**
  - Convert the Integer Number
  - into a String

- **LEFT$( String, # )**
  - Return the Specified Number
  - of Left Most
  - Characters in "String"

- **RIGHT$( String, # )**
  - Return the Specified Number
  - of Right Most
  - Characters in "String"

- **MID$( String, Start, # )**
  - Return/Overwrite the
  - Specified Number ("#") of
  - Characters at Position
  - "Start" in "String"

- **SPACE$( # )**
  - Returns a String of the
  - Specified Number ("#") of
  - ASCII Blanks

- **LTRIM$**
  - Remove the Leading Blanks
  - from a String

- **RTRIM$**
  - Remove the Trailing Blanks
  - from a String

- **INSTR( String, SubString )**
  - Return the Position of
  - "SubString" in "String"

- **UCASE$**
  - Convert all the Lower Case
  - Characters in a String to
  - Upper Case

- **LCASE$**
  - Convert all the Upper Case
  - Characters in a String to
  - Upper Case

- **LEN**
  - Return the Length of a
  - String

- **CLS**
  - Clear the Screen

- **CSRLIN**
  - Return the Current Line that
  - the Cursor is On
POS                     Return the Current Column
that the Cursor is On
LOCATE X, Y             Specify the Row/Column of
the Cursor [Top Left is
1,1]
SPC                     Move the Display the
Specified Number of Spaces
PRINT USING "Format"   Print the Value in the
Specified Format. "+", "#",
. *, ** Characters are
used for number formats
Set the Screen Mode. "Color"
is 0 to display on a
"Color" display, 1 to
display on a "Monochrome".
"Page" is the Page that
receives I/O and "Visual"
is the Page that is
currently active.
COLOR [foreground] [, [background] [,border]] Specify the Currently Active Colors
PALETTE [attribute, color]
Change Color Assignments.
VIEW [[SCREEN] (x1,y1) - (x2,y2)[,[color]]
[,[border]]] Create a small Graphics Window known as a
"Viewport"
WINDOW [[SCREEN] (x1,y1) - (x2,y2)]
Specify the Viewport’s
logical location on the
Display
PSET (x,y)[,color]      Put a Point on the Display
PRESET (x,y)            Return the Point to the
Background Color
LINE (x1,y1)-(x2,y2)[,[Color][,B|BF][,style]]
Draw a Line between the two
specified points. If "B" or
"BF" specified, Draw a Box
("BF" is "Filled")
CIRCLE (x,y), radius[, [color] [,start] [,end]
[,aspect]]] Draw the Circle at center
location and with the
specified “radius”, “start” and “end” are starting and ending angles (in radians). “aspect” is the circle’s aspect for drawing ellipses.

**DRAW CommandString**

Draw an arbitrary Graphics Figure. There should be spaces between the commands.

**Commands:**
- **U#** - Moves Cursor up # Pixels
- **D#** - Moves Cursor down # Pixels
- **E#** - Moves Cursor up and to the right # Pixels
- **F#** - Moves Cursor down and to the right # Pixels
- **G#** - Moves Cursor down and to the Left # Pixels
- **H#** - Moves Cursor up and to the left # Pixels
- **L#** - Moves Cursor Left # Pixels
- **R#** - Moves Cursor Right # Pixels
- **Mxy** - Move the Cursor to the Specified x,y Position
- **B** - Turn Off Pixel Drawing
- **N** - Turns On Cursor and Move to Original Position

**Position**
- **A#** - Rotate Shape in 90 Degree Increments
- **CH** - Set the Drawing Color
- **PHColor#Border** - Set the Shape Fill and Border Colors
- **SH** - Set the Drawing Scale
- **TH** - Rotates # Degrees

**LPRINT**
- Send Output to the Printer

**BEEP**
- “Beep” the Speaker

**SOUND Frequency, Duration**
- Make the Specified Sound on the PC’s Speaker

**PLAY NoteString**
- Output the Specified String of “Notes” to the PC’s Speaker
Chapter 15

DATE$  Return the Current Date
TIME$  Return the Current Time
TIMER  Return the Number of Seconds since Midnight
NAME FileName AS NewFileName  Change the Name of a File
KILL FileName  Delete the File
FILES [FileName.Ext]  List the File (MS-DOS "dir"). "FileName.Ext" can contain "Wild Cards"
OPEN FileName [FOR Access] AS #Handle  Open the File as the Specified Handle (Starting with the "#" Character).
   Access:
      I - Open for Text Input
      O - Open for Text Output
      A - Open to Append Text
      B - File is Opened to Access Single Bytes
      R - Open to Read and Write Structured Variables
CLOSE #Handle  Close the Specified File
RESET  Close all Open Files
EOF  Returns "True" if at the End of a File
READ #Handle, Variable  Read Data from the File
GET #Handle, Variable  Read a Variable from the File
INPUT #Handle, Variable  Read Formatted Data from the File using "INPUT", "INPUT USING" and "INPUT$" Formats
WRITE #Handle, Variable  Write Data to the File
PUT #Handle, Variable  Write a Variable to a File
PRINT #Handle, Output  Write Data to the File using the "PRINT" and "PRINT USING" Formats
SEEK #Handle, Offset  Move the File Pointer to the Specified Offset within the File
PicBasic

microEngineering Labs, Inc.'s ("meLab's"), "PicBasic" is an excellent tool for learning about the PICmicro® MCU, before taking the big plunge into assembly language programming. The source code required by the compiler is similar to the Parallax Basic Stamp BS2's "PBASIC" with many improvements and changes to make the language easier to work with and support different PICmicro® MCUs.

PicBasic does not currently have the ability to link together multiple source files which means that multiple source files must be "included" in the overall source. Assembly language statements are inserted in line to the application. PicBasic produces either assembler source files or completed ".hex" files. It does not create object files for linking modules together.

For additional information and the latest device libraries, look at the microEngineering Labs, Inc., Web page at:

http://www.melabs.com/mel/home.htm

PicBasic Pro is an MS-DOS command line application that is invoked using the statement:

PBP[W] [options...] source

"Options" are compiler execution options and are listed in the table below:

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h/-?</td>
<td>Display the help screen. The help screen is also displayed if no options or source file name is specified</td>
</tr>
</tbody>
</table>
550 Chapter 15

-ampasm Use the MPASM Assembler and not the PicBasic Assembler
-c Insert Comments into PicBasic Compiler produced Assembler Source File. Using this option is recommended if you are going to produce MPASM Assembler Source from PicBasic
-iPath Specify a new directory "Path" to use for include files in PicBasic
-lLibrary Specify a different library to use when compiling. Device specific libraries are provided by PicBasic when the processor is specified
-od Generate a listing, symbol table and map files
-ol Generate a listing file
-pPICmicro Specify the "PICmicro" MCU that the source is to be compiled into. If this parameter is not specified, then a PIC16F84 is used as the processor. "PICmicro" MCU is in the format: 16F84, where the "PIC" at the start of the Microchip part number is not specified.
-s Do not assemble the compiled code
-v Turn on "Verbose Mode" which provides additional information when the application is compiled

PicBasic does assume a constant set of configuration values. For most PICmicro® MCUs the configuration fuses are set as listed in the table below:

<table>
<thead>
<tr>
<th>Feature</th>
<th>PicBasic Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Protect</td>
<td>Off</td>
</tr>
<tr>
<td>Oscillator</td>
<td>XT - or Internal RC if 12Cxxx</td>
</tr>
<tr>
<td>WDT</td>
<td>On</td>
</tr>
<tr>
<td>PWRT</td>
<td>Off</td>
</tr>
</tbody>
</table>

Each byte takes place in one of the words; for example, “b4” is the least significant byte of “w2”. The 16-bit variables are defined as being a part of the 16-bits taken
up by “w0” (“b0” and “b1”). This method works well, but care has to be taken to make sure that the overlapping variables are kept track of and not used incorrectly. The most common problem for new Basic Stamp developers is defining a variable on b0 and w0 and having problems when a write to one variable overwrites the other.

To provide these variables to the PicBasic application, the Basic Stamp variable declaration files are defined in the following two “include” files that are shown within “include” statements below. Only one of these statements can be used in an application.

```
include "bs1defs.bas"
include "bs2defs.bas"
```

A much better way of declaring variables is to use the “var” directive to select the different variables at the start of the application and let the PicBasic compiler determine where the variables belong and how they are accessed (i.e., put in different variable pages). Along with the “var” directive, the “word”, “byte”, and “bit” directives are used to specify the size of the variable. Some example variable declarations are

```
WordVariable var word    ' Declare a 16 Bit Variable
ByteVariable var byte    ' Declare an 8 Bit Variable
BitVariable var bit      ' Declare a single byte Variable
```

Initial values for the variables cannot be made at the variable declarations.

Along with defining variables, the “var” directive can be used to define variable labels built out of previously defined variables to specify specific data. Using the variables above, I can break “WordVariable” up into a top half and bottom half and “ByteVariable” into specific bytes with the statements:
Variables can also be defined over registers. When the PicBasic libraries are merged with the source code, the standard PICmicro® MCU register names are available within the application. Using this capability, labels within the application can be made to help make the application easier to work with. For example, to define the bits needed for an LCD, the declarations below could be used:

```plaintext
LCDData var PORTB  ' PORTB as the 8 Bits of Data
LCDS var PORTA.0  ' RA0 is "E" Strobe
LCDRS var PORTA.1 ' RA1 is Data/Instruction Select
LCDRW var PORTA.2 ' RA2 is the Read/Write Select Bit
```

When variables are defined using the “var” and “system” directives, specific addresses can be made in the application. For example, the statement:

```plaintext
int_w var byte $0C system
```

will define the variable “_w” at address 0x00C in the system. This reserves address 0x00C and does not allow its use by any other variables. The bank of a variable can be specified using the “system” directive like:

```plaintext
int_status var byte bank0 system
```

These two options to the “var” directive are useful when defining variables for interrupt handlers as discussed below.
Along with redefining variables with the “var” statement, PicBasic also has the “symbol” directive. The symbol directive provides the same capabilities as the “var” statement and it is provided simply for compatibility with the BS1. If you were only developing PicBasic applications, I would recommend only using “var” and avoiding the “symbol” directive. Single dimensional arrays can be defined within PicBasic for each of the three data types when the variable is declared.

```plaintext
WordArray var word[10] 'Ten Word Array
BitArray var bit[12]     'Twelve Bit Array
```

Note that bits can be handled as an array element. Depending on the PICmicro® MCU part number, the maximum array sizes are

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Maximum Number of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>32</td>
</tr>
<tr>
<td>Byte</td>
<td>64</td>
</tr>
<tr>
<td>Bit</td>
<td>128</td>
</tr>
</tbody>
</table>

As part of the “bit” definition, I/O port pins are pre-defined within PicBasic. Up to 16 pins (addressed using the “Pin#” format, where “#” is the pin number) can be accessed, although how they are accessed changes according to the PICmicro® MCU part number the application is designed for. The pins for different parts are defined as:

<table>
<thead>
<tr>
<th>Number of Pins</th>
<th>Pins 0 - 7</th>
<th>Pins 8 - 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>GPIO</td>
<td>Mapped onto Pins</td>
</tr>
<tr>
<td>18</td>
<td>PORTB</td>
<td>PORTA</td>
</tr>
</tbody>
</table>
Chapter 15

Note that not all the ports that have 8 pins are specified. For example, accessing “RA6” in an 18-pin device (which does not have an “RA6” bit) will not do anything.

Constants are declared in a similar manner to variables, but by using the “con” directive with a constant parameter:

SampleConstant con 3 + 7 ' Define a Sample Constant

Constant values can be in four different formats. The table below lists the different formats and the modifiers to indicate to the PicBasic compiler which data type is being specified:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Modifier</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>None</td>
<td>PicBasic Default is Decimal</td>
</tr>
<tr>
<td>Hex</td>
<td>$</td>
<td>&quot;$&quot; is placed before the Number</td>
</tr>
<tr>
<td>Binary</td>
<td>%</td>
<td>&quot;%&quot; is placed before the Number</td>
</tr>
<tr>
<td>ASCII</td>
<td>&quot;</td>
<td>Double Quotes placed around a Single Character</td>
</tr>
</tbody>
</table>

In the table above, note that only an ASCII byte can be passed within double quotes. Some instructions (described below) can be defined with strings of characters that are enclosed within double quotes.

The “define” statement is used to change constants given defaults within the PICmicro® MCU when a PicBasic compiled application is running. The format is

DEFINE Label NewValue

The labels, their default values, and their values are listed in the table below:
<table>
<thead>
<tr>
<th>Define</th>
<th>Default</th>
<th>Optional Values</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUTTON_PAUSE</td>
<td>10</td>
<td>Any Positive Int</td>
<td>Button Debounce Delay in msecs</td>
</tr>
<tr>
<td>CHAR_PACING</td>
<td>1000</td>
<td>Any Positive Int</td>
<td>Time between SerOut Characters</td>
</tr>
<tr>
<td>DEBUG_BAUD</td>
<td>2400</td>
<td>Any</td>
<td>Specified Data Rate of Debug information</td>
</tr>
<tr>
<td>DEBUG_BIT</td>
<td>0</td>
<td>0 - 7</td>
<td>Output Pin for Debug Serial Output</td>
</tr>
<tr>
<td>DEBUG_MODE</td>
<td>1</td>
<td>0, 1</td>
<td>Polarity of Debug NRZ Output data. &quot;0&quot; Positive, &quot;1&quot; Inverted</td>
</tr>
<tr>
<td>DEBUG_PACING</td>
<td>1000</td>
<td>Any Positive Int</td>
<td>Time between Output Characters for DEBUG Statements</td>
</tr>
<tr>
<td>DEBUG_REG</td>
<td>PORTB</td>
<td>Any PORT</td>
<td>Port Debug Output Pin is Connected to</td>
</tr>
<tr>
<td>DEBUGIN_BIT</td>
<td>0</td>
<td>0 - 7</td>
<td>Input Pin for Debug Serial Output</td>
</tr>
<tr>
<td>DEBUGIN_MODE</td>
<td>1</td>
<td>0, 1</td>
<td>Polarity of Debug NRZ Input data. &quot;0&quot; Positive, &quot;1&quot; Inverted</td>
</tr>
<tr>
<td>DEBUGIN_REG</td>
<td>PORTB</td>
<td>Any PORT</td>
<td>Port Debug Input Pin is Connected to</td>
</tr>
<tr>
<td>HSER_BAUD</td>
<td>2400</td>
<td>Any</td>
<td>Hardware Serial Port’s Data Rate</td>
</tr>
<tr>
<td>HSER_SPBRG</td>
<td>25</td>
<td>0 - 0x0FF</td>
<td>Hardware Serial Port’s SPBRG Register Value</td>
</tr>
<tr>
<td>HSER_RCSTA</td>
<td>0x090</td>
<td>0 - 0xFF</td>
<td>Hardware Serial Port’s Initialization value for &quot;RCSTA&quot; register. Default set for Asynchronous Communications</td>
</tr>
<tr>
<td>HSER_TXSTA</td>
<td>0x020</td>
<td>0 - 0xFF</td>
<td>Hardware Serial Port’s Initialization value for &quot;TXSTA&quot; register. Default set for Asynchronous Communications</td>
</tr>
<tr>
<td>HSER_EVEN</td>
<td>1</td>
<td>0, 1</td>
<td>Hardware Serial Port’s Parity Select Values. Only used if Parity checking is desired</td>
</tr>
<tr>
<td>HSER_ODD</td>
<td>1</td>
<td>0, 1</td>
<td>Hardware Serial Port’s Parity Select Values. Only used if Parity checking is desired</td>
</tr>
<tr>
<td>Define</td>
<td>Default</td>
<td>Optional Values</td>
<td>Comments</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>-----------------</td>
<td>----------</td>
</tr>
<tr>
<td>I2C_HOLD</td>
<td>1</td>
<td>0, 1</td>
<td>Stop I2C transmission while the SCL line is held low</td>
</tr>
<tr>
<td>I2C_INTERNAL</td>
<td>1</td>
<td>0, 1</td>
<td>Set to use Internal EEPROM in the 12Cxx PICmicro® MCUs</td>
</tr>
<tr>
<td>I2C_SCLOUT</td>
<td>1</td>
<td>0, 1</td>
<td>Use a Bipolar Driver instead of an Open-Drain I2C Clock Driver</td>
</tr>
<tr>
<td>I2C_SLOW</td>
<td>1</td>
<td>0, 1</td>
<td>Run the I2C at no more than 100 kbps data rate</td>
</tr>
<tr>
<td>LCD_BITS</td>
<td>4</td>
<td>4, 8</td>
<td>Number of Bits for LCD Interface</td>
</tr>
<tr>
<td>LCD_DBIT</td>
<td>0</td>
<td>0, 4</td>
<td>Specify the Data bit for LCD Data</td>
</tr>
<tr>
<td>LCD_DREG</td>
<td>PORTA</td>
<td>Any PORT</td>
<td>Select the LCD Data Port</td>
</tr>
<tr>
<td>LCD_EBIT</td>
<td>3</td>
<td>0 - 7</td>
<td>Specify the Bit for the LCD Clock</td>
</tr>
<tr>
<td>LCD_EREG</td>
<td>PORTB</td>
<td>Any PORT</td>
<td>Specify the Port for the LCD ”B” Clock Bit</td>
</tr>
<tr>
<td>LCD_LINES</td>
<td>2</td>
<td>1, 2</td>
<td>Specify the Number of Lines on the LCD. Check Information on LCDs for how the multiple line specification is used in some single line LCDs.</td>
</tr>
<tr>
<td>LCD_RSBIT</td>
<td>4</td>
<td>Any PORT</td>
<td>LCD RS Bit Selection</td>
</tr>
<tr>
<td>LCD_RSREG</td>
<td>PORTA</td>
<td>Any PORT</td>
<td>LCD RS Bit Select Register</td>
</tr>
<tr>
<td>OSC</td>
<td>4</td>
<td>3, 4, 8, 10, 12, 16, 20</td>
<td>Specify PICmicro® MCU Operating Speed in MHz. Note &quot;3&quot; is actually 3.58</td>
</tr>
<tr>
<td>OSCCAL_1K</td>
<td>1</td>
<td>0, 1</td>
<td>Set OSCCAL for PIC12C672</td>
</tr>
<tr>
<td>OSCCAL_2K</td>
<td>1</td>
<td>0, 1</td>
<td>Set OSCCAL for PIC12C672</td>
</tr>
<tr>
<td>SER2_BITS</td>
<td>8</td>
<td>4 - 8</td>
<td>Specify Number of bits sent with &quot;SERIN2&quot; and &quot;SEROUT2&quot; instructions</td>
</tr>
</tbody>
</table>
The “OSC” define should be specified if serial I/O is going to be implemented in the PICmicro® MCU. This value is used by the compiler to calculate the time delays necessary for each bit.

Assembly language can be inserted at any point within a PicBasic application. Single instructions can be inserted by simply starting the line with a “@” character:

```
@bcf INTCON, T0IF       ; Reset T0IF Flag
```

Multiple lines of assembly language are prefaced by the “asm” statement and finished with the “endasm”. An example of this is shown below:

```
asm
    movlw 8             ; Loop 8x
    Loop
    bcf PORTA, 0        ; Pulse the Bit
    bcf PORTA, 0
    addlw $0FF          ; Subtract 1 from “w”
    btfss STATUS, Z     ; Do 8x
    goto Loop
endasm
```

Note that labels inside the assembler statements do not have a colon at the end of the string and that the traditional assembly language comment indicator (the semicolon [;]) is used.

Implementing interrupt handlers in PicBasic can be done in one of two ways. The simplest way of implementing it is using the “ON INTERRUPT GOTO Label” statement. Using this statement, any time an interrupt request is received, the “Label” specified in the ON INTERRUPT statement will be executed until there is a “resume” instruction, which returns from an interrupt. Using this type of interrupt handler, straight PicBasic statements can be used and assembly language statements avoided.
Chapter 15

The basic operation looks like:

```
ON INTERRUPT GOTO IntHandler
```

```
IntHandler:
    disable  ' Turn off interrupt and debug requests
    ;       ' Process Interrupt
    enable   ' Enable other interrupts and debug requests
    resume   ' Return to the executing code
```

The problem with this method is the interrupt handler is executed once the current instruction has completed. If a very long statement is being executed (say a string serial send), then the interrupt will not be serviced in a timely manner.

The best way of handling an interrupt is to add the interrupt handler as an assembly language routine. To reference the interrupt handler, the "define INTHAND Label" instruction is used to identify the label where the assembly language code is listed. The interrupt handler will be moved to start at address 0x004 in the mid-range devices.

A code template for generic mid-range PICmicro® MCU interrupt handlers is

```
int_w var byte 0x020 system  ' Define the
Variables
int_status var byte bank0 system
int_fsr var byte bank0 system
int pclath byte bank0 system
```
define INTHAND IntHandler

' Specify what the
' Interrupt
' Handler is

: ; Interrupt Handler - to be relocated to 0x00004
asm
IntHandler
movf int_w, w ; Save the Context
Registors
movf STATUS, w
bcf STATUS, RP0 ; Move to bank 0
bcf STATUS, RP1
movf int_status
movf FSR, w
movf int_fsr
movf PCLATH, w
movf int_pclath
clrf PCLATH
; #### - Execute Interrupt Handler Code Here
movf int_pclath, w ; Finished,
restore the
Context
movf PCLATH ; Registers
movf int_fsr, w
movf FSR
movf int_status, w
movf STATUS
swapf int_w, f
swapf int_w, w
retfie
endasm

In the interrupt template, note that the “worst case”
condition context register save is presented.
Mathematical operators used in assignment state-
ments and PicBasic instructions are very straightfor-
ward in PicBasic and work conventionally. In Basic
Stamp PBASIC, you must remember that the operations
execute from left to right. This means that the
statement:

\[ A = B + C \times D \]
which would be expected to operate as:

1. Multiply “C” and “D”
2. Add the results from one to “B”

in Parallax PBASIC, returns the result of:

1. Get the Sum of “B” and “C”
2. Multiply the results from one with “D”

PicBasic does not follow the PBASIC evaluation convention and returns the “expected” result from complex statements like the one above. This means that in PicBasic, you do not have to break complex statements up into single operations, like you do in PBASIC, to avoid unexpected expression evaluation. If you are using a Basic Stamp to “prototype” PicBasic applications, then break up the complex statements and use the temporary values.

The mathematical operators used are listed in the table below along with their execution priority and parameters. All mathematical operators work with 16-bit values.
<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest</td>
<td>Parm1 + Parm2</td>
<td>Return the Sum of &quot;Parm1&quot; and &quot;Parm2&quot;</td>
</tr>
<tr>
<td></td>
<td>Parm1 - Parm2</td>
<td>Return the Result of &quot;Parm2&quot; Subtracted from &quot;Parm1&quot;</td>
</tr>
<tr>
<td></td>
<td>Parm1 * Parm2</td>
<td>Return the least significant sixteen bits of the product of &quot;Parm1&quot; and &quot;Parm2&quot;. This is often referred to as Bytes 0 and 1 of the result</td>
</tr>
<tr>
<td></td>
<td>Parm1 */ Parm2</td>
<td>Return the middle sixteen bits of the product of &quot;Parm1&quot; and &quot;Parm2&quot;. This is often referred to as Bytes 1 and 2 of the result</td>
</tr>
<tr>
<td></td>
<td>Parm1 ** Parm2</td>
<td>Return the most significant sixteen bits of the product of &quot;Parm1&quot; and &quot;Parm2&quot;. This is often referred to as Bytes 2 and 3 of the result</td>
</tr>
<tr>
<td></td>
<td>Parm1 / Parm2</td>
<td>Return the number of times Parm2 can be divided into Parm1 evenly</td>
</tr>
<tr>
<td></td>
<td>Parm1 // Parm2</td>
<td>Return the remainder from dividing Parm2 into Parm1. This is known as the &quot;Modulus&quot;.</td>
</tr>
<tr>
<td></td>
<td>Parm1 &amp; Parm2</td>
<td>Return the bitwise value of &quot;Parm1&quot; AND &quot;Parm2&quot;</td>
</tr>
<tr>
<td></td>
<td>Parm1</td>
<td>Parm2</td>
</tr>
<tr>
<td></td>
<td>Parm1 ^ Parm2</td>
<td>Return the bitwise value of &quot;Parm1&quot; XOR &quot;Parm2&quot;</td>
</tr>
<tr>
<td></td>
<td>Parm1 &lt;&lt; Parm2</td>
<td>Shift &quot;Parm1&quot; to the left &quot;Parm2&quot; bits. The new least significant bits will all be zero</td>
</tr>
<tr>
<td></td>
<td>Parm1 &gt;&gt; Parm2</td>
<td>Shift &quot;Parm1&quot; to the right &quot;Parm2&quot; bits. The new most significant bits will all be zero</td>
</tr>
<tr>
<td></td>
<td>ABS Parm1</td>
<td>Return the magnitude of a number. (&quot;ABS -4&quot; is equal to &quot;ABS 4&quot; and returns &quot;4&quot;)</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm1 MAX Parm2</td>
<td>Return the higher Parameter</td>
<td></td>
</tr>
<tr>
<td>Parm1 MIN Parm2</td>
<td>Return the lower Parameter</td>
<td></td>
</tr>
<tr>
<td>Parm1 DIG Parm2</td>
<td>Return Digit Number &quot;Parm2&quot; (Zero Based) of &quot;Parm1&quot;. (&quot;123 DIG 1&quot; returns &quot;2&quot;)</td>
<td></td>
</tr>
<tr>
<td>DCD Parm1</td>
<td>Return a value with only the &quot;Parm1&quot; bit Set. (&quot;DCD 4&quot; returns &quot;00000000&quot;)</td>
<td></td>
</tr>
<tr>
<td>NCD Parm1</td>
<td>Return the Bit number of the highest set bit in &quot;Parm1&quot;</td>
<td></td>
</tr>
<tr>
<td>Parm1 REV Parm2</td>
<td>Reverse the Bits in &quot;Parm1&quot; from zero to &quot;Parm2&quot;. (&quot;$10101100 REV 4&quot; will return &quot;$10100011&quot;)</td>
<td></td>
</tr>
<tr>
<td>SQR Parm1</td>
<td>Return the Integer Square Root of &quot;Parm1&quot;</td>
<td></td>
</tr>
<tr>
<td>SIN Parm1</td>
<td>Return the Trigonometric &quot;Sine&quot; of &quot;Parm1&quot;. The returned value will be based on a circle of radius 127 and 256 degrees (not the traditional 360)</td>
<td></td>
</tr>
<tr>
<td>COS Parm1</td>
<td>Return the Trigonometric &quot;Cosine&quot; of &quot;Parm1&quot;. The returned value will be based on a circle of radius 127 and 256 degrees (not the traditional 360)</td>
<td></td>
</tr>
</tbody>
</table>
Along with the mathematical operators, the “if” statement provides the following “Test Conditions”. This is listed in the following table. Note that both the “BASIC” standard labels as well as the “C” standard labels are used. “Parm1” and “Parm2” are constants, variables, or statements made up of expression statements along with the different mathematical operators and test conditions.

When a test condition is true, a nonzero is returned, if it is false, then a zero is returned. Using this convention, single variable parameters can be tested in “if” statements rather than performing comparisons of them to zero.

<table>
<thead>
<tr>
<th>Test Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm1 = Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; equals &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 == Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; equals &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 &lt;&gt; Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; does not equal &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 != Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; does not equal &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 &lt; Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; is less than &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 &lt;= Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; is less than or equal to &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 &gt; Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; is greater than &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 &gt;= Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; is greater than or equal to &quot;Parm2&quot;</td>
</tr>
<tr>
<td>Parm1 AND Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; is Non-Zero and &quot;Parm2&quot; is Non-Zero</td>
</tr>
<tr>
<td>Parm1 &amp;&amp; Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; is Non-Zero and &quot;Parm2&quot; is Non-Zero</td>
</tr>
<tr>
<td>Parm1 OR Parm2</td>
<td>Return a Non-Zero if &quot;Parm1&quot; is Non-Zero or &quot;Parm2&quot; is Non-Zero</td>
</tr>
</tbody>
</table>
The PicBasic instructions are based on the Parallax Basic Stamp “PBASIC” language and while there are a lot of similarities, they are really two different languages. In the following table, all the PicBasic instructions are listed with indications of any special considerations that should be made for them with respect to being compiled in a PICmicro® MCU.

These “instructions” are really “library routines” that are called by the mainline of the application. I am mentioning this because you will notice that the size of the application changes based on the number of instructions that are used in the application. Program memory size can be drastically reduced by looking at the different instructions that are used and changing the statements to assembler or explicit PicBasic statements.

When the various instructions are specified, note that the square brackets (“[“ and “]”) are used to specify data tables in some instructions. For this reason, optional values use braces (“{“ and “}”), which breaks with the conventions used in the rest of the book.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH Index, [Label (,Label. . .)]</td>
<td>Jump to the “Label” specified by the value in “Index”. “Index” is zero based, so an Index of zero will cause execution jump to the first “Label”, an “Index” of one will cause execution to jump to the second “Label” and so on. This instruction only jumps within the current page; if a PICmicro® MCU with more than one page of program memory is used, then the “BRANCHL” instruction is recommended.</td>
</tr>
<tr>
<td>BRANCHL Index, [Label {,Label. . .}]</td>
<td>Jump to the “Label” specified by the value in “Index”. “Index” is zero based, so an Index of zero will cause execution jump to the first “Label”, an “Index” of one will cause execution to jump to the second “Label” and so on. This instruction can jump anywhere in PICmicro® program memory.</td>
</tr>
<tr>
<td>BUTTON Pin, Down, Delay, Rate, Bvar, Variable, Action</td>
<td>Jump to “Label” when the Button has been pressed for the specified number of milliseconds. “Rate” is how many invocations after the first “BUTTON” jump is true that an “autorepeat” happens. “Bvar” is a byte sized variable only used in this function. “Action” is whether or not you want the jump to take place when the key is pressed (“1”) or released (“0”).</td>
</tr>
<tr>
<td>CALL Label</td>
<td>Execute the assembly language “call” instructions</td>
</tr>
<tr>
<td>CLEAR</td>
<td>Load all the Variables with Zero</td>
</tr>
<tr>
<td>COUNT Pin, Period, Variable</td>
<td>Count the number of pulses on “Pin” that occur in “Period” msecs.</td>
</tr>
<tr>
<td>DATA @Location, Constant {,Constant. . .}</td>
<td>Store Constants in Data EEPROM starting at “Location” when the PICmicro® MCU is programmed. For data at different addresses, use multiple “DATA” statements.</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DEBUG Value</td>
<td>Define the &quot;DEBUG&quot; pin as output with the serial output parameters used in the &quot;DEBUG&quot; defines at reset. When this instruction is executed, pass the parameter data. If an ASCII &quot;#&quot; (0x023) is sent before a &quot;Value&quot;, the decimal numeric is sent, rather than the ASCII byte. This instruction (and &quot;DEBUGIN&quot;) can be used for serial I/O as they take up less space than the &quot;SERIN&quot; and &quot;SEROUT&quot; instructions</td>
</tr>
<tr>
<td>DEBUGIN {TimeOut, Label,] [Variable {,Variable...]}]</td>
<td>Define the &quot;DEBUGIN&quot; pin as an input with the serial input parameters used in the &quot;DEBUGIN&quot; defines at reset. When this instruction is executed, wait for a data byte to come in or jump to the label if the &quot;TimeOut&quot; value (which is specified in msecs) is reached</td>
</tr>
<tr>
<td>DISABLE</td>
<td>DISABLE Interrupts and Debug Operations. Interrupts will still be acknowledged by &quot;ON INTERRUPT GOTO&quot; Will not execute</td>
</tr>
<tr>
<td>DISABLE INTERRUPT</td>
<td>DISABLE Interrupts and Debug Operations. Interrupts will still be acknowledged by &quot;ON INTERRUPT GOTO&quot; Will not execute</td>
</tr>
<tr>
<td>DTMFOUT Pin, {On,Off,} [Tone{,Tone...}]</td>
<td>Output the Touch tone sequence on the specified pin. Tones 0 through 9 are the same as on the telephone keypad. Tone 10 is the &quot;*&quot; key and tone 11 is the &quot;#&quot; key. Tones 12 through 15 correspond to the extended key standards for &quot;A&quot; to &quot;D&quot;. Filtering is required on the pin output to &quot;smooth&quot; out the signal output</td>
</tr>
<tr>
<td>EEPROM Location, [Constant [[,Constant...]]]</td>
<td>Store new values in EEPROM when the PICmicro® MCU is programmed. This instruction is the same as &quot;DATA&quot;</td>
</tr>
<tr>
<td>ENABLE</td>
<td>Enable debug and interrupt processing that was stopped by &quot;DISABLE&quot;</td>
</tr>
</tbody>
</table>
ENABLE DEBUG
Enable debug operations that were stopped by "DISABLE"

ENABLE INTERRUPT
Enable Interrupt operations that were stopped by the
"DISABLE" and "DISABLE INTERRUPT" instructions

END
Stop processing the application and put the PICmicro® MCU in
a low power "Sleep" mode

FOR Variable = Start TO Stop {STEP Value}
Execute a Loop, first initializing "Variable" to the "Start"
value until it reaches the "Stop" Value. The increment value
defaults to one if no "STEP" value is specified. When "NEXT"
is encountered "Variable" is incremented and tested against
the "Stop" Value

NEXT {Variable}

FREQOUT Pin, On, Frequency
Output the specified "Frequency" on the "Pin" for "On" msecs.
{,Frequency}
If a second "Frequency" is specified, output this at the same
time. Filtering is required on the pin output to "smooth"
out the signal output

GOSUB Label
Call the subroutine that starts at address "Label". The
existence of "Label" is checked at compile time

GOTO Label
Jump to the code that starts at address "Label".

HIGH Pin
Make "Pin" an Output and drive it High

HSERIN {ParityLabel,} [TimeOut,Label,] [Variable {[[,Variable. . .]}]
Receive one or more bytes from the built in USART (if
present). The "ParityLabel" will be jumped to if the parity
of the incoming data is incorrect. To use "ParityLabel",
multiply sure the "HSER_EVEN" or "HSER_ODD" defines have been
specified

HSEROUT [Value {[,.Value. . .]}
Transmit one or more bytes from the built in USART (if
present)

I2CREAD DataPin, ClockPin, ControlByte, [Address,]
Read a Byte string from an I2C device. The "ControlByte" is
used to access the device with block or device select bits.
This instruction can be used to access internal EEPROM in the
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Variable</td>
<td>PIC12CEXXX devices by entering the “define I2C_INTERNAL 1”</td>
</tr>
<tr>
<td></td>
<td>statement at the start of the application code.</td>
</tr>
<tr>
<td>I2CWRITE DataPin,</td>
<td>Send a byte string to an I2C device. The “ControlByte” is used</td>
</tr>
<tr>
<td>ClockPin, Control,</td>
<td>to access the device with block or device select bits. This</td>
</tr>
<tr>
<td>[Address,]</td>
<td>instruction can be used to access internal EEPROM in the</td>
</tr>
<tr>
<td>[Value[,Value...]]</td>
<td>PIC12CEXXX devices by entering the “define I2C_Internal 1”</td>
</tr>
<tr>
<td></td>
<td>statement at the start of the application code.</td>
</tr>
<tr>
<td>IF Comp THEN Label</td>
<td>Evaluate the “Comp” Comparison Expression and if it is not</td>
</tr>
<tr>
<td>Label</td>
<td>equal to zero then jump to “Label”.</td>
</tr>
<tr>
<td>IF Comp THEN</td>
<td>Evaluate the “Comp” Comparison Expression and if it is not</td>
</tr>
<tr>
<td>Statement</td>
<td>equal to zero then execute the “Statements” below until</td>
</tr>
<tr>
<td></td>
<td>either an “ELSE” or an “ENDIF” statement is encountered. If “Comp”</td>
</tr>
<tr>
<td>ELSE</td>
<td>evaluates to zero, then skip over the “Statements” after the</td>
</tr>
<tr>
<td>Statement</td>
<td>“IF” statement are ignored to the “ELSE” or “ENDIF” Statements, after</td>
</tr>
<tr>
<td>ENDIF</td>
<td>which any Statements are executed</td>
</tr>
<tr>
<td>INCLUDE “file”</td>
<td>Load in “file.bas” in the current directory and insert it at</td>
</tr>
<tr>
<td></td>
<td>the current location in the source file</td>
</tr>
<tr>
<td>INPUT Pin</td>
<td>Put the specified pin into “Input Mode”</td>
</tr>
<tr>
<td>[LET] Assignment</td>
<td>Optional instruction value for an Assignment statement</td>
</tr>
<tr>
<td>LCDOUT Value[,Value…]</td>
<td>Send the specified Bytes to the LCD connected to the</td>
</tr>
<tr>
<td></td>
<td>PICmicro MCU. The LCD’s operating parameters are set with the</td>
</tr>
<tr>
<td></td>
<td>“LCD” defines. To send an instruction byte to the LCD, a</td>
</tr>
<tr>
<td></td>
<td>$0FE byte is sent first</td>
</tr>
<tr>
<td>LOCKEDOWN offset,</td>
<td>Go through a list of Constants with an “offset” and store</td>
</tr>
<tr>
<td>[Constant</td>
<td>the constant value at the offset in the second “Variable”.</td>
</tr>
</tbody>
</table>
If the "offset" is greater than the number of constants, then zero is returned in "Variable". "Offset" is zero based, the first constant is returned if "offset" is equal to zero.

Search the list and find the constant value that meets the condition "Test". If "Test" is omitted, then the "LOOKDOWN2" instruction behaves like the "LOOKDOWN" instruction with the "Test" is assumed to be and equals sign ("=").

Compare the first "Variable" value with a constant string and return the offset into the constant string in the second "Variable". If there is no match, then the second "Variable" is not changed.

Compare the first "Variable" value with a "Value" string and return the offset into the "Value" string in the second "Variable". If there is no match, then the second "Variable" is not changed. "LOOKUP2" differs from "LOOKUP" as the "Values" can be sixteen bit variable values.

Make "Pin" an output pin and drive it with a "High" voltage.

Put the PICmicro® MCU to "sleep" for the period value which is given in the table below:

<table>
<thead>
<tr>
<th>Period</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18 msecs</td>
</tr>
<tr>
<td>1</td>
<td>36 msecs</td>
</tr>
<tr>
<td>2</td>
<td>73 msecs</td>
</tr>
<tr>
<td>3</td>
<td>144 msecs</td>
</tr>
<tr>
<td>4</td>
<td>288 msecs</td>
</tr>
<tr>
<td>5</td>
<td>576 msecs</td>
</tr>
<tr>
<td>6</td>
<td>1,152 msecs</td>
</tr>
<tr>
<td>7</td>
<td>2,304 msecs</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ON DEBUG GOTO Label</td>
<td>When invoked, every time an instruction is about to be invoked, the Debug monitor program at &quot;Label&quot; is executed. Two Variables, the &quot;DEBUG_ADDRESS&quot; word and &quot;DEBUG_STACK&quot; byte must be defined as &quot;bank0 system&quot; bytes. To return from the debug monitor, a &quot;RESUME&quot; instruction is used.</td>
</tr>
<tr>
<td>ON INTERRUPT GOTO Label</td>
<td>Jump to the Interrupt Handler starting at &quot;Label&quot;. When the interrupt handler is complete, execute a &quot;RESUME&quot; instruction.</td>
</tr>
<tr>
<td>OUTPUT Pin</td>
<td>Put &quot;Pin&quot; into Output Mode</td>
</tr>
<tr>
<td>PAUSE Period</td>
<td>Stop the PICmicro® MCU from executing the next instruction for &quot;Period&quot; milliseconds. &quot;PAUSE&quot; does not put the PICmicro® MCU to &quot;sleep&quot; like &quot;NAP&quot; does.</td>
</tr>
<tr>
<td>PUSHEUS Period</td>
<td>Stop the PICmicro® MCU from executing the next instruction for &quot;Period&quot; microseconds.</td>
</tr>
<tr>
<td>PEEK Address, Variable</td>
<td>Return the Value at the register &quot;Address&quot; in &quot;Variable&quot;</td>
</tr>
<tr>
<td>POKE Address, Value</td>
<td>Write the register &quot;Address&quot; with the &quot;Value&quot;</td>
</tr>
<tr>
<td>POT Pin, Scale, Variable</td>
<td>Read a Potentiometer's wiper when one of its pins is connected to a capacitor. &quot;Scale&quot; is a value which will change the returned value until it is in the range of 0 to 0x0FF (255)</td>
</tr>
<tr>
<td>PULSIN Pin, State, Variable</td>
<td>Measure an incoming pulse width of &quot;Pin&quot;. &quot;State&quot; indicates the state of the expected Pulse. If a 4 MHz clock is used with the PICmicro® MCU, the time intervals have a granularity of 10 usecs</td>
</tr>
<tr>
<td>PULSOUT Pin, Period</td>
<td>Pulse the &quot;Pin&quot; for the &quot;Period&quot;. If the PICmicro® MCU is run with a 4 MHz clock, then the pulse &quot;Period&quot; will have a granularity of 10 usecs</td>
</tr>
<tr>
<td>PWM Pin, Duty, Cycle</td>
<td>Output a Pulse Width modulated signal on &quot;Pin&quot;. Each cycle is 5 usecs long for a PICmicro® MCU running at 4 MHz. &quot;Duty&quot;</td>
</tr>
</tbody>
</table>
selects the fraction of the cycles (zero to 255) that the PWM is active. "Cycle" specifies the number of cycles that is output.

**RANDOM Variable**

Load "Variable" with a pseudo-random Variable

**RCTIME Pin, State, Variable**

Measure the absolute time required for a signal to be delayed in a RC Network. If a 4 MHz oscillator is used with the PICmicro® MCU, then the value returned will be in 10 usec increments.

**READ Address, Variable**

Read the Byte in the built in Data EEPROM at "Address" and return its value into "Variable". This instruction does not work with the built in EEPROM of PIC12CExx parts.

**RESUME {Label}**

Restore execution at the instruction after the "ON DEBUG" or "ON INTERRUPT" instruction handler was executed. If a "Label" is specified then the hardware is returned to its original state and execute jumps to the code after "Label".

**RETURN**

Return to the instruction after the "GOSUB" instruction.

**REVERSE Pin**

Reverse the function of the specified "Pin". For example, if it were in "output mode", it is changed to "input mode".

**SERIN Pin, Mode, {Timeout, Label,} {Qualifier...} [Variable...]]**

Receive one or more asynchronous data bytes on "Pin". The "Pin" can be defined at run time. The "Qualifier" bytes are test qualifiers that only pass following bytes when the first byte of the incoming string match them. The "Timeout" value is in msecs and execution jumps to "Label" when the "Timeout" interval passes without any data being received. "Mode" is used to specify the operation of the Pin and is defined in the table below:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Baud Rate</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>T300</td>
<td>300</td>
<td>Positive</td>
</tr>
<tr>
<td>T1200</td>
<td>1200</td>
<td>Positive</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Mode, (ParityLabel,)</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>SERIN2 Pin(\FlowPin),</td>
<td>Receive one or more asynchronous data bytes on &quot;Pin&quot;.</td>
<td>&quot;FlowPin&quot; is used to control the input of data to the PICmicro MCU to make sure there is no overrun. If Even Parity is selected in the &quot;Mode&quot; Parameter, then any time an invalid byte is received, execution will jump to the &quot;ParityLabel&quot;. Input Timeouts can be specified in 1 msec intervals with no data received in the specified period causing execution to jump to &quot;Label&quot;. &quot;Mode&quot; selection is made by passing a sixteen bit variable to the SERIN2 instruction. The bits are defined as:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Mode</th>
<th>Timeout</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2400</td>
<td>2400</td>
<td>Positive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T9600</td>
<td>9600</td>
<td>Positive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N300</td>
<td>300</td>
<td>Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N1200</td>
<td>1200</td>
<td>Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N2400</td>
<td>2400</td>
<td>Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N9600</td>
<td>9600</td>
<td>Negative</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

- Positive
- Negative
DEC[1..5] Var  Receive Up to 5 Decimal Digits and store in "Var"
HEX[1..4] Var  Receive Up to 4 Hexadecimal Digits and store in "Var"
SKIP #       Skip "#" Received Characters
STEM Array\n\c      Receive a string of "n" characters and store in "Array". Optionally ended by character "c"
WAIT("String") Wait for the Specified "String" of Characters
WAITSTR Array\n      Wait for a Character String "n" characters long
SEROUT Pin,Mode, [Value[,Value...]] Send one or more asynchronous data bytes on "Pin". The "Pin" can be defined at run time. "Mode" is used to specify the operation of the Pin and the output driver and is defined in the table below:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Baud Rate</th>
<th>State</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>T300</td>
<td>300</td>
<td>Positive</td>
<td>CMOS</td>
</tr>
<tr>
<td>T1200</td>
<td>1200</td>
<td>Positive</td>
<td>CMOS</td>
</tr>
<tr>
<td>T2400</td>
<td>2400</td>
<td>Positive</td>
<td>CMOS</td>
</tr>
<tr>
<td>T9600</td>
<td>9600</td>
<td>Positive</td>
<td>CMOS</td>
</tr>
<tr>
<td>N100</td>
<td>300</td>
<td>Negative</td>
<td>CMOS</td>
</tr>
<tr>
<td>N1200</td>
<td>1200</td>
<td>Negative</td>
<td>CMOS</td>
</tr>
<tr>
<td>N2400</td>
<td>2400</td>
<td>Negative</td>
<td>CMOS</td>
</tr>
<tr>
<td>N9600</td>
<td>9600</td>
<td>Negative</td>
<td>CMOS</td>
</tr>
<tr>
<td>OT300</td>
<td>300</td>
<td>Positive</td>
<td>Open-Drain</td>
</tr>
<tr>
<td>OT1200</td>
<td>1200</td>
<td>Positive</td>
<td>Open-Drain</td>
</tr>
<tr>
<td>OT2400</td>
<td>2400</td>
<td>Positive</td>
<td>Open-Drain</td>
</tr>
<tr>
<td>OT9600</td>
<td>9600</td>
<td>Positive</td>
<td>Open-Drain</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td>Baud Rate</td>
<td>State</td>
<td>Driver</td>
</tr>
<tr>
<td>ON300</td>
<td>300</td>
<td>Negative</td>
<td>Open-Drain</td>
</tr>
<tr>
<td>ON1200</td>
<td>1200</td>
<td>Negative</td>
<td>Open-Drain</td>
</tr>
<tr>
<td>ON2400</td>
<td>2400</td>
<td>Negative</td>
<td>Open-Drain</td>
</tr>
<tr>
<td>ON9600</td>
<td>9600</td>
<td>Negative</td>
<td>Open-Drain</td>
</tr>
</tbody>
</table>

SEROUT2 Pin[{FlowPin}], Send one or more asynchronous data bytes on "Pin". "FlowPin" is used to control the output of data to the PICmicro® MCU to make sure there is no overrun. timeouts can be specified in 1 msec intervals with no "Flow" control on the receiver the specified period causing execution to jump to "Label". The optional "Pace" parameter is used to specify the length of time (measured in usecs) that the PICmicro® MCU delays before sending out the next character. "Mode" selection is made by passing a sixteen bit variable to the SERIN2 instruction. The bits are defined as:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CMOS/Open Drain Driver</td>
</tr>
<tr>
<td>14</td>
<td>Set if Input Data is Negative</td>
</tr>
<tr>
<td>13</td>
<td>Set if Even Parity is to be used with the Data</td>
</tr>
<tr>
<td>12-0</td>
<td>Data Rate Specification, found by the formula: Rate = (1,000,000/Baud) - 20</td>
</tr>
</tbody>
</table>

The "Specification" is a string of data qualifiers/modifiers and source values that are used to format the outgoing data. The output format data can be specified with an "I" prefix to indicate that the data type is to be sent before the data and the "S" prefix indicates that a sign ("-") indicator is sent for negative values. The qualifiers/modifiers are listed in [Specification].
the table below:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin{1...16} Var</td>
<td>Receive Up to 16 Binary Digits and store in &quot;Var&quot;</td>
</tr>
<tr>
<td>DEC{1...5} Var</td>
<td>Receive Up to 5 Decimal Digits and store in &quot;Var&quot;</td>
</tr>
<tr>
<td>HEX{1...4} Var</td>
<td>Receive Up to 4 Hexadecimal Digits and store in &quot;Var&quot;</td>
</tr>
<tr>
<td>SKIP #</td>
<td>Skip &quot;#&quot; Received Characters</td>
</tr>
<tr>
<td>STR Array\n\c</td>
<td>Receive a string of &quot;n&quot; characters and store in &quot;Array&quot;. Optionally ended by character &quot;c&quot;</td>
</tr>
<tr>
<td>WAIT(&quot;String&quot;)</td>
<td>Wait for the Specified &quot;String&quot; of Characters</td>
</tr>
<tr>
<td>WAITSTR Array\n</td>
<td>Wait for a Character String &quot;n&quot; characters long</td>
</tr>
</tbody>
</table>

SHIFTIN DataPin, ClockPin, Mode, [Variable{B|s}]

Synchronously shift data into the PICmicro® MCU.

The "Bits" Parameter is used to specify the number of bits that are actually shifted in (if "Bits" is not specified, the default is 8). The "Mode" Parameter is used to indicate how the data is to be transferred and the values are listed in the table below:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSBPRE</td>
<td>Most Significant Bit First, Read Data before pulsing Clock</td>
</tr>
<tr>
<td>LSBPRE</td>
<td>Least Significant Bit First, Read Data before pulsing Clock</td>
</tr>
<tr>
<td>MSBPOST</td>
<td>Most Significant Bit First, Read Data after</td>
</tr>
</tbody>
</table>
### Instruction Description

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode Function</td>
<td>Least Significant Bit First, Read Data after pulsing Clock</td>
</tr>
<tr>
<td>SHIFTOUT DataPin, ClockPin, Mode, [Variable{{Notes} Duration} {Note, Duration. . .}]</td>
<td>Synchronously shift data out of the PICmicro® MCU. The &quot;Bits&quot; Parameter is used to specify how many bits are to be shifted out in each word (if not specified, the default is 8). The &quot;Mode&quot; parameter is used to specify how the data is to be shifted out and the values are listed in the table below:</td>
</tr>
<tr>
<td>LSBFIRST</td>
<td>Least Significant Bit First</td>
</tr>
<tr>
<td>MSBFIRST</td>
<td>Most Significant Bit First</td>
</tr>
<tr>
<td>SLEEP Period</td>
<td>Put the PICmicro® MCU into &quot;Sleep&quot; mode for &quot;Period&quot; seconds</td>
</tr>
<tr>
<td>SOUND Pin, [Note, Duration {Note, Duration. . .}]</td>
<td>Output a string of Tones and Durations (which can be used to create a simple tune) on the &quot;Pin&quot;. Note &quot;0&quot; is silence and Notes &quot;128&quot; to &quot;255&quot; are &quot;white noise&quot;. Note &quot;1&quot; (78.5 Hz for a 4 MHz PICmicro® MCU) is the lowest valid tone and note &quot;127&quot; is the highest (10 kHz in a 4 MHz PICmicro® MCU). Duration is specified in 12 msec increments</td>
</tr>
<tr>
<td>STOP</td>
<td>Place the PICmicro® MCU into an endless loop. The PICmicro® MCU is not put into &quot;Sleep&quot; mode</td>
</tr>
<tr>
<td>SWAP Variable, Variable</td>
<td>Exchange the values in the two Variables</td>
</tr>
<tr>
<td>TOGGLE Pin</td>
<td>Toggle the Output Value of the Specified Pin</td>
</tr>
<tr>
<td>WHILE Cond</td>
<td>Execute the code between the &quot;WHILE&quot; and the &quot;WEND&quot; statements while the &quot;Cond&quot; condition returns a non-zero value. Execution exits the loop when &quot;Cond&quot; is evaluated to Zero</td>
</tr>
<tr>
<td>WRITE Address, Value</td>
<td>Write the Byte &quot;Value&quot; into the built in Data EEPROM. This instruction will not work with the built in EEPROM in the</td>
</tr>
</tbody>
</table>
PIC12CExxx devices

XIN DataPin,ZeroPin,
{Timeout,Label,}
{Variable
{,Variable...}}

Receive data from X-10 devices. "ZeroPin" is
used to detect the "Zero Crossing" of the
input AC Signal. Both "DataPin" and "ZeroPin"
should be pulled up with 4.7 K resistors. The optional
Timeout (specified in 8.33 msec intervals) will cause
execution to jump to "Label" if no data is received by the
specified interval. If the first Variable data destination is
sixteen bits, then both the "House Code" and the "Key Code"
will be saved. If the first Variable is eight bits in size,
then only the "Key Code" will be saved.

XOUT DataPin,ZeroPin,
{HouseCode\KeyCode
{\Repeat}{,Value...}}

Send X-10 data to other devices. The "ZeroPin" is an input and
should be pulled up with a 4.7K resistor. "HouseCode" is a
number between 0 and 15 and corresponds to the "House Code"
set on the X-10 Modules A through P. The "KeyCode" can either
be the number of a specific X-10 receiver or the function to
be performed by the module.
Visual Basic

Microsoft’s “Visual Basic” is probably the fastest way to get into Microsoft “Windows” application programming. The ease of using the language and development system also makes it great as a “what if” tool and allows you to write an application quickly to try out new ideas.

To create an application, the Primary dialog box (which is known as a “form” and is shown in Fig. 15.8) is created first, with different features (I/O boxes, buttons, etc.). These features are known as “controls” within Visual Basic. With the Window defined, by simply clicking on the different controls, subroutine prototypes to handle “events” (such as mouse clicks over these features) are automatically created. Additional features in Visual Basic’s source code editor allow

Figure 15.8 “Visual Basic” Development System
you to specify the control parameters (known as “properties”). Visual Basic applications are built around “The Dialog Box Editor” desktop. When application development is started, Visual Basic provides you with the initial “Dialog” box of the application that can be seen in Fig. 15.9. From here, “Dialog Resources” are selected from the “ToolBox” and placed upon the dialog.

Control attributes (also known as “Properties”) can be set globally from the Integrated Development Environment or from within the “Event Handlers”. The event handler’s code is written in pretty standard Microsoft BASIC. Once the handler prototypes are created by Visual Basic, it is up to the application developer to add the response code for the application. Visual Basic provides a large number of built-in functions, including trigonometric functions, logarithm
functions, and the ability to interface with the file system and dialog controls.

Variables in Visual Basic are typically "Integer", which is to say they are sixteen bit values in the ranges $-32768$ to $+32765$. Thirty-two bit integer variables can be specified by putting a "%" character at the end of the variable label. One important thing to note about variables is that they are local to the event routine they are used in unless they are declared globally in the "General Module", which executes at the beginning of the application and is not specific to any controls.

There are a number of controls that are basic to Visual Basic with others being available for download off the Internet or bought which can make your Visual Basic applications more impressive and lend "pizzazz" to Windows applications.

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull Downs</td>
<td>Selected from the &quot;Menu Editor&quot; icon on the &quot;ToolBar&quot;</td>
</tr>
<tr>
<td>PictureBox</td>
<td>Display Bitmaps and other graphic files on the Dialog Box</td>
</tr>
<tr>
<td>Label</td>
<td>Put Text in the Dialog Box</td>
</tr>
<tr>
<td>TextBox</td>
<td>Input/Output Text Box</td>
</tr>
<tr>
<td>Frame</td>
<td>Put a Frame around Resources</td>
</tr>
<tr>
<td>CommandButton</td>
<td>Button for Code Operation</td>
</tr>
<tr>
<td>CheckBox</td>
<td>For Checking Multiple Selections</td>
</tr>
<tr>
<td>OptionButton</td>
<td>Also known as the &quot;Radio Button&quot;, For Checking one selection for a list of Multiple options</td>
</tr>
<tr>
<td>ComboBox</td>
<td>Select or Enter Test in a Box/List</td>
</tr>
<tr>
<td>ListBox</td>
<td>List Data (with User controlled Scrolling)</td>
</tr>
<tr>
<td>HScrollBar</td>
<td>Provide Horizontal Scrolling in a Text or Graphic Output Control</td>
</tr>
<tr>
<td>VScrollBar</td>
<td>Provide Vertical Scrolling in a Text or Graphic Output Control</td>
</tr>
<tr>
<td>Timer</td>
<td>Cause a periodic interrupt</td>
</tr>
</tbody>
</table>
There are a number of controls that cannot be activated with a left button click and cannot be “seen” on the application’s form. The one that is used the most is the “Timer”. This control causes an event after a set period of microseconds. This control can be set within the dialog editor or modified within the application itself. The Timer can provide many different advanced functions without requiring any interrupt interfaces.

**MSComm control**

The MSComm control recommended initialization sequence is

1. Specify the Hardware Serial Port to be used.
2. Set the speed and data format to be used.
3. Define the buffer size.
4. Open the port and begin to use it.

The instructions used to perform these functions are placed in the “Form_Load” subroutine, which means the port is enabled before the primary dialog box is executing. The following code is an example of an MSComm object initialization:
Private Sub Form_Load()
' On Form Load, Setup Serial Port 3 for YAP Programmer
    MSComm3.CommPort = 3
    MSComm3.Settings = "1200,N,8,1"
    MSComm3.InputLen = 0
    MSComm3.PortOpen = True
    Text1.Text = "Turn on YAP Programmer"
End Sub

A 50-msec timer can be used to continually poll the serial port and display data in the "Text" Box when it is received:

Private Sub Timer1_Timer()
' Interrupt every 50 msecs and Read in the Buffer
    Dim InputString
    InputString = MSComm3.Input
    If (InputString <> "") Then
        If (Text1.Text = "Turn on YAP Programmer") Then
            Text1.Text = "" ' Clear the Display Buffer
        End If
        Text1.Text = Text1.Text + InputString
    End If
End Sub

Once the "MSComm" control is placed on the display, the following properties are used to control it:

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break</td>
<td>True/False</td>
<td>When set to &quot;True&quot;, Break Sends a &quot;0&quot; break signal until the property is changed to &quot;False&quot;.</td>
</tr>
</tbody>
</table>
CDHolding True/False Read only property that indicates if the “Carrier Detect” line is active. This is an important line to poll in applications which use modems.

CommEvent Integer Read only property that is only available while the application is running. If the application is running without any problems, this property returns zero. This property is read by the “OnComm” event handler code to process the reason why the “event” was caused.

CommID Object Read only property that returns an identifier for the serial port assigned to the MSComm control.

CommPort Integer Specify the “COMx” (1-3) serial port that is used by the MSComm control.

CTSHolding True/False Read only property that returns the current state of the serial port’s “Clear To Send” line.

DSRHolding True/False Read only property that returns the current state of the serial port’s “Data Set Ready” line.

DTREnable True/False Property used to specify the state of the “Data Terminal Ready” line.

EOFEnable True/False Specify whether or not an “OnComm” event will be generated if an “End Of File” character (0x01A) is encountered.
584 Chapter 15

Handshaking  0, 1, 2 or 3        Sets the current
            handshaking protocol for
            the serial port:
            0 - No handshaking
            (default)
            1 - XON/XOFF          
            Handshaking         
            2 - RTS/CTS (Hardware)
            Handshaking         
            3 - Both XON/XOFF and 
            RTS/CTS              
            Handshaking         

InBufferCount    Integer        Read only property
            indicating how many
            characters have been
            received by the serial
            port.

InBufferSize    Integer        Property used to specify
            the number of bytes
            available for the Input
            Data Buffer. The default
            size is 1024 bytes.

Input            String         Return a String of
            Characters from the
            Input Buffer.

InputMode       Integer        Specify how data is to be
            retrieved using the
            "Input" property. Zero
            specifies data will be
            received as Text
            (Default). One will
            specify that data will
            be passed without
            editing ("Binary" format).

InputLen        Integer        Sets the Maximum Number
            of characters that will
            be returned when the
            "Input" property is
            accessed. Setting this
            value to zero will return
            the entire buffer.
NullDiscard | True/False | Specify whether or not Null Characters are transferred from the port to the receiver buffer.

OutBufferCount | Integer | Read only property that returns the Number of Characters waiting in the Output Buffer.

OutBufferSize | Integer | Specify the size of the Output Buffer. The default is 512 Bytes.

Output | Integer | Output a string of characters through the serial port.

ParityReplace | Integer | Specify the character that will replace characters which have a “Parity” Error. The default character is “?” and the ASCII code for the replacement character must be specified.

PortOpen | True/False | Specify whether or not the data port is to be transmitting and receiving data. Normally a port is closed (“False”).

Rthreshold | Integer | Specify the number of characters before there is an “OnComm” event. The default value of zero disables event generation. Setting the “Rthreshold” to one will cause an “OnComm” event each time a character is received.
586 Chapter 15

RTS Enable True/False

Specify the value output on the "Request To Send" line.

Settings String

Send a String to the Serial Port to specify its operating characteristics. The String is in the format "Speed, Parity, Length, Stop" with the following valid parameter values:

- **Speed**: Data Rate of the Communication
  - 110
  - 300
  - 600
  - 1200
  - 2400
  - 9600 (Default)
  - 14400
  - 19200
  - 28800
  - 38400
  - 56000
  - 128000
  - 256000

- **Parity**: The type of error checking sent with the byte
  - E - Even Parity
  - M - Mark Parity
  - N - No Parity (Default)
  - O - Odd Parity
  - S - Space Parity

- **Length**: The number of bits transmitted at a time
  - 4 - 4 Bits
  - 5 - 5 Bits
  - 6 - 6 Bits
  - 7 - 7 Bits
  - 8 - 8 Bits (Default)

- **Stop**: The number of stop bits transmitted with the byte
Further enhancing the usefulness of the “MSComm” control is the “OnComm” event. This routine is similar to an interrupt, as it is requested after specified events in the serial port. The “CommEvent” property contains the reason code for the event. These codes include:

<table>
<thead>
<tr>
<th>CommEvent Identifier</th>
<th>CommEvent Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comEvSend</td>
<td>1</td>
<td>Specified Number of Characters Sent</td>
</tr>
<tr>
<td>comEvReceive</td>
<td>2</td>
<td>Specified Number of Characters Received</td>
</tr>
<tr>
<td>comEvCTS</td>
<td>3</td>
<td>Change in the “Clear To Send” line</td>
</tr>
<tr>
<td>comEvDSR</td>
<td>4</td>
<td>Change in the “Data Set Ready” line</td>
</tr>
<tr>
<td>comEvCD</td>
<td>5</td>
<td>Change in the “Carrier Detect” line</td>
</tr>
<tr>
<td>comEvRing</td>
<td>6</td>
<td>Ring Detect is Active</td>
</tr>
<tr>
<td>comEvEOF</td>
<td>7</td>
<td>“End Of File” Character Detected</td>
</tr>
</tbody>
</table>
### The "C" Language Declarations

**Constant declaration:**

```c
const int Label = Value;
```

**Variable declaration:**

```c
type Label [= Value];
```

"Value" is an optional Initialization Constant, where "type" can be:

- `char`
- `int`
- `unsigned int`
- `float`
Note that “int” is defined as the “word size” of the processor/operating system. For PCs, an “int” can be a Word (16 bits) or a Double Word (32 bits). For the PICmicro® MCU, an “int” is normally 8 bits (a byte).

There may also be other basic types defined in the language implementation. Single dimensional arrays are declared using the form:

```
type Label[ Size ] [= { Initialization Values..}];
```

Note that the array “Size” is enclosed within square brackets (“[” and “]”) and should not be confused with the optional “Initialization Values”.

Strings are defined as single dimensional ASCIIZ arrays:

```
char String[ 17 ] = "This is a String";
```

where the last character is an ASCII “NUL”.

Strings can also be defined as pointers to characters:

```
char *String = "This is a String";
```

although this implementation requires the text “This is a String” to be stored in two locations (in code and data space). For the PICmicro® MCU and other Harvard architected processors, the text data could be written into data space when the application first starts up as part of the language’s initialization.

Multidimensional Arrays are defined with each dimension separately identified within Square Brackets (“[” and “]”):

```
int ThreeDSpace[ 32 ][ 32 ][ 32 ];
```

Array Dimensions must be specified unless the Variable is a pointer to a Single Dimensional Array.
Pointers are declared with the "*" character after the "type"

```
char * String = "This is a String";
```

Accessing the address of the Pointer in Memory is accomplished using the "&" character:

```
StringAddr = &String;
```

Accessing the address of a specific element in a String is accomplished using the "&" character and a String Array Element:

```
StringStart = &String[n];
```

In the PC running MS-DOS, it is recommended that "far" (32 bit) pointers be always used with absolute offset:segment addresses within the PC memory space to avoid problems with varying segments. In the PICmicro® MCU, all addresses can be specified with two bytes.

The Variable’s “Type” can be “overridden” by placing the new type in front of the variable in single brackets:

```
(long) StringAddr = 0x0123450000;
```

**Statements**

**Application “Mainline”**.

```
main(envp)
  char *envp;
  {  // Application Code
      // Application Code
  }  // End Application
```
Function format.
Return_Type Function( Type Parameter [, Type Parameter..])
{   //  Function Start
    // Function Code
    return value;
} //  End Function

Function prototype.
Return_Type Function( Type Parameter [, Type Parameter..]);

Expression.
[(..] Variable | Constant [Operator [(..] Variable | Constant [][)]

Assignment statement.
Variable = Expression;

“C” conditional statements (consisting of “if”, “?”, “while”, “do”, “for” and “switch”).
The “if” statement is defined as

    if { Statement }
    ; | { Assignment Statement | Conditional Statement.. } | Assignment Statement | Conditional Statement
[else ;] { Assignment Statement | Conditional Statement.. } | Assignment Statement | Conditional Statement

The “?:” statement evaluates the statement (normally a comparison) and if not equal to zero, executes the first statement, else executes the statement after the “:”.

The “while” statement is added to the application following the definition below:

\[ \text{while ( } \text{Statement} \text{ ) ; } | \{ \text{Assignment Statement } | \text{Conditional Statement... } \} | \text{Assignment Statement } | \text{Conditional Statement} \]

The “for” statement is defined as:

\[ \text{for } \{ \text{initialization (Assignment) Statement; } \text{Conditional Statement; } \text{Loop Expression (Increment) Statement } \} | \{ \text{Assignment Statement } | \text{Conditional Statement... } \} | \text{Assignment Statement } | \text{Conditional Statement} \]

To jump out of a currently executing loop, “break” statement

\[ \text{break; } \]

is used.

The “continue” statement skips over remaining code in a loop and jumps directly to the loop condition (for use with “while”, “for” and “do/while” loops). The format of the statement is

\[ \text{continue;} \]

For looping until a condition is true, the “do/while” statement is used:

\[ \text{do} \]
\[ \text{Assignment Statement } | \text{Conditional Statement... } \]
\[ \text{while ( Expression ); } \]
To conditionally execute according to a value, the "switch" statement is used:

```c
switch( Expression ) {
    case Value:              // Execute if "Statement"
        // == "Value"
        [ Assignment Statement | Conditional Statement... ]
        [break;]
    default:                 // If no "case" Statements
        // are True
        [ Assignment Statement | Conditional Statement... ]
} // End switch
```

Finally, the "goto Label" statement is used to jump to a specific address:

```c
goto Label;
Label:
```

To return a value from a function, the "return" statement is used:

```c
return Statement;
```

### Operators

**Statement operators.**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical Negation</td>
</tr>
<tr>
<td>!</td>
<td>Bitwise Negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND, Address</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
</tr>
</tbody>
</table>
Subtraction, Negation
--- Decrement
* Multiplication, Indirection
/ Division
% Modulus
== Equals
!= Not Equals
< Less Than
<= Less Than or Equals To
<< Shift Left
> Greater Than
>= Greater Than or Equals To
>> Shift Right

Compound assignment operators.

Operator        Operation
&= AND with the Variable and Store Result in the Variable
|= OR with the Variable and Store
^= XOR with the Variable and Store
+= Add to the Variable
-= Subtract from the Variable
*= Multiply to the Variable
/= Divide from the Variable
%= Get the Modulus and Store in the Variable
<<< Shift Left and Store in the Variable
>>> Shift Right and Store in the Variable

Order of operations.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Priority</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] . -&gt;</td>
<td>Highest</td>
<td>Expression</td>
</tr>
<tr>
<td>- + * / %</td>
<td>Evaluation</td>
<td>Unary Operators</td>
</tr>
<tr>
<td>&lt; &lt;= &gt;= &gt;&gt;</td>
<td>Additive</td>
<td>Multiplicative</td>
</tr>
<tr>
<td>&lt; &lt;= &gt;= &gt;&gt;</td>
<td>Shifting</td>
<td>Comparison</td>
</tr>
</tbody>
</table>
### Directives

All Directives start with “#” and are executed before the code is compiled.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define Label[( Parameters)] Text</td>
<td>Define a Label that will be replaced with &quot;Text&quot; when it is found in the code. If &quot;Parameters&quot; are specified, then replace them in the code, similar to a macro.</td>
</tr>
<tr>
<td>#undefine Label</td>
<td>Erase the defined Label and Text in Memory.</td>
</tr>
<tr>
<td>#include &quot;File&quot;</td>
<td>Load the Specified File in Line to the Text. When &quot;&lt;&quot; &quot;&gt;&quot; encloses the Filename, then the file is found using the &quot;INCLUDE&quot; Environment Path Variable. If &quot;&quot; encloses the Filename, then the</td>
</tr>
</tbody>
</table>
596  Chapter 15

file in the
current directory
is searched before
checking the
"INCLUDE" Path.

#define Name           If the "Name" exists, then
Compile the
Following Code.

#define Name           If the "Name" does NOT
exist, then
Compile the
Following Code.

This Directive
works as an "else
#ifndef" to avoid
lengthy nested
"if"s.

If the previous
condition was
False, checks the
Condition.
The following words cannot be used in “C” applications as labels:

break
case
continue
default
do
else
for
goto
if
return
switch
while
598  Chapter 15

"Backslash" characters

<table>
<thead>
<tr>
<th>String</th>
<th>ASCII</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r</td>
<td>0x00D</td>
<td>Carriage Return</td>
</tr>
<tr>
<td>\n</td>
<td>0x00A</td>
<td>Line Feed</td>
</tr>
<tr>
<td>\f</td>
<td>0x00C</td>
<td>Form Feed</td>
</tr>
<tr>
<td>\b</td>
<td>0x008</td>
<td>Backspace</td>
</tr>
<tr>
<td>\t</td>
<td>0x009</td>
<td>Horizontal Tab</td>
</tr>
<tr>
<td>\v</td>
<td>0x00B</td>
<td>Vertical Tab</td>
</tr>
<tr>
<td>\a</td>
<td>0x007</td>
<td>Bell</td>
</tr>
<tr>
<td>'</td>
<td>0x027</td>
<td>Single Quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>0x022</td>
<td>Double Quote</td>
</tr>
<tr>
<td>\ \</td>
<td>0x03C</td>
<td>Backslash</td>
</tr>
<tr>
<td>\ddd</td>
<td>N/A</td>
<td>Octal Number</td>
</tr>
<tr>
<td>\xddd</td>
<td>0x0dd</td>
<td>Hexadecimal Character</td>
</tr>
</tbody>
</table>

Common C functions

As Defined by Kernighan and Ritchie:

Function          Operation
int getchar( void ) Get one Character from
                    "Standard Input" (the Keyboard). If no
                    Character available, then wait for it.
int putchar( int )  Output one Character to
                    the "Standard Output"
                    (the Screen).
int printf( char *Const[, arg...]) Output the "Const" String
                    Text. "Escape Sequence"
                    Characters for Output
                    are embedded in the
                    "Const" String Text.
                    Different Data Outputs
                    are defined using the
                    "Conversion Characters":
                    %d, %i - Decimal Integer
                    %o - Octal Integer
                    %x, %X - Hex Integer
                    (with upper or lower
                    case values). No leading
                    "0x" character String
                    Output
int scanf(char *Const, arg[, *arg...])

Provide Formatted Input
from the user. The
"Const" ASCIIZ String is
used as a "Prompt" for
the user. Note that the
input parameters are
always pointers.

"Conversion Characters"
are similar to "printf":
%d - Decimal Integer
%i - Integer. In Octal
if leading "0" or hex
if leading "0x" or "0X"
%o - Octal Integer
(Leading "0" Not
Required)
%x - Hex Integer
(Leading "0x" or "0X"
Not Required)
%c - Single Character
%s - ASCIIZ String of
Characters. When Saved,
a NULL character is put
at the end of the String
%f, %l, %g - Floating
Point Value with
optional sign, decimal
point and exponent
%% - Display "%" character in prompt
600  Chapter 15

handle fopen( char *FileName, char *mode )
Open File and Return Handle (or NULL for Error).
"mode" is a String consisting of the optional characters:
 r - Open File for Reading
 w - Open File for Writing
 a - Open File for Appending to Existing Files
Some systems handle "Text" and "Binary" files. A "Text" file has the CR/LF characters represented as a single CR. A "Binary" file does not delete any characters.

int fclose( handle )
Close the File.

int getc( handle )
Receive data from a file one character at a time. If at the end of an input file, then "EOF" is returned.

int putc( handle, char )
Output data to a file one character at a time. Error is indicated by "EOF" returned.

int fprintf( handle, char *Const[, arg...])
Output String of Information to a File. The same "Conversion Characters" and arguments as "printf" are used.

int fscanf( handle, char *Const, arg[, arg...])
Input and Process String of Information from a File. The same "Conversion Characters"
and arguments as "scanf" are used.

```c
int fgets( char *Line, int LineLength, handle )
Get the ASCII String from the file.

int fputs( char *line, handle )
Output an ASCII String to a file.

strcat( Old, Append )
Put ASCII "Append" String on the end of the "Old" ASCII String.

strncat( Old, Append, # )
Put "#" of characters from "Append" on the end of the "Old" ASCII String.

int strcmp( String1, String2 )
Compare two ASCII Strings. Zero is returned for match, negative for "String1" < "String2" and positive for "String1" > "String2".

int strncmp( String1, String2, # )
Compare two ASCII Strings for "#" characters. Zero is returned for match, negative for "String1" < "String2" and positive for "String1" > "String2".

strcpy( String1, String2 )
Copy the Contents of ASCII "String2" into "String1".

strncpy( String1, String2, # )
Copy "#" Characters from "String2" into "String1".

strlen( String )
Return the length of ASCII Character "String".

int strchr( String, char )
Return the Position of the first "char" in the ASCII "String".
602 Chapter 15

int strrchr(String, char) Return the Position of the last “char” in the ASCII “String”.

system(String) Executes the System Command “String”.

malloc(size) Allocate the Specified Number of Bytes of Memory. If insufficient space available, return NULL.

calloc(#, size) Allocate Memory for the specified “#” of data elements of “size”.

free(*) Free the Memory.

float sin(angle) Find the “Sine” of the “angle” (which in Radians).

float cos(angle) Find the “Cosine” of the “angle” (which in Radians).

float atan2(y, x) Find the “Arctangent” of the “X” and “Y” in Radians.

float exp(x) Calculate the natural exponent.

float log(x) Calculate the natural logarithm.

float log10(x) Calculate the base 10 logarithm.

float pow(x, y) Calculate “x” to the power “y”.

float sqrt(x) Calculate the Square Root of “x”.

float fabs(x) Calculate the Absolute Value of “x”.

float frand() Get a Random Number.

int isalpha(char) Return Non-Zero if Character is “a”-“z” or “A”-“Z”.

int isupper(char) Return Non-Zero if Character is “A”-“Z”.

int islower(char) Return Non-Zero if Character is “a”-“z”.

int isdigit(char) Return Non-Zero if Character is “0”-“9”.
### **PICmicro® MCU Enhancement Functions**

Useful Functions in PICmicro® MCU C implementations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>inp, outp</td>
<td>Provide method for directly accessing system registers.</td>
</tr>
<tr>
<td>SerIn, SerOut</td>
<td>NS2 Non-Return to Zero</td>
</tr>
<tr>
<td>I2C</td>
<td>I2C Interface</td>
</tr>
<tr>
<td>PFM</td>
<td>Measure/output PFM signals</td>
</tr>
</tbody>
</table>
This page intentionally left blank.
# Mathematical and Physical Constants

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU</td>
<td>149.59787x(10^6) km</td>
<td>Astronomical Unit (Distance from the Sun to the Earth)</td>
</tr>
<tr>
<td></td>
<td>92,955,628 miles</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2.99792458x(10^8) m/s</td>
<td>Speed of Light in a Vacuum</td>
</tr>
<tr>
<td>e</td>
<td>2.7182818285</td>
<td>Permittivity of Free Space</td>
</tr>
<tr>
<td>Epsilon-o</td>
<td>8.854187817x(10^-12) F/m</td>
<td></td>
</tr>
</tbody>
</table>
606 Chapter 16

- **Ev**: $1.60217733 	imes 10^{-19}$ J, Electron Volt Value
- **g**: 32.174 ft/sec^2, Acceleration due to gravity
- **h**: $6.626 	imes 10^{-34}$ Js, Planck Constant
- **k**: $1.380658 	imes 10^{-23}$ J/K, Boltzmann Entropy Constant
- **me**: $9.1093897 	imes 10^{-31}$ kg, Electron Rest Mass
- **mn**: $1.67493 	imes 10^{-27}$ kg, Neutron Rest Mass
- **mp**: $1.67263 	imes 10^{-27}$ kg, Proton Rest Mass
- **pc**: 2.06246x10^5 AU, Parsec
- **pi**: 3.1415926535898, Ratio of circumference to Diameter of a circle
- **R**: 8.314510 J/(K * mol), Gas Constant
- **sigma**: $5.67051 	imes 10^{-8}$ W/(m^2 * K^4), Stefan-Boltzmann Constant
- **u**: $1.66054 	imes 10^{-27}$ grams, Atomic Mass Unit
- **mu-o**: $1.25664 	imes 10^{-7}$ N/A^2, Permeability of Vacuum
- **None**: 331.45 m/s, Speed of Sound at Sea Level, in Dry Air at 20C
- **None**: 1480 m/s, Speed of Sound in Water at 20C

ASCII

The ASCII Definition uses the seven bits of each ASCII character.

<table>
<thead>
<tr>
<th>3-0</th>
<th>6-4</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Control</td>
<td>Characters</td>
<td>*</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>NUL</td>
<td>DLE</td>
<td>Space</td>
<td>0</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>DC1</td>
<td>DC2</td>
<td>HT</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
<td>B</td>
<td>R</td>
</tr>
<tr>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
<td>D</td>
<td>t</td>
<td>E</td>
</tr>
<tr>
<td>BS</td>
<td>FS</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
<td>E</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>SOH</td>
<td>DC1</td>
<td>$</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
<td>U</td>
<td>D</td>
</tr>
</tbody>
</table>
### ASCII control characters

The ASCII Control Characters were specified as a means of allowing one computer to communicate and control another. These characters are actually commands and if the BIOS or MS-DOS display or communications APIs are used with them they will revert back to their original purpose. Writing these values (all less than 0x020) to the display will display graphics characters in the IBM PC.

Normally, only "Carriage Return"/"Line Feed" are used to indicate the start of a line. "Null" is used to indicate the end of an ASCII string. "Backspace" will move the cursor back one column to the start of the line. The "Bell" character, when sent to MS-DOS will cause the PC's speaker to "beep". "Horizontal Tab" is used to move the cursor to the start of the next column that is evenly distributed by eight. "Form Feed" is used to clear the screen.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>NUL</td>
<td>&quot;Null&quot; - Used to indicate the end of a string</td>
</tr>
<tr>
<td>01</td>
<td>SOH</td>
<td>Message &quot;Start of Header&quot;</td>
</tr>
<tr>
<td>02</td>
<td>STX</td>
<td>Message &quot;Start of Text&quot;</td>
</tr>
<tr>
<td>03</td>
<td>ETX</td>
<td>Message &quot;End of Text&quot;</td>
</tr>
<tr>
<td>Code</td>
<td>Character</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>04</td>
<td>EOT</td>
<td>&quot;End of Transmission&quot;</td>
</tr>
<tr>
<td>05</td>
<td>ENQ</td>
<td>&quot;Enquire&quot; for Identification or Information</td>
</tr>
<tr>
<td>06</td>
<td>ACK</td>
<td>&quot;Acknowledge&quot; the previous transmission</td>
</tr>
<tr>
<td>07</td>
<td>BEL</td>
<td>Ring the &quot;BELL&quot;</td>
</tr>
<tr>
<td>08</td>
<td>BS</td>
<td>&quot;Backspace&quot; - Move the Cursor on column to the left</td>
</tr>
<tr>
<td>09</td>
<td>HT</td>
<td>&quot;Horizontal Tab&quot; - Move the Cursor to the Right to the next &quot;Tab Stop&quot; (Normally a column evenly divisible by eight)</td>
</tr>
<tr>
<td>0A</td>
<td>LF</td>
<td>&quot;Line Feed&quot; - Move the Cursor down one line</td>
</tr>
<tr>
<td>0B</td>
<td>VT</td>
<td>&quot;Vertical Tab&quot; - Move the Cursor down to the next &quot;Tab Line&quot;</td>
</tr>
<tr>
<td>0C</td>
<td>FF</td>
<td>&quot;Form Feed&quot; up to the start of the new page. For CRT displays, this is often used to clear the screen</td>
</tr>
<tr>
<td>0D</td>
<td>CR</td>
<td>&quot;Carriage Return&quot; - Move the Cursor to the leftmost column</td>
</tr>
<tr>
<td>0E</td>
<td>SO</td>
<td>Next Group of Characters do not follow ASCII Control conventions so they are &quot;Shifted Out&quot;</td>
</tr>
<tr>
<td>0F</td>
<td>SI</td>
<td>The following Characters do follow the ASCII Control conventions and are &quot;Shifted In&quot;</td>
</tr>
<tr>
<td>10</td>
<td>DLR</td>
<td>&quot;Data Link Escape&quot; - ASCII Control Character start of an Escape sequence. In most modern applications &quot;Escape&quot; (0x01B) is used for this function</td>
</tr>
<tr>
<td>11</td>
<td>DC1</td>
<td>Not defined - Normally application specific</td>
</tr>
<tr>
<td>12</td>
<td>DC2</td>
<td>Not defined - Normally application specific</td>
</tr>
<tr>
<td>13</td>
<td>DC3</td>
<td>Not defined - Normally application specific</td>
</tr>
<tr>
<td>14</td>
<td>DC4</td>
<td>Not defined - Normally application specific</td>
</tr>
<tr>
<td>15</td>
<td>NAK</td>
<td>&quot;Negative Acknowledge&quot; - the previous transmission was not properly received</td>
</tr>
<tr>
<td>16</td>
<td>SYN</td>
<td>&quot;Synchronous Idle&quot; - If the serial transmission uses a synchronous&quot;</td>
</tr>
</tbody>
</table>
protocol, this character is sent to ensure the transmitter and receiver remain synched.

17 ETB "End of Transmission Block"
18 CAN "Cancel" and disregard the previous transmission
19 EM "End of Medium" - Indicates end of a file. For MS-DOS files, 0x01A is often used instead.
1A SUB "Substitute" the following character with an incorrect one.
1B ESC "Escape" - Used to temporarily halt execution or put an application into a mode to receive information.
1C FS Marker for "File Separation" of data being sent.
1D GS Marker for "Group Separation" of data being sent.
1E RS Marker for "Record Separation" of data being sent.
1F US Marker for "Unit Separation" of data being sent.

**ANSI display control sequences**

From MS-DOS applications you can move the cursor or change the current display colors one of two ways. Normally I use the BIOS functions and direct writes to video RAM. The second way is to load the "ANSI.SYS" device driver in the "config.sys" using the statement:

```
device = [d:][path]ANSI.SYS
```

When the "Escape Sequences" listed below are output using the standard output device (using the MS-DOS APIs), the commands are executed. This method is not often used for two reasons. The first is that it is much slower than using the BIOS APIs and writing directly to video RAM. For an application that
seems to change the screen in the blink of an eye, the ANSI Display Control Sequences are not the way to do it. The second is that "ANSLSYS" takes away 10 KBytes of memory that would normally be available for applications.

There are two advantages to using the ANSI Display Control Sequences. The first is that it will make applications very portable. Passing the source to another system's just requires recompilation and linking. The second advantage is that sending data serially to a receiver set up able to receive these sequences (set up as an "ANSI" or "VT100 Compatible" Terminal), will provide simple graphic operations in an application.

In the table below, "ESC" is the ASCII "Escape" Character 0x01B.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Esc[#h</td>
<td>Set the PC's Display mode. This is not available in &quot;true&quot; ANSI compatible devices.</td>
</tr>
<tr>
<td># = 0 - 40x25 Monochrome</td>
<td></td>
</tr>
<tr>
<td># = 1 - 40x25 Color</td>
<td></td>
</tr>
<tr>
<td># = 2 - 80x25 Monochrome</td>
<td></td>
</tr>
<tr>
<td># = 3 - 80x25 Color</td>
<td></td>
</tr>
<tr>
<td># = 4 - 320x200 Color Graphics</td>
<td></td>
</tr>
<tr>
<td># = 5 - 320x200 Monochrome Graphics</td>
<td></td>
</tr>
<tr>
<td># = 6 - 640x200 Monochrome Graphics</td>
<td></td>
</tr>
<tr>
<td># = 7 - wrap to next line at line end</td>
<td></td>
</tr>
<tr>
<td># = 14 - 640x200 Color Graphics</td>
<td></td>
</tr>
<tr>
<td># = 15 - 640x350 Monochrome Graphics</td>
<td></td>
</tr>
<tr>
<td># = 16 - 640x480 Color Graphics</td>
<td></td>
</tr>
<tr>
<td># = 17 - 640x480 Color Graphics</td>
<td></td>
</tr>
<tr>
<td># = 18 - 640x480 Color Graphics</td>
<td></td>
</tr>
<tr>
<td># = 19 - 320x200 Color Graphics</td>
<td></td>
</tr>
<tr>
<td>Esc[#l</td>
<td>Reset the PC's Display mode. This is not available in &quot;true&quot; ANSI compatible devices.</td>
</tr>
<tr>
<td># = 0 - 40x25 Monochrome</td>
<td></td>
</tr>
<tr>
<td># = 1 - 40x25 Color</td>
<td></td>
</tr>
<tr>
<td># = 2 - 80x25 Monochrome</td>
<td></td>
</tr>
<tr>
<td># = 3 - 80x25 Color</td>
<td></td>
</tr>
</tbody>
</table>
Data Tables 611

# = 4 - 320x200 Color Graphics
# = 5 - 320x200 Monochrome Graphics
# = 6 - 640x200 Monochrome Graphics
# = 7 - do not wrap at line end

Esc[#m Set Character Attributes
# = 0 - Normal (gray on black)
# = 1 - Intensity Bit set for Foreground Colors
# = 4 - Underscore Characters in MDA
# = 5 - Blink Characters in MDA
# = 7 - Reverse the Character Foreground Color with the background
# = 8 - Make MDA Characters Invisible
# = 30 - Black Foreground
# = 31 - Red Foreground
# = 32 - Green Foreground
# = 33 - Yellow Foreground
# = 34 - Blue Foreground
# = 35 - Magenta Foreground
# = 36 - Cyan Foreground
# = 37 - White Foreground
# = 40 - Black Background
# = 41 - Red Background
# = 42 - Green Background
# = 43 - Yellow Background
# = 44 - Blue Background
# = 45 - Magenta Background
# = 46 - Cyan Background
# = 47 - White Background

Esc[2] Clear the Display
Esc[K Erases from the Current Cursor Position to End of the Line

Esc[6n Device Status Report - request the current position to be returned in the "Standard Input" Device

Esc[#1;R This is the Current Cursor Row ("#") and Column ("1") loaded into the "Standard Input" after a "Device Status Report"

Esc[#1;f Move Cursor to Row "#" and Column "1"

Esc[#1;P Move Cursor to Row "#" and Column "1"

Esc[#1;R Move Cursor to Row "#" and Column "1"

Esa[#A Move the Cursor Up # Rows

Esa[#B Move the Cursor Down # Rows

Esa[#C Move the Cursor to the Right by # Columns
612 Chapter 16

IBM PC extended ASCII characters

The additional 128 characters shown in Fig. 16.2 can do a lot to enhance a character mode application without having to resort to using graphics. These enhancements include special characters for languages other than English, engineering symbols, and simple graphics characters. These simple graphics characters allow lines, and boxes in applications can be created (Figs. 16.1 and 16.2).

Windows ASCII characters

ASCII control characters do have meaning in Windows applications and do not have corresponding graphics characters for video RAM. The Windows character set starts with the “Blank” (ASCII 0x020) and only has the 232 upper characters defined. This character set is based on ASCII with the upper 128 characters defined for special functions and “National Languages” (Fig. 16.3).

EBCDIC

“Extended Binary-Coded Decimal Interchange Code”. In the Table below, empty spaces do not have any characters. Note that EBCDIC is an 8-bit code.
### Data Tables

<table>
<thead>
<tr>
<th>Hex</th>
<th>0x</th>
<th>1x</th>
<th>2x</th>
<th>3x</th>
<th>4x</th>
<th>5x</th>
<th>6x</th>
<th>7x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>x3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>x4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>x5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>x6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>x7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>x8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>x9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>xA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>K</td>
<td>l</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>xB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>\</td>
<td>l</td>
<td>\</td>
</tr>
<tr>
<td>xC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td>\</td>
<td>n</td>
<td>\</td>
</tr>
<tr>
<td>xE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O</td>
<td>/</td>
<td>o</td>
<td>/</td>
</tr>
</tbody>
</table>

**Figure 16.1** IBM PC “Extended ASCII” Set 0-0x07F
### Chapter 16

**Figure 16.2** IBM PC “Extended ASCII” Set 0x080-0x0FF

<table>
<thead>
<tr>
<th>Hex</th>
<th>0x</th>
<th>9x</th>
<th>Ax</th>
<th>Bx</th>
<th>Cx</th>
<th>Dx</th>
<th>Ex</th>
<th>Fx</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>Ç</td>
<td>Ė</td>
<td>Ė</td>
<td>Ĥ</td>
<td>L</td>
<td>L</td>
<td>α</td>
<td>=</td>
</tr>
<tr>
<td>x1</td>
<td>Ū</td>
<td>Ī</td>
<td>Ī</td>
<td>Ī</td>
<td>ą</td>
<td>ą</td>
<td>ą</td>
<td>ą</td>
</tr>
<tr>
<td>x2</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>x3</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>x4</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>x5</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>x6</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>x7</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>x8</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>x9</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>xA</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>xB</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>xC</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>xD</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>xE</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
<tr>
<td>xF</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
<td>Ė</td>
</tr>
</tbody>
</table>
Data Tables 615

Figure 16.3 Microsoft Windows "Arial" Font

Audio Notes

Notes around Middle "C". Note that an Octave above is twice the note frequency and an Octave below is one-half the note frequency.

<table>
<thead>
<tr>
<th>Note</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>392 Hz</td>
</tr>
</tbody>
</table>
### Touch-Tone Telephone Frequencies

<table>
<thead>
<tr>
<th>Frequency</th>
<th>1209 Hz</th>
<th>1336 Hz</th>
<th>1477 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td>697 Hz</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>770 Hz</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>852 Hz</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>941 Hz</td>
<td>*</td>
<td>0</td>
<td>#</td>
</tr>
</tbody>
</table>

### Modem “AT” Commands

“AT” refers to the command “prefix” that is sent before each command to the modem. All Commands (except for “A/”) must start with the ASCII Characters “AT” and end with an ASCII Carriage Return (0x00D).

Command “A/” will cause the modem to repeat the last command. The command will repeat upon receipt of the “/” character.

Command “+++” will force the modem from “on-line” state to local (“AT Command Set”) state. Do not pass data to the modem for one second before and one second after this command.
<table>
<thead>
<tr>
<th>Command</th>
<th>Operation</th>
<th>Expected Reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>If &quot;AT&quot; Sent without a prefix, then Modem is tested</td>
<td>&quot;OK&quot;</td>
</tr>
<tr>
<td>A</td>
<td>&quot;ATA&quot; forces the modem to take the line &quot;off hook&quot;.</td>
<td>&quot;OK&quot;</td>
</tr>
<tr>
<td></td>
<td>Before executing this command, make sure the string &quot;RING&quot; has been received by the modem</td>
<td></td>
</tr>
<tr>
<td>B#</td>
<td>Set the Communications Preference</td>
<td>&quot;OK&quot;</td>
</tr>
<tr>
<td></td>
<td># = 0, CCITT Mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 1, Bell 103/212A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default - V.21/V.22 (High Speed)</td>
<td></td>
</tr>
<tr>
<td>DP</td>
<td>&quot;ATDP $###&quot; Dial the Specified Number using &quot;Pulse Dialing&quot;.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;,&quot; in digit string causes a delay.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;#&quot; in digit string causes the modem to wait for a dial tone before continuing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;;&quot; in digit string causes a wait for calling card tone.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;;&quot; in digit string causes a wait for quiet period</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;;&quot; at the end of the digit string causes the modem to go on hook and off hook momentarily.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;K&quot; at the end of the digit string causes the modem to go on hook and into &quot;auto answer&quot; mode after dialing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;;&quot; at the end of the digit string causes the modem to go into the local command state after connecting.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;$#$&quot; dials the number stored in location &quot;#$&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NO DIALTONE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NO ANSWER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NO CARRIER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BUSY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 300</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 1200</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 1200/REL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 2400</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 2400/REL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 4800</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 4800/REL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 7200</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 7200/REL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 9600</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 9600/REL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 12000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 12000/REL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 14400</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNECT 14400/REL</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Operation</td>
<td>Expected Reply</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>DT</td>
<td>&quot;ATDT #######&quot; Dial the Specified Number using &quot;Tone Dialing&quot;. &quot;.&quot; in digit string causes a delay. &quot;#&quot; in digit string causes the modem to wait for a dial tone before continuing. &quot;;&quot; in digit string causes a wait for calling card tone. &quot;@&quot; in digit string causes a wait for quiet period. &quot;!&quot; in digit string causes the modem to go on hook and off hook momentarily. &quot;R&quot; at the end of the digit string causes the modem to go on hook and into &quot;auto answer&quot; mode after dialing. &quot;S&quot; at the end of the digit string causes the modem to go into the local command state after connecting. &quot;S#&quot; dials the number stored in location &quot;#&quot;</td>
<td>NO DIALTONE NO ANSWER NO CARRIER BUSY CONNECT 300 CONNECT 300/REL CONNECT 4800 CONNECT 4800/REL CONNECT 7200 CONNECT 7200/REL CONNECT 9600 CONNECT 9600/REL CONNECT 12000 CONNECT 12000/REL CONNECT 14400 CONNECT 14400/REL</td>
</tr>
</tbody>
</table>
| E#      | Turn on or off the AT Command "Echo" State. 

# = 0, Turn off "Echo" Mode 

# = 1, Turn on "Echo" Mode (Default) | OK |
| N[#]    | First Enter "++" Command and then send "ATH#". 

# = 0, Put modem on hook 

# = 1, Put modem off hook | OK |
| I# | Request Modem Information | # = 0, Product ID  
|    |                          | # = 1, Modem Code  
|    |                          | # = 2, "OK"  
|    |                          | # = 3, Country Code  
|    |                          | # = 4, Return Features  
| L# | Speaker Code (0 Soft, 9 Loud) | OK  
| M# | Control Speaker          | OK  
|    | # = 0, Turn off Speaker   |  
|    | # = 1, Turn on Speaker until Carrier Established (Default) |  
|    | # = 2, Leave Speaker on Continuously |  
|    | # = 3, Speaker on except when dialing |  
| N# | Specify Communication Preference | OK  
|    | # = 0, Use S37 for Speed Selection. If S37 = 0, then connect at Highest Speed Possible |  
|    | # = 1, Connect at Speed Set in S37 |  
| O# | Return to on line state  | OK  
|    | # = 0, Return to on line state after using "+++" Command Sequence |  
|    | # = 1, Return to on line state after Carrier "retrain" |  
| P  | Enable Pulse Dialing     |  
| Q# | Specify modem reply returned. See "V6" | OK |  
|    | # = 0, Send Result Codes (Default) |  
|    | # = 1, Turn off Messages |  
|    | # = 2, Send Result Codes when Originating call |  

Note: The table provides instructions for various commands related to modem and speaker control. Each command has associated parameters and actions to be performed.
<table>
<thead>
<tr>
<th>Command</th>
<th>Operation</th>
<th>Expected Reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>S#?</td>
<td>Return the Contents of the Register &quot;#&quot;</td>
<td>Register Contents</td>
</tr>
<tr>
<td>S#=Constant</td>
<td>Set the Register &quot;#&quot; to &quot;Constant&quot;</td>
<td>OK</td>
</tr>
<tr>
<td>T</td>
<td>Enable Tone Dialing</td>
<td>OK</td>
</tr>
<tr>
<td>V#</td>
<td>Verbalize Commands. See &quot;Q#&quot;</td>
<td>OK</td>
</tr>
<tr>
<td>M#</td>
<td>Process Result Codes</td>
<td>OK</td>
</tr>
<tr>
<td>X#</td>
<td>Output Active Result Codes</td>
<td>OK</td>
</tr>
<tr>
<td>Y#</td>
<td>Indicate &quot;Break&quot; interval</td>
<td>OK</td>
</tr>
<tr>
<td>Z#</td>
<td>Modem Reset</td>
<td>OK</td>
</tr>
</tbody>
</table>

# = 0, Displays Response Numbers
# = 1, Displays Response Reply (Default)
# = 0, Do not display "Carrier" Information (Default)
# = 1, Display "Carrier" Information
# = 0, Return only Error and "CONNECT" Replies
# = 1, Return only Error and Initial "CONNECT" Replies
# = 2, Return only Error and Initial "CONNECT" Replies
# = 3, Return all Error and Initial "CONNECT" Replies
# = 4, Return all Replies (Default)
# = 0, Breaks are Ignored (Default)
# = 1, Hang up when Break Received
# = 2, Return to Command State but do NOT hang up when Break Received
# = n, Load Profile n
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;F</td>
<td>Recall Default Profile</td>
<td>OK</td>
</tr>
<tr>
<td>&amp;G#</td>
<td>Specify Guard Tone Transmission</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td># = 0, No Guard Tone (Default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 1, Output 1.8 KHz Guard Tone</td>
<td></td>
</tr>
<tr>
<td>&amp;L#</td>
<td>Specify Leased Line for Signal Lock</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td># = 0, Dial up line (Default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 1, Conditioned Leased Line</td>
<td></td>
</tr>
<tr>
<td>&amp;Q#</td>
<td>Select Connection Mode</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td># = 0, Asynchronous, No Error Control</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 5, Fastest Connection Possible made, Fallback if Problems</td>
<td></td>
</tr>
<tr>
<td>&amp;V</td>
<td>Display Current and Saved Profiles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Profile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saved Profile 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saved Profile 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saved Telephone Numbers</td>
<td></td>
</tr>
<tr>
<td>&amp;W#</td>
<td>Save Current Profile</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td># = 0, Save Current in Profile 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 1, Save Current in Profile 1</td>
<td></td>
</tr>
<tr>
<td>&amp;Y#</td>
<td>Specify Start up Profile &quot;N&quot;</td>
<td>OK</td>
</tr>
<tr>
<td>&amp;Z#=###...</td>
<td>Save Specified Telephone Number. Note</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>Digit String can have the parameters Listed in &quot;DP&quot; and &quot;DT&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 16

Modem registers

All registers are 8 bits in size and take the range 0x000 to 0x0FF unless otherwise noted. Registers handle numeric data as decimal rather than Hex. Below are Hex Values shown for compatibility with this chapter.
<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Number of Rings before Auto-Answer</td>
<td>0</td>
</tr>
<tr>
<td>S1</td>
<td>Ring Counter</td>
<td>N/A</td>
</tr>
<tr>
<td>S2 (&quot;+&quot;&quot;)</td>
<td>Escape Character (7 Bit ASCII)</td>
<td>0x02B</td>
</tr>
<tr>
<td>S3</td>
<td>Line End Character</td>
<td>0x00D (CR)</td>
</tr>
<tr>
<td>S4</td>
<td>Line Feed Character</td>
<td>0x00A (LF)</td>
</tr>
<tr>
<td>S5</td>
<td>Backspace Character</td>
<td>0x008 (BS)</td>
</tr>
<tr>
<td>S6</td>
<td>Initial Dialing Wait (in Seconds)</td>
<td>2</td>
</tr>
<tr>
<td>S7</td>
<td>Carrier Wait (Seconds)</td>
<td>50</td>
</tr>
<tr>
<td>S8</td>
<td>Pause Time (Seconds)</td>
<td>2</td>
</tr>
<tr>
<td>S9</td>
<td>Carrier Detect Response Time (1/10 Seconds)</td>
<td>6</td>
</tr>
<tr>
<td>S10</td>
<td>Disconnect Time (1/10 Seconds)</td>
<td>14</td>
</tr>
<tr>
<td>S11</td>
<td>Tone Dialing Spacing (msecs)</td>
<td>95</td>
</tr>
<tr>
<td>S12</td>
<td>Escape Code Guard Time (1/50 Seconds)</td>
<td>50</td>
</tr>
<tr>
<td>S18</td>
<td>Self Test Duration (Seconds)</td>
<td>0</td>
</tr>
<tr>
<td>S36</td>
<td>Negotiation Failure Response (Settings)</td>
<td>5</td>
</tr>
<tr>
<td># = 0</td>
<td>Attempt V.42</td>
<td></td>
</tr>
<tr>
<td># = 1</td>
<td>Attempt V.42/Attempt MNP</td>
<td></td>
</tr>
<tr>
<td># = 4</td>
<td>Attempt V.42/Attempt MNP</td>
<td></td>
</tr>
<tr>
<td># = 5</td>
<td>Attempt V.42/Attempt MNP/Attempt Asynchronous Connection</td>
<td></td>
</tr>
<tr>
<td>S37</td>
<td>Desired Connection Speed</td>
<td>0</td>
</tr>
<tr>
<td># = 0</td>
<td>Connect at Highest Possible Speed (Default)</td>
<td></td>
</tr>
<tr>
<td># = 3</td>
<td>300 bps</td>
<td></td>
</tr>
<tr>
<td># = 5</td>
<td>1200 bps</td>
<td></td>
</tr>
<tr>
<td># = 6</td>
<td>2400 bps</td>
<td></td>
</tr>
<tr>
<td># = 7</td>
<td>4800 bps</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># = 8, 7200 bps</td>
<td></td>
<td></td>
</tr>
<tr>
<td># = 9, 9600 bps</td>
<td></td>
<td></td>
</tr>
<tr>
<td># = 10, 12000 bps</td>
<td></td>
<td></td>
</tr>
<tr>
<td># = 11, 14400 bps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S38</td>
<td>Delay before hang up (Seconds)</td>
<td>0S46</td>
</tr>
<tr>
<td></td>
<td>V.42 bis Data Compression Settings</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td># = 136, V.42 only</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 138, V.42 with V.42 bis compression (Default)</td>
<td></td>
</tr>
<tr>
<td>S48</td>
<td>Feature Negotiation</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td># = 0, Negotiation Disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 3, Negotiation without Detection Phase</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 7, Negotiation with Detection Phase (Default)</td>
<td></td>
</tr>
<tr>
<td>S95</td>
<td>Error Control Negotiation Messaging</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td># = 1, Not Used</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 4, Enables Carrier Messages Only</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 8, Enables Carrier and Protocol Messages Only</td>
<td></td>
</tr>
<tr>
<td></td>
<td># = 32, Enables Carrier, Protocol and Compression Messages (Default)</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>.-</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-...</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>-.-.</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>-..</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>..-.</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>--.</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>..</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>.---</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>-.</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>.-.</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>.--</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>--.-</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>.-.</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>..-</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>...-</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>.-</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>-..-</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>-.--</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>--..</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.----</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>..---</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...--</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>....-</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>-....</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>--...</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>---..</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>----.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>Period</td>
<td>.-.-.-</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>--..-</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>---..</td>
<td></td>
</tr>
<tr>
<td>Dash</td>
<td>-----</td>
<td></td>
</tr>
</tbody>
</table>
626    Chapter 16

/  ------
?  ------
Error  --------
End Trans  ----
Inv Trans  ---

Phonetic Alphabets
Letter  Engineering  Aviation
A      Able          Alpha
B      Baker         Bravo
C      Charlie       Charlie
D      Dog           Delta
E      Easy          Echo
F      Fox           Foxtrot
G      George        Gulf
H      Harry         Hotel
I      Izzy          India
J      Joe           Juliet
K      Kitten        Kilo
L      Larry         Lima
M      Mike          Mike
N      Nancy         November
O      Oscar         Oscar
P      Peter         Papa
Q      Quincy        Quebec
R      Robert        Romeo
S      Sam           Sierra
T      Tom           Tango
U      Under         Uniform
V      Vic           Victor
W      Walter        Whiskey
X      X-Ray         X-Ray
Y      Young         Yankee
Z      Zebra         Zulu

“Ten” Radio Codes
Code  Message
10-1  Receiving Poorly, Bad Signal
10-2  Receiving OK, Strong Signal
10-3  Stop Transmitting
10-4  Message Received
10-5 Relay Message
10-6 Busy, Please Stand By
10-7 Out of Service
10-8 In Service
10-9 Repeat Message
10-10 Finished, Standing By
10-11 Talk Slower
10-12 Visitors Present
10-13 Need Weather/Road Conditions
10-16 Pickup Needed at ________
10-17 Urgent Business
10-18 Is there anything for us
10-19 Nothing for you, Return to Base
10-20 My Location is ________
10-21 Use a Telephone
10-22 Report in Person to ________
10-23 Stand By
10-24 Finished Last Assignment
10-25 Contact
10-26 Disregard Last Information
10-27 I'm Changing to Channel ________
10-28 Identify your Station
10-29 You Time is up for Contact
10-30 Doesn't Conform to FCC Rules
10-32 I'll Give you a Radio Check
10-33 Emergency Traffic at this Station
10-34 Help Needed at this Station
10-35 Confidential Information
10-36 The Correct Time is ________
10-37 Wrecker needed at ________
10-38 Ambulance needed at ________
10-39 Your Message has been Delivered
10-41 Please Change to Channel ________
10-42 Traffic Accident at ________
10-43 Traffic Congestion at ________
10-44 I have a Message for ________
10-45 All Units Within Range Please Report In
10-50 Break Channel
10-60 What is the Next Message Number
10-62 Unable to Copy, Please call on Telephone
10-63 Net Directed to ________
10-64 Net Clear
10-65 Standing By, Awaiting Your Next Message
10-67 All Units Comply
10-70 Fire at ________
628 Chapter 16

10-71 Proceed with Transmission in Sequence
10-73 Speed Trap at
10-75 Your Transmission is Causing Interference
10-77 Negative Contact
10-81 Reserve Hotel Room for ______
10-82 Reserve Room for

10-84 My Telephone Number is ______
10-85 My Address is
10-89 Radio Repairman is Needed at ______
10-90 I have TVI
10-91 Talk Closer to the Microphone
10-92 Your Transmitter Needs Adjustment
10-93 Check my Frequency on this Channel
10-94 Please give me a Long Count
10-95 Transmit Dead Carrier for 5 Seconds
10-99 Mission Completed, All Units Secure
10-200 Police Needed at ______
Resistor Color Coding

Color Coding on resistors is based on the “Bands” around the device (Fig. 17.1).

The Actual Value is determined as:

\[
\text{Resistance} = (\text{First Digit} \times 10 + \text{Second Digit}) \times \text{Multiplier}
\]

<table>
<thead>
<tr>
<th>Number</th>
<th>Color</th>
<th>Band1</th>
<th>Band2</th>
<th>Band3</th>
<th>Band4</th>
<th>Optional Band5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Black</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
<td>10**</td>
<td>0 N/A</td>
</tr>
<tr>
<td>1</td>
<td>Brown</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10**</td>
<td>1% Tolerance</td>
</tr>
<tr>
<td>2</td>
<td>Red</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10**</td>
<td>2% Tolerance</td>
</tr>
</tbody>
</table>
630 Chapter 17

![Resistor Bands Diagram](Image)

**Figure 17.1** Resistor Bands

<table>
<thead>
<tr>
<th>Band Color</th>
<th>1st Digit</th>
<th>2nd Digit</th>
<th>Multiplier</th>
<th>Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>10 ** 3 N/A</td>
</tr>
<tr>
<td>Yellow</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>10 ** 4 N/A</td>
</tr>
<tr>
<td>Green</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10 ** 5 0.5% Tolerance</td>
</tr>
<tr>
<td>Blue</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>10 ** 6 0.25% Tolerance</td>
</tr>
<tr>
<td>Violet</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>10 ** 7 0.1% Tolerance</td>
</tr>
<tr>
<td>Gray</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>10 ** 8 0.05% Tolerance</td>
</tr>
<tr>
<td>White</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>10 ** 9 N/A</td>
</tr>
<tr>
<td>N/A Gold</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10 ** -1 5% Tolerance</td>
</tr>
<tr>
<td>N/A Silver</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10 ** -2 10% Tolerance</td>
</tr>
</tbody>
</table>

**Electromagnetic Spectrum**

<table>
<thead>
<tr>
<th>Frequency Range</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Hz</td>
<td>Direct Current (DC)</td>
</tr>
<tr>
<td>15-20,000 Hz</td>
<td>Audio Frequencies</td>
</tr>
<tr>
<td>30-15,000 Hz</td>
<td>Human Hearing</td>
</tr>
<tr>
<td>16-4186.01 Hz</td>
<td>Musical Scales</td>
</tr>
<tr>
<td>10 KHz - 16 KHz</td>
<td>&quot;Ultrasonics&quot;</td>
</tr>
<tr>
<td>3-30 KHz</td>
<td>Very Low Frequency</td>
</tr>
<tr>
<td>3-30 Hz</td>
<td>Extremely Low Frequency</td>
</tr>
<tr>
<td>30-300 Hz</td>
<td>Radio Transmissions</td>
</tr>
<tr>
<td>30 KHz - 300 KHz</td>
<td>Ultra Low Frequency Radio Transmissions</td>
</tr>
<tr>
<td>53.5 KHz - 170.5 KHz</td>
<td>Radio Frequencies</td>
</tr>
<tr>
<td></td>
<td>AM Broadcast Bands</td>
</tr>
</tbody>
</table>
### Miscellaneous Electronics

<table>
<thead>
<tr>
<th>Frequency Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5 MHz - 4 MHz</td>
<td>80 Meter Amateur Band</td>
</tr>
<tr>
<td>7 MHz - 7.3 MHz</td>
<td>40 Meter Amateur Band</td>
</tr>
<tr>
<td>10.10 MHz - 10.150 MHz</td>
<td>10 Meter Amateur Band</td>
</tr>
<tr>
<td>14.10 MHz - 14.35 MHz</td>
<td>20 Meter Amateur Band</td>
</tr>
<tr>
<td>18.068 MHz - 18.168 MHz</td>
<td>17 Meter Amateur Band</td>
</tr>
<tr>
<td>21.00 MHz - 21.45 MHz</td>
<td>15 Meter Amateur Band</td>
</tr>
<tr>
<td>24.890 MHz - 24.990 MHz</td>
<td>12 Meter Amateur Band</td>
</tr>
<tr>
<td>26.965 MHz - 27.405 MHz</td>
<td>Citizens Band (&quot;CB&quot;)</td>
</tr>
<tr>
<td>26.95 MHz - 27.54 MHz</td>
<td>Industrial, Scientific, Medical Use</td>
</tr>
<tr>
<td>28.00 MHz - 29.70 MHz</td>
<td>10 Meter Amateur Band</td>
</tr>
<tr>
<td>30 MHz - 300 MHz</td>
<td>Very High Frequencies</td>
</tr>
<tr>
<td>30 MHz - 50 MHz</td>
<td>Police, Fire, Forest, Highway, Railroad</td>
</tr>
<tr>
<td>50 MHz - 54 MHz</td>
<td>6 Meter Amateur Band</td>
</tr>
<tr>
<td>54 MHz - 72 MHz</td>
<td>TV Channels 2 to 4</td>
</tr>
<tr>
<td>72 MHz - 76 MHz</td>
<td>Government</td>
</tr>
<tr>
<td>76 MHz - 88 MHz</td>
<td>TV Channels 5 and 6</td>
</tr>
<tr>
<td>88 MHz - 108 MHz</td>
<td>FM Broadcast Band</td>
</tr>
<tr>
<td>108 MHz - 118 MHz</td>
<td>Aeronautical Navigation</td>
</tr>
<tr>
<td>118 MHz - 136 MHz</td>
<td>Civil Communication Band</td>
</tr>
<tr>
<td>144 MHz - 148 MHz</td>
<td>Government</td>
</tr>
<tr>
<td>144 MHz - 148 MHz</td>
<td>2 Meter Amateur Band</td>
</tr>
<tr>
<td>174 MHz - 216 MHz</td>
<td>TV Channels 7 through 13</td>
</tr>
<tr>
<td>216 MHz - 470 MHz</td>
<td>Miscellaneous Communication</td>
</tr>
<tr>
<td>470 MHz - 806 MHz</td>
<td>TV Channels 14 through 69</td>
</tr>
<tr>
<td>806 MHz - 890 MHz</td>
<td>Cellular Telephone</td>
</tr>
<tr>
<td>890 MHz - 3000 MHz</td>
<td>Miscellaneous Communication</td>
</tr>
<tr>
<td>3 GHz - 30 GHz</td>
<td>Miscellaneous Communication/Radar</td>
</tr>
<tr>
<td>30 GHz - 300 GHz</td>
<td>Super High Frequencies/Radar</td>
</tr>
</tbody>
</table>

**Wavelength**
- **30 um - 0.76 um**: Infrared Light and Heat
632 Chapter 17

0.76 um - 0.39 um Visible Light
6470 - 7000 Angstroms Red Light
5850 - 6470 Angstroms Orange Light
5750 - 5850 Angstroms Yellow Light
5560 - 5750 Angstroms Maximum Visibility Light
4912 - 5560 Angstroms Green Light
4240 - 4912 Angstroms Blue Light
4000 - 4240 Angstroms Violet Light
320 - 4000 Angstroms Ultraviolet Light
0.032 - 0.00001 um X-Rays
0.00001 - 0.0000006 um Gamma Rays
< 0.0005 Angstroms Cosmic Rays

Radar bands

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Band</th>
</tr>
</thead>
<tbody>
<tr>
<td>390 - 1,550 MHz</td>
<td>L</td>
</tr>
<tr>
<td>1,550 - 5,200 MHz</td>
<td>S</td>
</tr>
<tr>
<td>5,200 - 10,900 MHz</td>
<td>X</td>
</tr>
<tr>
<td>10,900 - 36,000 MHz</td>
<td>K</td>
</tr>
<tr>
<td>36,000 - 46,000 MHz</td>
<td>Q</td>
</tr>
<tr>
<td>46,000 - 56,000 MHz</td>
<td>V</td>
</tr>
<tr>
<td>56,000 - 100,000 MHz</td>
<td>W</td>
</tr>
</tbody>
</table>

Digital Logic

The Output/Threshold Levels for +5V Logic is:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Input Threshold</th>
<th>Output &quot;Low&quot;</th>
<th>Output &quot;High&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTL</td>
<td>1.4 Volts</td>
<td>0.3 Volts</td>
<td>3.3 Volts</td>
</tr>
<tr>
<td>HC</td>
<td>2.4 Volts</td>
<td>0.1 Volts</td>
<td>4.9 Volts</td>
</tr>
<tr>
<td>HC-T</td>
<td>1.4 Volts</td>
<td>0.1 Volts</td>
<td>4.9 Volts</td>
</tr>
<tr>
<td>CMOS</td>
<td>2.5 Volts</td>
<td>0.1 Volts</td>
<td>4.9 Volts</td>
</tr>
</tbody>
</table>

Gates

The six most common Logic Gates are:
### Type    Symbol

<table>
<thead>
<tr>
<th>NOT</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AND</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot; &quot;B&quot;</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot; &quot;B&quot;</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XOR</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot; &quot;B&quot;</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAND</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot; &quot;B&quot;</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOR</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot; &quot;B&quot;</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Miscellaneous Electronics 633

#### Flip flops

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</th>
<th>Operation</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>R Q</td>
<td>Q_0 Q_0</td>
<td>Reset</td>
</tr>
<tr>
<td></td>
<td>S Q</td>
<td>0 0</td>
<td>Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Saved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1</td>
<td>Illegal</td>
</tr>
</tbody>
</table>
### 634 Chapter 17

#### JK

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q</th>
<th>_Q</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Set</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Qo</td>
<td>Saved</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>_Qo</td>
<td>Togle</td>
</tr>
</tbody>
</table>

#### T

<table>
<thead>
<tr>
<th>Q</th>
<th>_Q</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qo</td>
<td>_Qo</td>
<td>Saved</td>
</tr>
<tr>
<td>_Qo</td>
<td>Qo</td>
<td>Toggle</td>
</tr>
</tbody>
</table>

#### D

<table>
<thead>
<tr>
<th>D</th>
<th>Clk</th>
<th>Q</th>
<th>_Q</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>Dwn</td>
<td>0</td>
<td>1</td>
<td>Latch &quot;0&quot;</td>
</tr>
<tr>
<td>Up</td>
<td>Dwn</td>
<td>1</td>
<td>0</td>
<td>Latch &quot;1&quot;</td>
</tr>
<tr>
<td>0</td>
<td>Qo</td>
<td>_Qo</td>
<td>Saved</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Qo</td>
<td>_Qo</td>
<td>Saved</td>
<td></td>
</tr>
</tbody>
</table>
DC Electronics Formulas

Ohm’s Law:

\[ V = IR \]

Power:

\[ P = VI \]

Series Resistance:

\[ R_t = R_1 + R_2 \ldots \]
636 Chapter 18

Parallel Resistance:
\[ R_t = \frac{1}{\left( \frac{1}{R_1} + \frac{1}{R_2} \right)} \]

Two Resistors in Parallel:
\[ R_t = \frac{R_1 \times R_2}{R_1 + R_2} \]

Series Capacitance:
\[ C_t = \frac{1}{\left( \frac{1}{C_1} + \frac{1}{C_2} \right)} \]

Parallel Capacitance:
\[ C_t = C_1 + C_2 \]

Wheatstone Bridge:

\[ \frac{R_u}{R} = \frac{R_1 \times R_3}{R_2} \]
When NoCurrent Flow in the Meter

AC Electronics Formulas

Resonance:
\[ \text{frequency} = \frac{1}{\left( 2 \times \pi \times \sqrt{L \times C} \right)} \]

RC Time Constant:
\[ \tau = R \times C \]

RL Time Constant:
\[ \tau = \frac{L}{R} \]
RC Charging:
\[ V(t) = V_f \times (1 - e^{-t/T}) \]
\[ i(t) = i_f \times (1 - e^{-t/T}) \]

RC Discharging:
\[ V(t) = V_i \times e^{-t/T} \]
\[ i(t) = i_i \times e^{-t/T} \]

Coil Inductance Formulas:
- Coil around Linear Form:
  \[ \text{Inductance} = \text{Permeability of Form} \times \left( \frac{\text{Number of Turns}^2 \times \text{Form Area}}{\text{Coil Length}} \right) \]
- Coil Around Toroid with a square cross-section:
  \[ \text{Inductance} = \ln\left( \frac{\text{Outer Diameter}}{\text{Inner Diameter}} \right) \times \text{Permeability of Form} \times \left( \frac{\text{Number of Turns}^2 \times \text{Thickness of Toroid}}{2 \pi} \right) \]

Transformer Current/Voltage:
- \[ \text{Turns Ratio} = \frac{\text{Number of Turns on Primary ("p") Side}}{\text{Number of Turns on Secondary ("s") Side}} \]
- \[ \text{Turns Ratio} = \frac{V_s}{V_p} = \frac{I_p}{I_s} \]

Transmission Line Characteristic Impedance:
\[ Z_o = \sqrt{\frac{L}{C}} \]
638  Chapter 18

**Mathematical Formulas**

Frequency = Speed / Wavelength

For Electromagnetic Waves:

Frequency = c / Wavelength

Perfect Gas Law:

\[ PV = nRT \]

**Boolean Arithmetic**

Identify Functions:

\[ A \text{ AND } 1 = A \]
\[ A \text{ OR } 0 = A \]

Output Set/Reset:

\[ A \text{ AND } 0 = 0 \]
\[ A \text{ OR } 1 = 1 \]

Identity Law:

\[ A = A \]

Double Negation Law:

\[ \text{NOT}(\text{NOT}(A)) = A \]

Complementary Law:

\[ A \text{ AND } \text{NOT}(A) = 0 \]
Formulas 639

A OR NOT( A ) = 1

Idempotent Law:

A AND A = A
A OR A = A

Commutative Law:

A AND B = B AND A
A OR B = B OR A

Associative Law:

(A AND B) AND C = A AND (B AND C) = A AND B AND C
(A OR B) OR C = A OR (B OR C) = A OR B OR C

Distributive Law:

A AND (B OR C) = (A AND B) OR (A AND C)
A OR (B AND C) = (A OR B) AND (A OR C)

De Morgan’s Theorem:

NOT( A OR B ) = NOT( A ) AND NOT( B )
NOT( A AND B ) = NOT( A ) OR NOT( B )

Note:

AND is often represented as multiplication, nothing between terms, "." or "*".
OR is often represented as addition with "+" between terms.
Conversions

1 Inch = 2.54 Centimeters
1 Mile = 1.609 Kilometers
1 Ounce = 29.57 Grams
1 Gallon = 3.78 Liters
1 Atmosphere = 29.9213 Inches of Mercury
   = 14.6960 Pounds per Square Inch
   = 101.325 kiloPascals

10,000,000,000 Angstroms = 1 Meter
1,000,000 Microns = 1 Meter
Tera = 1,000 Giga
Giga = 1,000 Mega
Mega = 1,000 Kilo
Kilo = 1,000 Units
Unit = 100 Centi
Unit = 1,000 Milli

1 Hour = 3,600 Seconds
1 Year = 8,760 Hours
Microchip

Microchip's corporate headquarters is

Microchip Technology, Inc.
2355 W. Chandler Blvd.
Chandler, AZ 85224
Phone: (480) 786-7200
Fax: (480) 917-4150

Their Web site ("Planet Microchip") is at
http://www.microchip.com and contains a complete

Copyright 2001 The McGraw-Hill Companies. Click Here for Terms of Use.
set of data sheets in .pdf format for download as well as the latest versions of MPLAB. Also on the website is the link to http://buy.microchip.com, which is Microchip's on-line ordering system for parts and development tools.

Microchip puts on a series of seminars throughout the world every year. Information on these events can be found on the microchip Web page.

**PICmicro® MCU Books**

Note that Microchip has excellent datasheets available for download from their Web site as well as available on CD-ROM.

*Programming and Customizing the PICmicro® MCU Eight Bit Microcontroller—Second Edition*

Author: M. Predko
ISBN: 0-07-136175-8

*Design with PIC Microcontrollers*

Author: J. B. Peatman
ISBN: 0-13-759259-0

*PICTUTOR*

Author: J. Becker
URL:

http://www.matrixmultimedia.co.uk/picprods.htm

*PIC'n Techniques*

Author: D. Benson
ISBN: 0-9654162-3-2
Resources

PIC'n Up the Pace
Author: D. Benson
ISBN: 0-9654162-1-6

Serial PIC'n
Author: D. Benson
ISBN: 0-9654162-2-4

Easy PIC'n
Author: D. Benson
ISBN: 0-9654162-0-8

Author: L. Duarte
ISBN: 0-79061-153-8

An Introduction to PIC Microcontrollers
Author: R. A. Penfold
ISBN: 0-85934-394-4

Practical PIC Microcontroller Projects
Author: R. A. Penfold
ISBN: 0-85934-444-4

A Beginners Guide to the Microchip PIC—2nd Edition
Author: N. Gardner
ISBN: 1-899013-01-6

PIC Cookbook
Author: N. Gardner
ISBN: 1-899013-02-4

Useful Books
Here are a collection of books that are useful for developing electronics and software for applications. Some of
these are hard to find, but definitely worth the effort in finding them in a used bookstore.

*The Art of Electronics*–1989

Horowitz and Hill’s definitive book on electronics—a complete electrical engineering course wrapped up in 1125 pages. Some people may find it to be a bit too complex, but just about any analog electronics question you could have will be answered in this book. The digital information in this book is less complete.

ISBN: 0-521-37095-7

*Bebop to the Boolean Boogie*–1995

Somewhat deeper in digital electronics (and less serious) than *The Art of Electronics*, Clive Maxwell’s introduction to electronics stands out with clear and insightful explanations of how things work and why things are done the way they are. It distinguishes itself from other books by explaining Printed Wiring Assembly technology (PCB Boards, Components, and Soldering). This book complements *The Art of Electronics* very nicely.


*The Encyclopedia of Electronic Circuits*–Volume 1 to 7

Rudolf Graf’s Encyclopedia series of Electronic Circuits is an excellent resource of circuits and ideas that have been cataloged according to circuit type. Each book contains thousands of circuits and can really make your life easier when you are trying to figure out how to do something. Each volume contains an index listing circuits for the current volume and the previous ones.
CMOS Cookbook–Revised 1988

In CMOS Cookbook, Don Lancaster introduces the reader to basic digital electronic theory, while also explaining the operation of CMOS gates, providing hints on soldering and prototyping, listing common CMOS parts (along with TTL pinout equivalents) and providing a number of example circuits (including a good basic definition of how NTSC video works). The update by Howard Berlin has made sure the chips presented in the book are still available. In the 1970s, Don Lancaster also wrote the TTL Cookbook, 555 Timer Cookbook, and Active Filter Cookbook.

ISBN: 0-7506-9943-4

The TTL Data Book for Design Engineers–Texas Instruments

I have a couple of 1981 printed copies of the second edition of this book and they are all falling apart from overuse. The Texas Instruments TTL data books have been used for years by hundreds of thousands of engineers to develop their circuits. Each datasheet is complete with pinouts, operating characteristics, and internal circuit diagrams. While the data books are not complete for the latest “HC” parts, they will give you just about everything you want to know about the operation of small scale digital logic.

ISBN: N/A
This book/CD-ROM package was written to give a clear introduction to the PC, from a “bottoms up” hardware perspective as well as an explanation of how code works in the PC. Along with explaining the architecture, there are also over twenty applications that will help the reader understand exactly how MS-DOS and Windows code executes in the PC and how hardware is accessed using the various interfaces available within the PC.


This book is designed as an easy to use pocket reference for programmers and engineers working on the PC. Along with detailing the PC’s architecture, the Intel 8086 and later microprocessors are described. The instruction sets used in the processor are listed along with addressing and value information. The information is useful for all PCs from the first 8088s to the most modern multi-Pentium III systems.

ISBN: 0-07-135525-1

Thom Hogan’s 850 page book is just about the best and most complete reference that you can find anywhere on the PC. This book basically ends at the 386 (no 486, Pentiums of any flavor, PCI, Northbridge, Southbridge or SuperIO, or any ASICs of any type), but is the most complete PC reference that explains BIOS, all the “Standard” I/O, DOS and Windows 3.x Interfaces you can find.

Ed Nisley’s book is an almost complete opposite to the previous two books and *The Programmer’s PC Source Book*. Where the others’ books focus is on documenting the innards of the PC, Nisley’s shows you how to practically interface to the PC’s “Industry Standard Architecture” (“ISA”) bus and if you follow through the book you will end up with an LCD graphic display. Theory, register addresses, and programming information is available in this book, but it is presented as Ed works through the projects. This book is a resource that you can go back to and look at actual scope photographs of bus accesses or discussions on how to decode bus signals. There are a lot of great tricks in this book that can be used for many different PC interfacing applications.


*Handbook of Microcontrollers*–1998

Introduction and complete reference package for modern 8-bit embedded microcontrollers. As well as providing technical and processor programming information on the: Intel 8051, Motorola 68HC05, Microchip PICmicro® MCU, Atmel AVR and Parallax Basic Stamp, datasheets, development tools and sample applications are included on the included CD-ROM. To help with your future applications, interfacing to RS-232, I2C, LCD and other devices is explored and a fair amount of space is devoted to such advanced topics as Fuzzy Logic, Compilers, Real Time Operating Systems (I have included a sample one for the 68HC05), and Network Communications.

ISBN: 0-07-913716-4
648 Chapter 19


This is an excellent introduction to assembly language programming with a fairly low level approach concentrating on Microsoft’s “MASM” and Borland’s “TASM”. “Debug.com” is used exclusively as a debug tool, which makes this book reasonably inexpensive to get involved with.


Brian W. Kernighan, Dennis M. Ritchie’s classic text explaining the “C” programming language has not lost any of its usefulness since its first introduction. This book has probably been used by more students in more colleges and universities than any other. Despite the fact that the book was written originally for a programming course, the writing is crisp, sharp, and easily understandable.

ISBN: 0-13110-362-8

**PICList Internet List Server**

These guidelines should be used and followed for any list server or news group. After the guidelines, there are instructions for subscribing to the PICList.

1. Don’t subscribe to a list and then immediately start sending questions to the list. Instead, wait a day or so to get the hang of how messages are sent and replied to on the list and get a “feel” for the best way of asking questions.
2. Some lists send an email sent to them back to the author (while others do not). If you receive a copy of your first email, don’t automatically assume that it is a “bounce” (wrong address) and resend it. In this case, you might want to wait a day or so to see if any replies show up before trying to resend it. Once you have been on the list for a while, you should get an idea of how long it takes to show up on the list and how long it takes to get a reply.

3. If you don’t get a reply to a request, don’t get angry or frustrated and send off a reply demanding help. There is a good chance that nobody on the list knows exactly how to solve your problem. In this case, try to break down the problem and ask the question a different way.

4. Do not count on getting replies to questions within minutes. Nobody on the PICList is paid to reply to your questions. The majority of people who reply are doing so to help others. Please respect that and do not badger, and help out in anyway that you can.

5. If you are changing the “Subject” line of a post, please reference the previous topic (i.e., put in "was: ‘...’"). This will help others keep track of the conversation.

6. When replying to a previous post, try to minimize how much of the previous note is copied in your note and maximize the relevance to your reply. This is not to say that none of the message should be copied or referenced. There is a very fine balance between having too much and too little. The sender of the note you are replying to should be referenced (with their name or ID).
My rule of thumb is, if the original question is less than ten lines, I copy it all. If it is longer, then I cut it down (identifying what was cut out with a "SNIP" Message), leaving just the question and any relevant information as quoted. Most mail programs will mark the quoted text with a ">" character, please use this convention to make it easier for others to follow your reply.

7. If you have an application that doesn't work, please don't copy the entire source code into an email and post it to a list. As soon as I see an email like this I just delete it and go on to the next one (and I suspect that I’m not the only one). Also, some lists may have a message size limit (anything above this limit is thrown out) and you will not receive any kind of confirmation.

If you are going to post source code: keep it short. People on the list are more than happy and willing to answer specific questions, but simply copying the complete source code in the note and asking a question like "Why won't the LCD display anything" really isn't useful for anybody. Instead, try to isolate the failing code and describe what is actually happening along with what you want to happen. If you do this, chances are you will get a helpful answer quickly.

A good thing to remember when asking why something won't work, make sure you discuss the hardware that you are using. If you are asking about support hardware (i.e., a programmer or emulator), make sure you describe your PC (or workstation) setup. If your application isn't working as expected, describe the hardware that you are using and what
you have observed (i.e., if the clock lines are wig-
gling, or the application works normally when you 
put a scope probe on a pin).

8. You may find a totally awesome and appropriate 
Web page and want to share it with the list. Please 
make it easier on the people in the list to cut and 
paste the URL by putting it on a line all by itself in 
the format:

http://www.awesome-pic-page.com

9. If you have a new application, graphic, or whatever, 
that takes up more than 1K which you would like to 
share with everyone on the list, please don’t send it 
as an attachment in a note to the list. Instead, either 
indicate that you have this amazing piece of work 
and tell people that you have it and where to re-
quest it (either to you directly or to a Web server ad-
dress). If a large file is received many list servers 
may automatically delete it (thrown into the “bit 
bucket”) and you may or may not get a message 
telling you what happened.

If you don’t have a Web page of your own or one 
you can access, requesting that somebody put it on 
their Web page or ftp server is acceptable.

10. Many of these List Servers are made available, main-
tained, and/or moderated by a device manufacturer. 
Keep this in mind if you are going to advertise your 
own product and understand what the company’s 
policy is on this before sending out an advertise-
ment.

The PICList is quite tolerant of advertisements of 
relevant products. If you are boarding puppies or
have something equally non-PICmicro® MCU related, find somewhere else to advertise it.

11. Putting job postings or employment requests may be appropriate for a list (like the previous point, check with the list’s maintainer). However, I don’t recommend that the rate of pay or conditions of employment should be included in the note (unless you want to be characterized as cheap, greedy, unreasonable, or exploitive).

12. “Spams” are sent to every list server occasionally. Please do not “reply” to the note even if the message says that to get off the spammer’s mailing list just “reply”. This will send a message to everyone in the list. If you must send a note detailing your disgust, send it to the spam originator (although to their ISP will probably get better results).

   NOTE: There are a number of companies sending out bogus spams to collect the originating addresses of replying messages and sell them to other companies or distributors of addresses on CD-ROM. When receiving a spam, see if it has been sent to you personally or the list before replying—but beware if you are replying to the spam, you may be just sending your e-mail address for some company to resell to real spammers.

13. Following up with the previous message, if you are going to put in pointers to a list server, just put a hyperlink to the list server request email address, NOT TO THE LIST SERVER ITSELF. If you provide the address to the list server, spammers can pull the link from your page and use it as an address to send spams to. By not doing this, you will be minimizing the opportunity for spammers to send notes to the list.
14. By sending off-topic messages, while it is tolerated, you will probably bring lots of abuse upon yourself, especially if you are belligerent about it. An occasional notice about something interesting or a joke is fine as long as it is unusual and not likely to attract a lot of replies.

If you feel it is appropriate to send an off-topic message; some lists request that you put "[OT]" in the subject line, some members of the list use mail filters and this will allow them to ignore the off-topic posts automatically.

Eventually a discussion (this usually happens with off-topic discussions) will get so strung out that there are only two people left arguing with each other. At this point stop the discussion entirely or go ‘private’. You can obtain the other person’s e-mail address from the header of the message—send your message to him or her and not to the entire list. Everyone else on the list would have lost interest a long time ago and probably would like the discussion to just go away (so oblige them).

15. Posts referencing Pirate sites and sources for "cracked" or "hacked" software are not appropriate in any case and may be illegal. If you are not sure if it is okay to post the latest software you've found on the Web, then DON'T until you have checked with the owners of the software and gotten their permission. It would also be a good idea to indicate in your post that you have the owner's permission to distribute cracked software.

A variety of different microcontrollers are used in "Smart Cards" (such as used with Cable and Satellite scrambling) or video game machines and
654  Chapter 19

asking how they work will probably result in abusive replies at worst or having your questions ignored at best. If you have a legitimate reason for asking about smart cards, make sure you state it in your email to the list.

16. When you first subscribe to a list, you will get a reply telling you how to unsubscribe from the list. DON'T LOSE THIS NOTE. In the past in some lists, people having trouble unsubscribing have sent questions to the list asking how and sometimes getting angry when their requests go unheeded. If you are trying to unsubscribe from a list and need help from others on the list, explain what you are trying to do and how you've tried to accomplish it.

17. When working with a list server, do not have automated replies sent. If they are enabled, then all messages sent by the server to you will be replied to back to the list server. This is annoying for other list members and should be avoided.

18. Lastly, please try to be courteous to all on the list. Others may not have your knowledge and experience or they may be sensitive about different issues. There is a very high level of professionalism on all the lists presented below, please help maintain it. Being insulting or rude will only get you the same attitude back and probably will lead to your posts and legitimate questions being ignored in the future by others on the list who don't want to have anything to do with you.

To put this succinctly: "Don't be offensive or easily offended."
To subscribe to the PICList, send an email to

listserv@mitvma.mit.edu

with the message:

subscribe piclist <I>your name</I>

in the body of your email.

Save the confirmation message; this will give you the instructions for signing off the list as well as instructions on how to access more advanced PICList List Server functions.

To sign off the list, send a note to the same address (listserv@mitvma.mit.edu) with the message:

signoff piclist

When signing off the PICList make sure that you are doing it from the ID that you used to sign on to the list.

Once you have subscribed to the PICLIST, you will begin receiving mail from

piclist@mitvma.mit.edu

Emails can be sent to this address directly or can be replied to directly from your mailer. The list archive is available at:

http://www.iversoft.com/piclist/

and it has a searchable summary of the emails that have been sent to the PICList.
Recommended PICmicro® MCU Web Sites

At the time of writing, there is somewhere in the neighborhood of one thousand Web pages devoted to the PICmicro® MCU with different applications, code snippets, code development tools, programmers, and other miscellaneous information on the PICmicro® MCU and other microcontrollers. The following sites are excellent places to start and work through.

The author's Web page has the latest PICmicro® MCU information as well as errata for this book and sample PICmicro® MCU projects.

http://www.myke.com


http://www.geocities.com/SiliconValley/Way/5807/

Bob Blick's Web site. Some interesting PICmicro® MCU projects that are quite a bit different than the run of the mill.

http://www.bobblick.com/

Scott Dattalo's highly optimized PICmicro® MCU math algorithms. The best place to go if you are looking to calculate Trigonometric Sines in a PICmicro® MCU.

http://www.dattalo.com/technical/software/software.html

Along with the very fast PICmicro® MCU routines, Scott has also been working on some GNU General Purpose License Tools designed to run under Linux.
Resources

The tools can be downloaded from:

http://www.dattalo.com/gnupic/gpsim.html
http://www.dattalo.com/gnupic/gpasm.html

Marco Di Leo’s “PIC Corner”. Some interesting applications including information on networking PICmicro® MCUs and using them for cryptography.

http://members.tripod.com/~mdileo/

Dontronics Home Page. Don McKenzie has a wealth of information on the PICmicro® MCU as well as other electronic products. There are lots of useful links to other sites and it is the home of the SimmStick.

http://www.dontronics.com/

Fast Forward Engineering. Andrew Warren’s page of PICmicro® MCU information and highly useful question/answer page.

http://home.netcom.com/~fastfwd/

Steve Lawther’s list of PICmicro® MCU Projects. Interesting PICmicro® MCU (and other microcontroller) projects.

http://ourworld.compuserve.com/homepages/steve_lawther/ucindex.htm

Eric Smith’s PIC Page. Some interesting projects and code examples to work through.

http://www.brouhaha.com/~eric/pic/
658  Chapter 19

Rickard's PIC-Wall. Good site with a design for PICmicro® MCU-based composite video game generator.

http://www.efd.lth.se/~e96rg/pic.html

PicPoint—Lots of good projects to choose from including 5 MB free to anyone that wants to start their own PICmicro® MCU Web page.

http://www.picpoint.com/

MicroTronics—Programmers and Application reviews.

http://www.eedevl.com/index.html

Tony Nixon’s “Pic ‘n Poke” Development Systems Home page to the “Pic ‘n Poke” development system. This system includes an animated simulator that is an excellent tool for learning how data flows and instructions execute in the PICmicro® MCU microcontroller.

http://www.picnpoke.com/

Periodicals

Here are a number of magazines that do give a lot of information and projects on PICmicro® MCUs. Every month, each magazine has a better than 50% chance of presenting a PICmicro® MCU application.

Circuit Cellar Ink

Subscriptions:
P.O. Box 698
Holmes, PA 19043-9613
1(800)269-6301
Web Site: http://www.circellar.com/
BBS: (860)871-1988
Resources 659

Poptronics
Subscriptions:
Subscription Department
P.O. Box 55115
Boulder, CO
1(800)999-7139
Web Site: http://www.gernsback.com

Microcontroller Journal
Web Site: http://www.mcjournal.com/
This is published on the Web.

Nuts & Volts
Subscriptions:
430 Princeland Court
Corona, CA 91719
1(800)-783-4624
Web Site: http://www.nutsvolts.com

Everyday Practical Electronics
Subscriptions:
EPE Subscriptions Dept.
Allen House, East Borough,
Wimborne, Dorset,
BH21 1PF
United Kingdom
+44 (0)1202 881749
Web Site: http://www.epemag.wimborne.co.uk

Useful Web Sites
While none of these are PICmicro® MCU specific, they are a good source of ideas, information, and products that will make your life a bit more interesting and maybe give you some ideas for projects for the PICmicro® MCU.
Seattle Robotics Society
http://www.hhhh.org/srs/
The Seattle Robotics Society has lots of information on interfacing digital devices to such "real world" devices as motors, sensors, and servos. They also do a lot of exciting things in the automation arena. Most of the applications use the Motorola 68HC11.

List Of Stamp Applications (L.O.S.A)
http://www.hth.com/losa.htm
The List of Parallax Basic Stamp Applications will give you an idea of what can be done with the Basic Stamp (and other microcontrollers, such as the PICmicro® MCU). The list contains projects ranging from using a Basic Stamp to giving a cat medication to providing a simple telemetry system for model rockets.

Adobe PDF Viewers
http://www.adobe.com
Adobe .pdf file format is used for virtually all vendor datasheets, including the devices presented in this book (and their datasheets on the CD-ROM).

“PKZip” and “PKUnZip”
http://www.pkware.com
PKWare's "zip" file compression format is a "Standard" for combining and compressing files for transfer.

Hardware FAQs
http://paranoia.com/~filipg/HTML/LINK/LINK_IN.html
A set of FAQs (Frequently Asked Questions) about the PC and other hardware platforms that will come in
useful when interfacing a microcontroller to a Host PC.

http://www.innovatus.com

Innovatus has made available "PICBots", an interesting PICmicro® MCU simulator which allows programs to be written for virtual robots which will fight amongst themselves.

Part Suppliers

The following companies supplied components that are used in this book. I am listing them because they all provide excellent customer service and are able to ship parts anywhere you need them.

Digi-Key

Digi-Key is an excellent source for a wide range of electronic parts. They are reasonably priced and most orders will be delivered the next day. They are real lifesavers when you're on a deadline.

Digi-Key Corporation
701 Brooks Avenue South
P.O. Box 677
Thief River Falls, MN 56701-0677

Phone: 1(800)344-4539 [1(800)DIGI-KEY]
Fax: (218)681-3380

http://www.digi-key.com/

AP Circuits

AP Circuits will build prototype bare boards from your "Gerber" files. Boards are available within three days. I have been a customer of theirs for several years and
they have always produced excellent quality and been helpful in providing direction to learning how to develop my own bare boards. Their Web site contains the “EasyTrax” and “GCPrevue” MS-DOS tools necessary to develop your own Gerber files.

Alberta Printed Circuits Ltd.
#3, 1112-40th Avenue N.E.
Calgary, Alberta T2E 5T8
Phone: (403)250-3406
BBS: (403)291-9342
Email: staff@apcircuits.com
http://www.apcircuits.com/

Wirz Electronics
Wirz Electronics is a full service Microcontroller component and development system supplier. Wirz Electronics is the main distributor for projects contained in this book and will sell assembled and tested kits of the projects. Wirz Electronics also carries the “SimmStick” prototyping systems as well as their own line of motor and robot controllers.

Wirz Electronics
P.O. Box 457
Littleton, MA 01460-0457
Toll Free in the USA & Canada: 1(888)289-9479
[1(888)BUY-WIRZ]
Email: sales@wirz.com
http://www.wirz.com/
Resources 663

Tower Hobbies
Excellent source for Servos and R/C parts useful in homebuilt robots.

Tower Hobbies
P.O. Box 9078
Champaign, IL 61826-9078
Toll Free Ordering in the USA & Canada: 1(800)637-4989
Toll Free Fax in the USA & Canada: 1(800)637-7303
Toll Free Support in the USA & Canada: 1(800)637-6050
Phone: (217)398-3636
Fax: (217)356-6608
Email: orders@towerhobbies.com
http://www.towerhobbies.com/

Jameco
Components, PC Parts/Accessories, and hard to find connectors.

Jameco
1355 Shoreway Road
Belmont, CA 94002-4100
Toll Free in the USA & Canada: 1(800)831-4242
http://www.jameco.com/

JDR
Components, PC Parts/Accessories, and hard to find connectors.
Chapter 19

JDR Microdevices
1850 South 10th St.
San Jose, CA 95112-4108
Toll Free in the USA & Canada: 1(800)538-5000
Toll Free Fax in the USA & Canada: 1(800)538-5005
Phone: (408)494-1400
Email: techsupport@jdr.com
BBS: (408)494-1430
Compuserve: 70007,1561
http://www.jdr.com/JDR

Newark
Components—Including the Dallas Line of Semiconductors (the DS87C520 and DS275 is used for RS-232 Level Conversion in this book).

Toll Free in the USA & Canada: 1(800)463-9275
[1(800)4-NEWARK]
http://www.newark.com/

Marshall Industries
Marshall is a full-service distributor of Philips microcontrollers as well as other parts.

Marshall Industries
9320 Telstar Avenue
El Monte, CA 91731
1(800)833-9910
http://www.marshall.com
Mouser Electronics

Mouser is the distributor for the Seiko S7600A TCP/IP Stack Chips.

Mouser Electronics, Inc.
958 North Main Street
Mansfield, Texas 76063
Sales: (800) 346-6873
Sales: (817) 483-6888
Fax: (817) 483-6899
Email: sales@mouser.com
http://www.mouser.com

Mondo-tronics Robotics Store

Self-proclaimed as "The World's Biggest Collection of Miniature Robots and Supplies" and I have to agree with them. This is a great source for Servos, Tracked Vehicles, and Robot Arms.

Order Desk
Mondo-tronics Inc.
524 San Anselmo Ave #107-13
San Anselmo, CA 94960
Toll Free in the USA & Canada: 1(800)374-5764
Fax: (415)455-9333
http://www.robotstore.com/
This page intentionally left blank.
Index

ABS, BASIC function, 541
ABS, PicBasic, 560
AC electronics formulas, 636–637
AC power control, 373, 373
acceleration due to gravity, 606
Access Bank, 143
accumulator, 136, 143
ADCON0/ADCON1 registers, 166–167, 180–182, 195–197, 252–257
Add, 23, 31, 42–43, 63–66, 106, 123
Add Constant, 41
ADD Parm, 113
ADDB, 115
adddf, 106
addc, 106, 110
addition
16-bit numbers, 485–488
BASIC function, 541
C language, 593, 594
PicBasic, 560

addlw, 31, 41, 63–64, 468
addresses
C language, 590
mid-range device, 154–155
ADIF register, 168
ADRES module, 168
ADRESL/ADRESH registers, 168, 182, 197, 255–256
ALUSTA register, 170
analog to digital conversion (ADC), 252–257, 253
AND, 23, 31, 32, 44, 67, 113, 123, 124
16-bit numbers, bitwise operations, 488
BASIC function, 542

Note: Boldface numbers indicate illustrations.
AND, continued
    bitwise operations, 2, 478–479
    Boolean, 638, 639
    C language, 503, 504, 595
    PicBasic, 560
AND gate, 633, 633
andlw, 23, 31, 44, 67, 113
andwf, 23, 32, 44, 67, 113
ANSI display control sequences, 609–612
application code template, 539
application software (See also BASIC; C language; PICBasic; Visual Basic)
application code template, 539
BASIC language (See also BASIC; PICBasic; Visual Basic), 540–588
C language (See also C language), 588–603
include (inc) files, 536
linking-linked applications, 536–539
standard declaration files, 536
Visual Basic, 578–588,
578
architectures, processor, 123–150
Access Bank, 143
accumulator, 136, 143
arithmetic logic unit (ALU), 123–124, 124
BSR Register, 137, 138, 144, 149
call stack, 130, 134
fast stack, 147, 149–150
File (Variable)
    Registers, 132
File Register Bank, 137
FSR (index) register, 127, 131, 144–145
INDF register, 145
INTCON register, 134–135
interrupt handler skeleton, 135–136, 142, 150
interrupt operation, 134–135, 140–141, 150
interrupt requests, 134–135, 142
jumps, 129, 134
low-end, 125–130
memory space, 142
mid-range, 130–136
multiplier, 142
OPTION register, 136
oscillator calibration
(OSCCAL) register, 126
PCL register, 134, 140
PCLATH register, 134, 140, 147
PCLATU register, 147
PIC17Cxx, 136–142, 137
PIC18Cxx, 142–150, 143
PIE/PIR registers, 134–135
program counter, 128–130, 129, 133–134, 133, 137, 139–140, 140, 143, 147–150, 148
push operations, 145
register access, 126–127, 130–132, 131, 137–138, 144–146, 144
reset address, 137, 143
Reset addresses, 126
Special Function Registers (SFR), 132, 137
STKPTR register, 148–149
WREG, 143, 145
arithmetic logic unit (ALU), 123–124, 124
arrays
C language, 589–590
PicBasic, 553
ASCII, BASIC function, 545
ASCII code
ANSI display control sequences, 609–612
backslash characters, C language, 598
caracter set/table, 606–607
closest characters, 607–609
convert ASCII byte to hex nybble, 471
convert ASCII to uppercase, 475
convert nybble to ASCII, 470–471
IBM PC extended ASCII, 612, 613, 614
Windows ASCII characters, 612, 615
assembler directives, MPLAB, 526–535
assembly language use, PicBasic, 557
assignment operator
BASIC function, 542
C language, 591
associative law, Boolean, 639
astronomical unit (AU), 605
asynchronous serial communications, 164
asynchronous serial I/O software routines, 347–357
AT command, modems, 216–221
atan2, C language, 602
ATN, BASIC function, 541
atomic mass unit, 606
B, 106
backslash characters, 598
BADRAM, MPLAB
BAKSEL/ BANKISEL, MPLAB, 526
BASE, BASIC function, 540
base addresses, serial port, 443–444
BASIC language (See also BASIC; PICBasic; Visual Basic), 540–588
data types, 540
functions, 540–541, 542–548
Microsoft enhancements, 542–548
operators, 541–542
order of operations, 542
PicBasic, 549–577
variables, 540
Basic SSP (BSSP), 240
bc, 68, 106
bcf, 23, 24, 32, 45, 68, 104, 115, 116, 119
BDC, 106
BEEP, BASIC function, 547
binary numbers, 2
BIOS interfaces
ISA bus, 436–440
parallel port, 459–460
bit banging master interface, 12C, 334–338
bits per second (bps), 1
bitwise AND, 2
bitwise operations, 478–479
16-bit numbers, 488
C language, 593, 595
PicBasic, 560
bitwise OR, 2
bitwise XOR, 2
bn, 69
bnc, 69, 106
BNDC, 107
brn, 70
bnov, 70
bnz, 71, 107
Boltzmann entropy constant, 606
books on PICMicro® MCU, 642–648
Boolean arithmetic, 638
boot code, 401–402
bov, 71
bra, 72
Branch, 68, 69, 70, 71, 72, 74, 106, 107
BRANCH, PicBasic, 564
branching, conditional branching, 465–466
BRANCHL, PicBasic, 564
break, C language, 592, 597
brown out reset circuit, 279
bsf, 45, 72, 104, 115, 119
BSR Register, 137, 138, 144, 149, 169, 172, 203
BSSP modules, 161
btfsc, 24, 32, 33, 45, 72, 115, 116, 117, 118, 120, 121, 122
btfss, 45, 73, 115, 116, 117, 118, 119, 120, 121, 122
btg, 45, 73
bubble sort, compare and swap if Y greater than X, 476
buffers, I/O, 161
built-in hardware features, 207–276
analog to digital conversion (ADC), 252–257, 253
capture/compare/PWM (CCP) module, 228–232
CCP modules, 228–232
CCP1CON/CCP2CON registers, 229
CCPR register, 229
comparators, 257–261, 258, 260
configuration registers, 207–208
EEPROM data memory access, 263–269, 265
EPROM program memory access, 269–271
flash program memory access, 271–272
input/output ports, 217–221, 218, 219, 220
INTCON registers, 225
OPTION register, 216–217, 223
oscillators, 208–213, 209, 210, 212
parallel memory, external, 273–276, 274, 275
parallel slave port (PSP), 261–263, 261
PIE/PIR registers, 225, 228
PORTxx registers, 217–221, 218, 219, 220
prescaler, 223–224, 224
PWM mode, 230–232
672  Index

built-in hardware features, continued
  sleep function, 213–215, 215
  Synchronous Serial Protocol (SSP) module, 239–252, 240
  T2CON register, 227
  TMR0 register/counter, 222–223, 222
  TMR1 register, 224–226, 225
  TMR2 register, 227–228, 227
  TRIS registers, 217–221, 218, 219, 220
  USART module, 232–239, 233–236
  Watchdog Timer, 221, 221
button interface, 284–287, 284
  BUTTON, PicBasic, 564
  BUTTON_PAUSE, PicBasic, 555
bytes, 1
  counting 1s in a byte, 476–477
  generating parity for byte, 477
bz, 74, 107

C language, 588–603
  addresses, 590
  arrays, 589–590
  assignment, 591
  backslash characters, 508
  compound assignment operators, C language, 594
  conditional statements, 591–593
  data types, 588
  declarations, 588
  directives, 595–597
  enhancement functions, 603
  expressions, 591
  Function format, 591
  Function prototype, 591
  functions, 591, 598–603
  Mainline application statement, 590
  operator order, C language, 594
  operators, 593–595
  pointers, 590
  reserved words, 597
  string operations, 589, 601–602
  variables, 588
CA2L/CA2H registers, 176–177, 184
CA4L/CA4H registers, 184
  calibration register, 157, 220
<table>
<thead>
<tr>
<th>Call</th>
<th>24, 33, 45, 52, 74, 89, 104, 119, 128–130, 140, 149, 156</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump outside current page</td>
<td>461–462</td>
</tr>
<tr>
<td>Call stack</td>
<td>130, 134</td>
</tr>
<tr>
<td>CALL</td>
<td>PicBasic, 564</td>
</tr>
<tr>
<td>Calloc</td>
<td>C language, 602</td>
</tr>
<tr>
<td>Capacitance</td>
<td>parallel capacitance, 636</td>
</tr>
<tr>
<td></td>
<td>series capacitance, 636</td>
</tr>
<tr>
<td>Capture and compare operations</td>
<td>PWM (CCP) module, 228–232</td>
</tr>
<tr>
<td></td>
<td>TMR1/TMR2 registers, 160</td>
</tr>
<tr>
<td>Carry flag, rotating byte in place</td>
<td>469</td>
</tr>
<tr>
<td>CASE</td>
<td>BASIC function, 544</td>
</tr>
<tr>
<td>Case</td>
<td>C language, 597</td>
</tr>
<tr>
<td>CBLOCK</td>
<td>MPLAB, 526</td>
</tr>
<tr>
<td>CCP register</td>
<td>159, 160–161, 228–232</td>
</tr>
<tr>
<td>CCP1CON/CCP2CON registers</td>
<td>193–195, 229</td>
</tr>
<tr>
<td>CCPRx registers</td>
<td>194, 195, 229</td>
</tr>
<tr>
<td>Ceramic resonator oscillators</td>
<td>208, 209, 211</td>
</tr>
<tr>
<td>CHARRACING, PicBasic</td>
<td>555</td>
</tr>
<tr>
<td>Characteristic impedance</td>
<td>transmission line, 637</td>
</tr>
<tr>
<td>Charging/discharging, RC</td>
<td>637</td>
</tr>
<tr>
<td>Check Register</td>
<td>27, 35</td>
</tr>
<tr>
<td>CHRS, BASIC function</td>
<td>545</td>
</tr>
<tr>
<td>CIRCLE, BASIC function</td>
<td>546–547</td>
</tr>
<tr>
<td>CJA</td>
<td>117, 120</td>
</tr>
<tr>
<td>CJAE</td>
<td>117, 121</td>
</tr>
<tr>
<td>CJB</td>
<td>121</td>
</tr>
<tr>
<td>CJBE</td>
<td>121</td>
</tr>
<tr>
<td>CJE</td>
<td>121, 122</td>
</tr>
<tr>
<td>CJE</td>
<td>122</td>
</tr>
<tr>
<td>Clear, 23, 24, 32, 33, 45, 68, 75, 107</td>
<td></td>
</tr>
<tr>
<td>Clear Watchdog Timer</td>
<td>24, 33, 46, 75</td>
</tr>
<tr>
<td>CLEAR</td>
<td>PicBasic, 564</td>
</tr>
<tr>
<td>Clock</td>
<td>synchronization (See also TMRxx registers), 222</td>
</tr>
<tr>
<td>USART module</td>
<td>232–239, 233–236</td>
</tr>
<tr>
<td>CLOSE</td>
<td>BASIC function, 548</td>
</tr>
<tr>
<td>Clr</td>
<td>112</td>
</tr>
<tr>
<td>CLR Parm</td>
<td>112</td>
</tr>
<tr>
<td>CLRb</td>
<td>115</td>
</tr>
<tr>
<td>Clrc</td>
<td>107</td>
</tr>
<tr>
<td>Clrcl</td>
<td>107</td>
</tr>
<tr>
<td>Clrf</td>
<td>24, 33, 45, 75</td>
</tr>
<tr>
<td>Clrw</td>
<td>24, 33, 112</td>
</tr>
</tbody>
</table>
674  Index

cldwldt, 24, 33, 46, 75, 112, 221
crdz, 107
CLS, BASIC function, 545
CLZ, 115
CMCON register, 165–166, 257–258
CMOS, 632
code snippets, 461–481
  16-bit pulse
    measurement, 5-cycle delay, 473–474
bitwise operations, 478–479
compare and swap if Y greater than X, 476
conditional branching, 465–466
constant division, 480–481
constant multiplication, 479–480
convert ASCII byte to hex nybble, 471
convert ASCII to uppercase, 475
convert nybble to ASCII, 470–471
copy bits between registers, 470
counting 1s in a byte, 476–477
detect change in register, 474
divide by three, 472–473
generating parity for byte, 477
increment/decrement w, 468–469
jump outside current page, 461–462
keeping variable within range, 477–478
negating register contents, 467–468
rotating byte in place, 469
swap contents of registers, 475–476
swap w contents with register, 475
swapping bit pairs, 478
T0CKI used as interrupt source pin, 471–472
table code, 462–465
test byte within register, 474
time delays, 466–467
CODE, MPLAB, 526–527
coil inductance, 637
collisions, SSP module, 251–252, 252
COLOR, BASIC function, 546
color codes, resistors, 629–630, 630
comf, 25, 33, 46, 75–76, 112, 114, 115
comment, BASIC function, 541
CommEvent, Visual Basic and Windows programming, 587–588
commutative law, Boolean, 639
comparators, 257–261, 258, 260
CMCON register, 257–258
reference voltages, 259–261
compare and swap if Y greater than X, 476
Compare, 46–47, 62, 76–77, 78, 102
Complement, 25, 33, 46, 75–76, 124
complementary law, Boolean, 638
concatenation, BASIC function, 542
conditional branching, 465–466
conditional statements, C language, 591–593
CONFIG statements, 208
CONFIG, MPLAB, 527
configuration registers, 207–208
CONST, BASIC function, 543
CONSTANT, MPLAB, 527
constants
16-bit numbers, bitwise operations, 488
division, 480–481
mathematical constants, 605–606
multiplication, 479–480
physical constants, 605–606
PicBasic, 554
continue, C language, 592, 597
contrast voltage circuit, LCD, 297, 297
control attributes (properties), Visual Basic and Windows programming, 579, 582–588
control characters, ASCII, 607–609
controls, Visual Basic and Windows programming, 580–581
conventions, 1–2
conversion table, 640
copy bits between registers, 470
COS, BASIC function, 541
cos, C language, 602
COS, PicBasic, 561
COUNT, PicBasic, 564
counter, program counter, 385–386
676  Index

counting 1s in a byte, 476–477
cpfseq, 46, 76
cpfsgt, 77
cpfslt, 47, 78
CPUSTA register, 171
cqfsgt, 46–47
crystal oscillators, 208, 209, 210, 211, 212
CSB, 118
CSBE, 118
CSE, 118
CSNE, 118
CSRLIN, BASIC function, 545
Ctrl key combinations, Microsoft compatible editor, 517–518
current source/sink requirements, 500
DA/DATA/DB, MPLAB, 527
Dallas Semiconductor one-wire interface, 357–363, 358, 359
DATA, BASIC function, 540, 543
data memory, EEPROM data memory access, 263–269, 265
data types
  BASIC function, 540
  C language, 588
  PicBasic, 554
DATA, PicBasic, 564
DATES, BASIC function, 548
daw, 47, 78
DB-25, 442, 442
DC electronics formulas, 636
DCD, PicBasic, 561
dcfsnz, 49, 81
DDRB/CD/E/F/G registers, 173, 174, 179, 180
De Morgan’s theorem, Boolean, 630
DE, MPLAB, 528
DEBUG, PicBasic, 565
DEBUG_xxx, PicBasic, 555
debugging, 207, 507–515
DEBUGIN, PicBasic, 565
DEC Parm, 114
decf, 25, 34, 48, 79–80, 112, 114, 115
decfisz, 25, 34, 49, 80, 117, 120
Decimal Adjust, 47, 78
decimal numbers, 2 declarations, C language, 588
DECLARE, BASIC function, 543
Decrement, 25, 34, 48–49, 79–80, 123
16-bit numbers, 484
C language, 594
decrement w, 468–469, 468
| DECSZ, 117 |
| DEF, BASIC function, 543 |
| default, C language, 597 |
| DEFDBL, BASIC function, 543 |
| define, C language, 595 |
| DEFINE, MPLAB, 528 |
| DEFINT, BASIC function, 542 |
| DEFLNG, BASIC function, 543 |
| DEFNSG, BASIC function, 543 |
| DEFSTR, BASIC function, 542 |
| delays, time delays, 466–467 |
| detect change in register, 474 |
| device pinouts, 7–19 |
| 18-pin mid-range, 11, **11** |
| 28-pin mid-range, 11, **11** |
| 28-pin mid-range, 13, **13** |
| 40-pin mid-range, 12, **12** |
| 44-pin PLCC mid-range, 12, **12** |
| 44-pin QFP mid-range, 13, **13** |
| 64-pin mid-range, 14, **14** |
| low-end, 8–9 |

mid-range, 10–14
PIC17Cxx, 14–16, **15**
PIC18Cxx, 17–19, **18–19**
DIG, PicBasic, 561
digital logic, 282, 632
DIM, BASIC function, 540
directives, C language, 595–597
DISABLE, PicBasic, 565
display control sequences, ANSI, 609–612
distributive law, Boolean, 639
division
16-bit numbers, 495–498
BASIC function, 541
C language, 594
constant division, 480–481
divide by three, 472–473
PicBasic, 560
DJNZ, 120
DO LOOPs, BASIC function, 544
DO WHILE, BASIC function, 544
do, C language, 591, 597
do/while loop, C language, 592
double negation law, Boolean, 638
downloadable programmers, 406
DRAW, BASIC function, 547
DT, MPLAB, 528
DTMFOUT, PicBasic, 565
dual inline packages (DIP), 7, 10
DW, MPLAB, 528
18-pin mid-range pinouts, 11, 11
8250 UART
    block diagram, 443, 444
    registers, serial port, 444–449
EBCDIC code, 612, 615
edge detect circuits, 229
EEADR/EEDATA registers, 157–158, 263
EECON1/EECON2 registers, 158, 263
EEDATA register, 263
EEPROM, 157, 158, 565
EEPROM data memory
    access, 263–269, 265
    access control, 263–264
    address data, 268
    driver circuit, 265–266
    EEADR register, 263
    EECON1/EECON2
    registers, 263
    EEDATA register, 263
    GPIO port, 266
    operations, 267
PIC12CE5xx, 267
Read operations, 267, 268–269
reads, 266
start/stop bits, 266
Write operations, 264
write interrupt request
    bit (EEIE), 158
electromagnetic spectrum, 630–632
electron rest mass, 606
electron volt, 606
eif, C language, 596
else, C language, 597
ELSE, MPLAB, 528
ELSE, PicBasic, 567
ELSEIF, BASIC function, 544
ENABLE DEBUG, PicBasic, 566
ENABLE INTERRUPT,
    PicBasic, 566
ENABLE, PicBasic, 565
END, BASIC function, 541
END DEF, BASIC function, 543
END FUNCTION, BASIC
    function, 543
END SELECT, BASIC
    function, 544
END SUB, BASIC function, 543
END, MPLAB, 529
END, PicBasic, 566
ENDC, MPLAB, 529
endif, C language, 597
ENDIF, MPLAB, 529
ENDIF, PicBasic, 567
ENDIF, PicBasic, 567
ENDIF, PicBasic, 567
ENDM, MPLAB, 529
ENDW, MPLAB, 529
enhancement functions, C language, 603
EOF, BASIC function, 548
EPROM program memory access, 269–271
equals, C language, 594
EQV, BASIC function, 542
ERASE, BASIC function, 543
ERR, BASIC function, 544
ERROR, BASIC function, 544
ERROR, MPLAB, 529
ERRORLEVEL, MPLAB, 529
EXIT, BASIC function, 544
EXITM, MPLAB, 529–530
EXP, BASIC function, 541
exp, C language, 602
EXPAND, MPLAB, 530
exponentiation, BASIC function, 541
expressions, C language, 591
EXTERN, MPLAB, 530
external oscillators, 208, 211, 212, 213
external parallel memory, 273–276, 274, 275
40-pin mid-range pinouts, 12, 12
44-pin PLCC mid-range pinouts, 12, 12
44-pin QFP mid-range pinouts, 13, 13
fabs, C language, 602
farads, 2
fast stack, 147, 149–150
fast verify waveforms, programming, 384–385,
fclose, C language, 600
feature to part number table, 3–5
fgets, C language, 601
File (Variable) Registers, 137
File Register Bank, 137
FILES, BASIC function, 548
FILL, MPLAB, 530
FIX, BASIC function, 542
flash program memory access, 271–272
flash programming circuit, 384, 392, 394
flip-flops, 633–634, 634
float, C language, 602
FOR, BASIC function, 541
for, C language, 591, 592, 597
FOR, PicBasic, 566
fprintf, C language, 600
fputs, C language, 601
Index

frand, C language, 602
free, C language, 602
FREQOUT, PicBasic, 566
frequency, 638
fscanf, C language, 600
getchar, C language, 598
gigahertz (GHz), 1
GLOBAL, MPLAB, 530
GOSUB, BASIC function, 540
GOSUB, PicBasic, 566
goto, 25, 34, 49, 81, 104, 119, 120, 121, 122, 128–130, 140, 156
BASIC function, 540
C language, 593, 597
jump outside current page, 461–462
PicBasic, 566
GPIO port, EEPROM, 266
greater than, 542, 594
H-bridge motor driver circuit, 365–366, 366
Hall-effect sensors, 374, 374
hardware FAQs, 660–661
hardware interfacing (See also PC interface), 277–378
AC power control, 373, 373
asynchronous serial I/O software routines, 347–357
audio output, 371–372, 372
button interface, 284–287, 284
gas constant, 606
gas law, 638
gates, logic, 632–633, 633
generating parity for byte, 477
GET, BASIC function, 548
getc, C language, 600
Dallas Semiconductor
one-wire interface,
357–363, 358, 359
digital logic interface,
282
Hall-effect sensors, 374,
374
I2C bit banging master
interface, 334–338
infrared TV remote
control (Sony),
374–378, 375
input/output
combination, 291,
291, 292
light emitting diodes
(LEDs), 293–296,
294, 295
liquid crystal display
(LCD) (See also
liquid crystal
display), 296–333
motor drivers, 365–370,
366, 367
open collector/open
drain I/O simulation,
292–293, 293
parallel bus device
interface, 282–284,
283
potentiometer reading
using parallel I/O
pins, 363–365, 363,
364
power connections,
277–281, 278, 279,
280
R/C servo control,
370–371, 371
reset circuit, 281, 282
RS-232 interfaces,
338–345, 340, 341,
342, 344
RS-422 interface,
345–346, 345, 346,
347
RS-485 interface,
345–346, 345, 346,
347
switch matrix
keypad/keyboard
interface, 287–291,
287, 288
HC logic, 632
HCT logic, 632
hertz, 1
hex file format,
programming, 379–381
hexadecimal numbers, 2
convert ASCII byte to
hex nibble, 471
HIGH, PicBasic, 566
high-voltage circuit, 279,
280
HS oscillator, 213, 504–505
HSER_xxx, PicBasic, 555
HSERIN/HSEROUT,
PicBasic, 566
Index

I/O port definition, PicBasic, 553–554
I/O space map, 407–421
I2C
I2C operation
  bit banging master
  interface, 334–338
I2C_xxx, PicBasic, 556
I2CREAD, PicBasic, 566
I2CWRITE, PicBasic, 567
SSP module, 244–252
IBM PC extended ASCII, 612, 613, 614
ICSP programming
  connector, 405–406, 406
IDATA, MPLAB, 530–531
idempotent law, Boolean, 639
identity function, Boolean, 638
identity law, Boolean, 638
IDLOCS, MPLAB, 531
If and IF THEN
  BASIC function, 541
  BASIC function, 544
  C language, 591, 596, 597
  MPLAB, 531
  PicBasic, 563–564, 567
ifdef, C language, 596
IFDEF, MPLAB, 531
IFNDEF, MPLAB, 531
LINZ, 120
IMP, BASIC function, 542
INC Parm, 114
incf, 26, 34, 50, 82–83, 112, 114, 115
incfsz, 26, 35, 51, 83–84, 117, 120
include (inc) files, 536
Include
  C language, 595
  MPLAB, 531
  PicBasic, 567
Increment, 26, 34, 35, 50–51, 82–84, 123
  16-bit numbers, 484
  increment/decrement w, 408–469
increment, C language, 593
INCSZ, 117
INDFxx registers, 145, 170, 172, 203, 204
inductance, 637
infrared TV remote control
  (Sony), 374–378, 375
infsnz, 51, 84, 484
INKEYS, BASIC function, 545
inp, C language, 603
INPUT, BASIC function, 540, 548
INPUT, PicBasic, 567
input/output
  asynchronous serial I/O
  software routines,
  347–357
audio output, 371–372,
  372
combining I/O, 291,
  291, 292
I/O pin current
  capabilities, 500
I/O space map, 407–421
ISA bus, pinouts,
  425–428, 425, 426
open collector/open
  drain I/O simulation,
  292–293, 293
parallel port, 455–460,
  455
PicBasic I/O port
  definition, 553–554
ports, 217–221, 218,
  219, 220
potentiometer reading
  using parallel I/O pins,
  363–365, 363, 364
serial port, 441–454
INPUTS, BASIC function, 545
INSTR, BASIC function, 545
instruction programming,
  388–390
instruction sets, 21–122
liquid crystal displays
  (LCD), 298–299
  low-end, 22–30
mid-range, 31–39
MPASM, 104–110, 111
Parallax Inc., 111–122
parameters, 21–22
PASM, 111–112
PIC17Cxx, 40–62
PIC18Cxx, 63–103
PicBasic, 565–577
special instruction
  mnemonics, 104–110
INT, BASIC function, 541
INTCON registers, 134–135,
  157, 158, 204–205, 225
internal clocking oscillators,
  208, 209
Internet list servers,
  PICmicro® MCU,
  648–655
interrupt handler skeleton,
  135–136, 142, 150
interrupt handlers, PicBasic,
  557–559
interrupt requests, 134–135,
  142
interrupts/interrupt
  operations, 134–135,
  140–141, 150
INTCON register, 157
interrupt function by
  number, PC, 422–424
ISA bus, 429–430, 429
PicBasic, 557–559
register mapping, 157,
  158
interrupts/interrupt operations, continued
RS-232 serial port
interrupt APIs (14h), 451–454
serial port, 446, 446, 449–450
T0CKI used as interrupt source pin, 471–472
INTSTA register, 172
ior, 35
iorlw, 26, 51, 85, 114, 115
iorwf, 26, 35, 52, 85, 114
IPR registers, 190–192
ISA bus
BIOS interfaces, 436–440
interrupts, 429–430, 429
keyboard and mouse ports, 430
keyboard commands, 441
keyboard controller commands, 436
keyboard scan codes, 432–435
keyboard timing diagrams, 431–432
keyboard/mouse connectors, 430–431, 431
PC interfaces, 425–428, 425, 426
isalnum, C language, 603
isalpha, C language, 602
isdigit, C language, 602
islower, C language, 602
isspace, C language, 603
isupper, C language, 602
JB, 120
JC, 120
JMP Parm, 119
JNB, 120
JNC, 120
JNZ, 120
jump operations, 129, 134
jump outside current page, 461–462
jump to, 25, 34
JZ, 120
keyboards
commands, 441
connectors, 430–431, 431
controller commands, 436
Ctrl key combinations, Microsoft compatible editor, 517–518
key scan, 287–291
ports, 430
scan codes, 432–435
switch matrix interface, 287–291, 287, 288
timing diagrams, 431–432
Index 685

KILL, BASIC function, 548
kilohertz (kHz), 1
Kirashov, Dmitry, 476
labels, PicBasic, 557
LATA/LATB/LATC/LATD/
LATE registers, 188
LBOUND, BASIC function,
543
lcall, 52, 104, 107, 119
LCASES, BASIC function,
545
LCD drivers, 10
LCD_xxx, PicBasic, 556
LCDOUT, PicBasic, 567
least significant bit (LSB),
PicBasic, 560
LED control registers,
register mapping, 168
LEFTS, BASIC function, 545
LEN, BASIC function, 545
less than, 542, 594
LET, BASIC function, 540
lfsr, 86
lgoto, 108
light emitting diodes
(LEDs), 293–296, 294, 295
light, speed of, 605
LINE, BASIC function, 546
LINE INPUT, BASIC
function, 544
linking-linked applications,
536–539
liquid crystal display (LCD),
296–333
basic interface macro
(LCD8), 305–310
bit descriptions for
commands, 299–300
color box, 304–305,
304
character set, 302, 303,
304
contrast voltage circuit,
297, 297
four-bit mode
initialization/interface,
315–324
initialization, 305
instruction set,
298–299
moving to locations
within display,
300–305, 302
polling LCD, LCD8Poll
macro, 310–315
read/write waveforms,
297–298, 298
two-wire interface,
324–333, 324, 325
LIST, MPLAB, 531–532
LJMP, 119
Load, 27, 36, 86, 108,
110
LOCAL, MPLAB, 532–533
LOCATE X, Y, BASIC
function, 546
686 Index

LOG, BASIC function, 541
log, C language, 602
log10, C language, 602
logic interface, 282
logic, digital, 632
Long Call, 107
Long Goto, 108
LOOKDOWN/LOOKDOWN2, PicBasic, 567–568
LOOKUP/LOOKUP2, PicBasic, 568
loop, 492, 592
LOW, PicBasic, 568
low-end device pinouts, 8–9, 8
LP oscillator; 213, 501–502
LPRINT, BASIC function, 547
LSET Addr, 116
LTRIMS, BASIC function, 545
LVDCON register, 201–202
MACRO, MPLAB, 533
Mainline application
statement, C language, 590
malloc, C language, 602
mapping
I/O space map, 407–421
register mapping, 151–206
mass, 606
Master SSP (MSSP) module, 162–163, 345–347, 251–252, 252
mathematical constants, 605–606
mathematical operators, PicBasic, 559–560
MAX, PicBasic, 561
MAXRAM, MPLAB, 533
megahertz (MHz), 1
memory map, PC interfaces, 407, 408
memory space, 142
MESSG, MPLAB, 533
metric conversion, 640
Microchip Corporation, 641–642
mid-range device pinouts, 10–14
MIDS, BASIC function, 545
milli-/microseconds, 2
MIN, PicBasic, 561
MOD, BASIC function, 542
modems
AT command, 216–221
registers, 622–624
modulus, C language, 594
Morse code, 625–626
MOSFET, 209
most significant bit (MSB), PicBasic, 560
motor drivers/control, 365–370, 366, 367
H-bridge circuit, 365–366, 366
pulse wave modulation (PWM) control, 366–367, 367
stepper motor control, 368–370, 369
mouse ports, 430
MOV, 112
MOVB, 116
move, 27, 30, 39, 40, 52, 53, 86, 87, 123, 169
movf, 27, 35, 86, 112, 113, 114, 115, 117, 120, 121, 122
movf, 86
movfp, 52
movlw, 108, 110
movlw, 52, 87
movlw, 27, 36, 53, 87, 112, 113, 114, 117, 118, 120, 121, 122, 404
movpf, 53
MOVSZ, 117
MOV, 27, 36, 53, 87, 112, 113, 119, 404
MPASM, 104–110, 111
MPLAB, 208, 519–535, 520
assembler directives, 526–535
Asynchronous Stimulus window, 521–522, 522
clocked stimulus, 522–523, 522
Ctrl key combinations, Microsoft compatible editor, 517–518
file formats accessed, 519–520
function integration, 519
High option, 522
Low option, 522
MPLAB (See also MPLAB), 519–535, 520
MPSIM.INI, 518–519
Pulse option, 522
Register modify, 523–524, 524
register monitoring (Watch window), 520–521, 521
Register stimulus, 523–524, 524
Stimulus (.STI) files, 525
Toggle option, 522
MPSIM.INI, 518–519
MS-comm control, Visual Basic and Windows programming, 581–588
mull, 87
mullw, 53
mullwf, 53, 87–88
Index

multiplication
16-bit numbers, 492–495
BASIC function, 541
C language, 594
constant multiplication, 479–480
PicBasic, 560
multiplier, 142
Multiply, 53, 87–88

NAME, BASIC function, 548
NAND gate, 633, 633
NAP, PicBasic, 568
NCD, PicBasic, 561
NEG Parm, 114
Negate, 54, 88, 108, 123
negating register contents, 467–468
negative active pin, 2
negf, 104, 108
negw, 54, 88
neutron rest mass, 606
NEXT, PicBasic, 566
No operation, 27, 36, 54, 89, 215
NOEXPAND, MPLAB, 533
NOLIST, MPLAB, 533
non-return to zero (NRZ), 164
nop, 27, 36, 54, 89, 116, 215, 271–272, 403
NOR gate, 633, 633

NOT, 115
bitwise operations, 478–479
Boolean, 638, 639
not equals, 542, 594
NOT gate, 633, 633
nybble
convert ASCII byte to hex nybble, 471
convert nybble to ASCII, 470–471

Ohm’s law, 635
ON DEBUG GOTO, PicBasic, 569
ON ERROR GOTO, BASIC function, 544
ON GOTO, BASIC function, 541
ON INTERRUPT GOTO,
PicBasic, 569
OnComm event, Visual Basic and Windows programming,
357–363, 358, 359
OPEN, BASIC function, 548
open collector/open drain I/O simulation, 292–293,
Index 689

operators
BASIC function, 541–542
C language, 593–595
PicBasic, 559–560
option, 27, 36, 112, 281
OPTION, BASIC function, 540
OPTION register, 136, 152–156, 216–217, 223
options, PicBasic, 549–550
OR, 26, 35, 51, 52, 85, 114, 124
BASIC function, 542
bitwise operations, 2, 478–479
Boolean, 638, 639
C language, 593, 594, 595
PicBasic, 560
OR gate, 633, 633
order of operations, BASIC function, 542
order of operations, C language, 594
ORG, MPLAB, 533
OSC, PicBasic, 556–557
OSCCAL register, 152, 157, 209, 220
OSCCAL, 1K/2K, PicBasic, 556
OSCCON register, 202
oscillator calibration (OSCCAL) register, 126
Oscillator Mode, 207
oscillators, 208–213, 209, 210, 212
analog to digital conversion (ADC), 254
ceramic resonator, 208, 209, 211
crystal, 208, 209, 210, 211, 212
external, 208, 211, 212, 213
HS, 213, 504–505
internal clocking, 208, 209
LP, 213, 501–502
modes, 211, 213
OSCCAL register, 157
PIC18Cx, 211, 214
oscillators, continued
R/C network, 208, 209, 210, 213, 500–501, 502
speed ranges, 211
TMR1, 213, 224–226, 225
XT, 213, 502–504, 503
outp, C language, 603
output set/reset, Boolean, 638
OUTPUT, PicBasic, 569

page address, 129
PAGE, MPLAB, 533
PAGESEL, MPLAB, 533
PALETTE, BASIC function, 546
Parallax Inc. MCU
instruction set, 111–122
parallel bus device interface, 282–284, 283
parallel capacitance, 636
parallel memory, external, 273–276, 274, 275
parallel port, 455–458
base registers, 457
BIOS interfaces, 459–460
block diagram, 455–456, 455
connectors, 455–456, 456
data output waveform, 458, 458
potentiometer reading
using parallel I/O pins, 363–365, 363, 364
registers, 457–458
Parallel protocol
programming, 381–385
parallel resistance, 636
parallel slave port (PSP), 261–263, 261
PIE/PIR registers, 158
PORTxx registers, 262–263
TRISE registers, 262
parameters, 2, 21–22
parity, generating parity for byte, 477
parsec, 606
part suppliers, 661–665
PASM instruction set, 111–112
PAUSE, PicBasic, 569
PAUSEUS, PicBasic, 569
PC interfaces (See also hardware interfacing), 407–460
8250 UART registers, 444–449
BIOS interfaces, ISA bus, 436–440
BIOS interfaces, parallel port, 459–460
I/O space map, 407–421
interrupt function by number, 422–424
interruption, ISA bus, 429–430, 429
interrupts, RS-232 APIs (14h), 451–454
interrupts, serial port, 449–450
ISA bus, pinouts, 425–428, 425, 426
keyboard and mouse ports, 430
keyboard commands, 441
keyboard controller commands, 436
keyboard scan codes, 432–435
keyboard timing diagrams, 431–432
keyboard/mouse connectors, 430–431, 431
memory map, 407, 408
parallel port, 455–460, 455
serial port base addresses, 443–444
serial port, pinouts, 441–454, 442
PCKATHU register, 206
PCL register, 134, 140, 170, 206
PCLATH register, 134, 140, 147, 156, 170, 206, 464
PCLATU register, 147, 464
PCON register, 156
PEEK, PicBasic, 569
perfect gas law, 638
periodicals on PICmicro topics, 658–659
peripheral device interrupts, PIE/PIR registers, 157–158
permeability of vacuum, 606
permittivity of free space, 605
phonetic alphabet, 626
physical constants, 605–606
pi, 606
PIC12C5xx architecture, 125
pinout, 8, 8
programming, 390, 391
PIC12C6xx architecture, 130
PIC12CE5xx EEPROM, 267
PIC14000 architecture, 130
pinouts, 10, 10, 13, 13
PIC16C5xx architecture, 125, 130
pinouts, 8, 8, 9, 9
programming, 390, 391
PIC16C54 pinouts, 9
PIC16C56 pinouts, 9
PIC16C57 pinouts, 9
PIC16C6xx architecture, 130
PIC16C7x architecture, 130
PIC16C8x architecture, 130
692  Index

PIC16C9xx
architecture, 130
pinouts, 10, 10, 14
PIC16F8x programming
cycle, 393–394
PIC17C4x 40-pin pinouts, 15
PIC17C4xx 44-pin pinouts, 16
PIC17C7xx 64-pin pinouts, 15
PIC17Cxx
accumulator, 136
architecture, 124,
136–142, 137
banking scheme, 169
EPROM program
memory access,
269–271
File Register Bank,
137
instruction set, 40–62
interrupt handler
skeleton, 142
interrupt operation,
140–141
parallel memory,
external, 273–276,
274, 275
pinouts, 14–16, 15, 16
primary banks, 169
primary register set,
169
program counter, 137,
139–140, 140
programming, 394–398,
395, 396, 397
register access, 137–138
register mapping,
169–185
PIC18C2X2 28-pin
pinouts, 17
PIC18C4X2 40-pin
pinouts, 18
PIC18C4X2 44-pin pinouts,
18, 19
PIC18Cxx
Access Bank, 143
accumulator, 143
architecture, 124,
142–150, 143
fast stack, 147, 149–150
instruction set, 63–103
interrupt handler
skeleton, 150
interrupt operation, 150
interrupt requests, 142
memory space, 142
multiplier, 142
oscillator/oscillator
modes, 211, 214
pinouts, 17–19, 18–19
program counter, 143,
147–150, 148
programming, 402–405,
403, 404
push operations, 145
register access,
144–146, 144
register mapping, 186–206
reset address, 143
PicBasic, 549–577
arrays, 553
assembly language use, 557
configuration values, 550
constant definition, 554
data types, 554
developer/supplier, website, 549
I/O port definition, 553–554
If statement, 563–564
instructions, 565–577
interrupt handlers, 557–559, 557
labels, 557
mathematical operators, 559–560
options, 549–550
symbols, 553
variables, 550–553
PIC12C509 pinout, 8, 8
PICList Internet List Server, 648–655
PIE/PIR registers, 134–135, 157, 158, 175, 178–179, 189–190, 191, 225, 228
pin-through-hole (PTH), 7, 8
Planck’s constant, 606
PLAY, BASIC function, 547
PLUSW0/1/2 registers, 203–204
pointers, C language, 590
POKE, PicBasic, 569
Pop, 89
ports, keyboard and mouse ports, 430
POS, BASIC function, 546
POSTDEC0/1/2 registers, 203–204
POSTINC0/1/2 registers, 203–204
POT, PicBasic, 569
potentiometer reading using parallel I/O pins, 363–365, 363, 364
pow, C language, 602
power, 635
power connections, 277–281, 278, 279, 280
Power Down state, 57, 94
PR1/PR2 register, 176, 200
PR3L/PR3H register, 176
pragma, C language, 597
PREINC0/1/2 registers, 203–204
prescaler, 223–234, 224
PRESET, BASIC function, 546
PRINT, BASIC function, 540, 548
PRINT USING, BASIC function, 546
printf, C language, 598–599
PROCESSOR, MPLAB, 534
PRODL/PRODH registers, 185, 205
program counter, 128–130, 129, 133–134, 133, 137, 139–140, 140, 143, 147–150, 148
program memory checking before programming, 387–388
EPROM program memory access, 269–271
flash program memory access, 271–272
Program Memory Protection, 207
programming, 379–406
boot code, 401–402
counter, program counter, 385–386
data packet commands, 392–393
fast verify waveforms, 384–385, 384
flash programming circuit, 391, 392, 394
hex file format, 379–381
ICSP programming connector, 405–406, 406
initializing programming mode, 385–386, 386
instruction programming, 388–390
low-end devices, 381–385, 382, 383, 384
mid-range devices, 385–394
Parallel protocol, 381–385
PIC12C50x, 390
PIC16C50x, 390
PIC16F8x programming cycle, 393–394
PIC17Cxx ICSP, 398–402, 401
PIC17Cxx, 394–398, 395, 396, 397
PIC18Cxx, 402–405, 403, 404
program memory check, 387–388
programming mode initialization, 385–386, 386
serial programming, 385–394
third-party/downloadable programmers, 406
voltage/current requirements, 391–392
programming connector, ICSP, 405–406, 406
proton rest mass, 606
PSET, BASIC function, 546
PSMODE, 158
pulse measurement, 16-bit, 5-cycle delay, 473–474
pulse wave modulation (PWM) motor control, 366–367, 367
PULSIN/PULSOUT, PicBasic, 569
Push, 89, 145
PUT, BASIC function, 548
putc, C language, 600
putchar, C language, 598
PW1DCH/PW2DCH registers, 176
PW1DCL/PW2DCH registers, 176
PW3DCL/PW3DCH registers, 184
PWM mode, 230–232, 569
quad plastic chip carriers, 7
? (question mark command), 540, 591
R/C network oscillators, 208, 209, 210, 213, 500–501, 502
R/C servo control, 370–371, 371
radar bands, 632
radio Ten codes, 626–628
RADIX, MPLAB, 534
RANDOM, PicBasic, 570
RANDOMIZE, BASIC function, 540
RC charging/discharging, 637
RC time constant, 636
rcall, 89
RCIE/RCIF registers, 164
RCON register, 200–201
RCREG register, 164, 174, 179, 193
RCSTA register, 174, 192, 236, 237
RCTIME, PicBasic, 570
Read, 61, 100–101
READ, BASIC function, 540, 544
READ, PicBasic, 570
REDM, BASIC function, 543
reference voltages, 259–261
register access, 126–127, 130–132, 131, 137–138, 144–146, 144
radar bands, 632
radio Ten codes, 626–628
RADIX, MPLAB, 534
RANDOM, PicBasic, 570
RANDOMIZE, BASIC function, 540
RC charging/discharging, 637
RC time constant, 636
rcall, 89
RCIE/RCIF registers, 164
RCON register, 200–201
RCREG register, 164, 174, 179, 193
RCSTA register, 174, 192, 236, 237
RCTIME, PicBasic, 570
Read, 61, 100–101
READ, BASIC function, 540, 544
READ, PicBasic, 570
REDM, BASIC function, 543
reference voltages, 259–261
register access, 126–127, 130–132, 131, 137–138, 144–146, 144
696  Index

registers/register mapping
(See also individual register name entries), 151–206
addresses, mid-range devices, 154–155
copy bits between registers, 470
detect change in register, 474
interrupts, 157, 158
low-end devices, 151–154, 152
mid-range devices, 154–168
modem registers, 622–624
MSSP module, 162–163, 162
negating register contents, 467–468,
non-return to zero (NRZ), 164
parallel port, 457–458
swap contents of registers, 475–476
swap w contents with register, 475
Synchronous Serial Protocol (SSP)
module, 161–162, 161
test byte within register, 474
REM, BASIC function, 541
remote control, infrared, TV (Sony), 374–378, 375
RES, MPLAB, 534
reserved words, C language, 597
Reset, 90, 126, 137, 143
RESET, BASIC function, 548
reset circuit, 281, 282
reset parameters, 207
resistors
color coding, 629–630,
630 parallel resistance, 636
series resistance, 635
resonance, 636
resources, 641–665
RESTORE, BASIC function, 541
RESUME, BASIC function, 544
Resume Execution, 27, 36, 37
RESUME, PicBasic, 570
RET, 119
retfie, 36, 54, 90
retlw, 27, 36, 55, 90, 119
return, 27, 37, 54, 55, 90, 91,
149
BASIC function, 540
C language, 597
PicBasic, 570
RETW, 119
Index 697

REV, PicBasic, 561
REVERSE, PicBasic, 570
RIGHTS, BASIC function, 545
RL Parm, 115
RL time constant, 636
rlcf, 55, 91
rlcnf, 55, 92
rlf, 28, 37, 112
RND, BASIC function, 541
Rotate, 28, 37, 55–56, 91, 92, 93
rotate byte in place, 469
RR Parm, 115
rrcf, 56, 92
rrcnf, 56, 93
rrf, 28, 37, 112, 115
RS-232 interfaces, 338–345,
340, 341, 342, 344, 345, 346, 347
interrupt APIs (14h), 451–454
RS-422 interface, 345–346,
345, 346, 347
RS-485 interface, 345–346,
345, 346, 347
16-bit numbers, 483–498
addition, 485–488
address declaration/defining, 483–484
bitwise operations, constants and variables, 488
comparing variables, 489–491
decrement, 484
division, 495–498
increment, 484
multiplication, 492–495
subtraction, 485–488
Save Stack Pointer, 24, 33
SB, 117
SC, 117
scan codes, keyboard, 432–435
scanf, C language, 599–600
Schmidt trigger, 209, 222
SCREEN, BASIC function, 546
SEEK, BASIC function, 548
SELECT, BASIC function, 544
sensors, Hall-effect sensors, 374, 374
separators, BASIC function, 541
SER2_BITS, PicBasic, 556
serial communications, 164
serial port
8250 UART block diagram, 443, 443
8250 UART registers, serial port, 444–449
asynchronous serial I/O
software routines, 347–357
698 Index

serial port, continued
  base addresses, 443–444
  connector pinouts, 442, 442
  DB-25, 442, 442
  interrupt enable
    hardware, 446, 446
    interrupts, 449–450
    PC interfaces, 441–454
  RS-232 interrupt APIs
    (14h), 451–454
  RS-232, 338–345, 340, 341, 342, 344, 442, 442
  RS-422 interface
    345–346, 345, 346, 347
  RS-485 interface
    345–346, 345, 346, 347
  serial programming, 385–394
  series capacitance, 636
  series resistance, 635
  SERIN/SEROUT, PicBasic, 570–574
  SERIN2/SEROUT2, PicBasic, 571–575
  servo control, R/C, 370–371, 371
  Set, 24, 32, 45, 56, 72, 93, 108
  SET, MPLAB, 534
  SETB, 115
  setc, 108
  setdc, 108
  setf, 56, 93
  setz, 108
  seven-segment LED display, 293–296, 294, 295
  SGN, BASIC function, 541
  Shift, 124
  shift left/right, C language, 594
  shift, PicBasic, 560
  SHIFTI NSH IFT OUT, PicBasic, 575–576
  SIN, BASIC function, 541
  sin, C language, 602
  SIN, PicBasic, 561
  Skip, 24, 32, 33, 109, 119
  skpc, 109
  skpdc, 109
  skpnc, 109
  skpndc, 109
  skpnc, 109
  skpndz, 109
  skpz, 109
  sleep, 28, 37, 57, 94, 116, 213–215, 215
  SLEEP, PicBasic, 576
  SNSN, 117
  software development tools, 517–603
  sound
    audio notes, 615–616
    speed of, 606
  SOUND, BASIC function, 547
Index 699

SOUND, PicBasic, 576
SPACE, MPLAB, 534
SPACES, BASIC function, 545
SPBRG register, 174, 179, 193, 233, 237
SPC, BASIC function, 546
speakers, audio output, 371–372, 372
Special Function Registers (SFR), 132, 137
special instruction mnemonics, 104–110
speed of light, 605
speed of sound, 606
SPI operation, SSP module, 239–244, 241, 242, 243
SQR, BASIC function, 541
SQR, PicBasic, 561
sqrt, C language, 602
SSPADD register, 182, 199, 245, 247
SSPBUF register, 161, 184, 199, 243–245
standard declaration files, 536
Standby mode, 28, 37
STC, 115
Stefan-Boltzmann constant, 606
stepper motor control, 368–370, 369
Stimulus (.STI) files, MPLAB, 525
STKPTR register, 148–149, 206
STOP, BASIC function, 541
STOP, PicBasic, 576
store value, 27, 36
strcat, C language, 601
strchr, C language, 601
strcmp, C language, 601
strcpy, C language, 601
strncpy, C language, 601
strchr, C language, 601
string operations
  BASIC function, 541
  C language, 589, 601–602
strlen, C language, 601
strncat, C language, 601
strncpy, C language, 601
strstr, C language, 602
STRS, BASIC function, 545
STZ, 115
SUB, BASIC function, 543
SUB Parm, 113
700  Index

SUBB, 115
subcf, 109–110
sublw, 38, 57–58, 95–96, 468
SUBTITLE, MPLAB, 534
Subtract, 29, 38, 57–60, 94–99, 109–110, 123
subtraction
  16-bit numbers, 485–488
  BASIC function, 541
  C language, 594
  PicBasic, 560
subwf, 29, 38, 58–59, 97–98, 112, 113, 117, 118, 120, 121, 122
subwfb, 59–60, 94–95, 98–99, 486
suppliers, 661–665
surface mount technology (SMT), 7, 8
Swap, 29, 39, 60, 100, 475–476
SWAP Parm, 115
SWAP, PicBasic, 576
swapf, 29, 39, 60, 100, 112, 115
swapping bit pairs, 478
switch matrix
  keypad/keyboard interface, 287–291, 287, 288
switch, C language, 591, 593, 597
switches, Hall-effect sensors, 374, 374
symbols, PicBasic, 553
synchronization, 222
synchronous serial communications, 164
Synchronous Serial Protocol (SSP) module, 239–252, 240
  Basic SSP (BSSP), 240
  clock setting, 240
  collisions, 251–252, 252
  I2C operation, 244–252, 244
  Master SSP (MSSP), 162, 245–247, 251–252, 252
  receive data from I2C device, 249–251
  register mapping, 161–162
  SDA/SDL connections, 247, 247
  send data to I2C device, 248–249
  SPI operation, 239–244, 241, 242, 243
  SSPADD register, 245, 247
  SSPBUF register, 243–245
  SSPCON register, 240–241, 244–245, 252
SSPSTAT register, 241–242, 245, 246
system, C language, 602

28-pin mid-range pinouts, 11, 11, 13, 13
T0CKI used as interrupt source pin, 471–472
T0CON register, 202–203
T1CON register, 177–178, 200
T2CON register, 159–160, 199–200, 227
T3CON register, 185, 193
TAB, BASIC function, 541
TABLATH/TABLATL
  registers, 205, 274, 398, 399
table code, 462–465
tablrd, 61, 100–101, 274, 275
tablwt, 61, 101–102, 274
TBLPTHR/TBLPTRL
  registers, 172, 205, 274, 399
thldr, 402
thlwt, 402, 404
template, application code, 539
“Ten” radio codes, 626–628
test byte within register, 474
Test Register, 45, 72, 73, 115
thermometer application, one-wire interface,
  358–359, 358
third-party/downloadable programmers, 406
thyristors, 373, 373
time constants, 636
time delays, 466–467
TIMER, BASIC function, 548
TIMES, BASIC function, 548
timing operations,
  TMR0/TMR1/TMR2
  registers, 159–160
TITLE, MPLAB, 534
thlr, 61
thlr, 61
TMR0/TMR1/TMR2
  registers, 159–160, 176, 200, 203, 213, 222–228,
    222, 225, 227, 399
T0CKI used as interrupt source pin, 471–472
TMR0L/TMR0H registers, 203
TMR1L/TMR1H registers, 200
TMR2H/TMR2L registers, 176, 193
TMR3H/TMR3L registers, 172
Toggle Bit, 45, 73
TOGGLE, PicBasic, 570
tolower, C language, 603
702 Index

TOSL/TOSH register, 206
TOSTA register, 170–171
TOSU register, 206
Touch-Tone telephone
frequencies, 616
toupper, C language, 603
transformer current/voltage,
637
transmission line
characteristic
impedance, 637
triacs, 373, 373
tris, 30, 39, 112, 188–189,
218, 283
TRIS registers, 152, 158,
217–221, 218, 219,
220, 262
tstfz, 62, 102
TTL, 632
turns ratio, transformers,
637
TV remote control, infrared
(Sony), 374–378, 375
TX register, 164
TXIE register, 164
TXREG register, 164, 174,
179, 192, 234, 238–239
TXSTA register, 164, 174,
179, 192, 236–237

UARTS
8250 registers, serial
port, 444–449

8250 UART block
diagram, 443, 443
UBOUND, BASIC function,
543
UCASES, BASIC function,
545
UDATA, MPLAB, 535
UDATA_ACS, MPLAB, 535
UDATA_OVR, MPLAB, 535
UDATA_SHR, MPLAB, 535
undfne, C language, 595
UNDEFINE, MPLAB, 535
units of measure, 1–2
USART module, 232–239,
233–236
asynchronous data,
235–236
data errors, 235
holding register, 235
RCSTA register, 236,
237
register mapping, 164
SPBRG register, 233,
237
TXREG register, 234
TXREG register,
238–239
TXSTA register,
236–237

VAR, BASIC function, 545
VARIABLE, MPLAB, 535
variables
  16-bit numbers, bitwise operations, 488
  16-bit numbers, comparing values, 489–491
  BASIC function, 540, 542
  C language, 588
  keeping variable within range, 477–478
  PicBasic, 550–553
  Visual Basic and Windows programming, 580
  VIEW, BASIC function, 546
  Visual Basic and Windows programming, 578–588, 578
  CommEvent, 587–588
  control attributes (properties), 579, 582–588
  controls, 580–581
  desktop, 579
  functions, 579–580
  MScomm control, 581–588
  OnComm event, 587–588
  variables, 580
  Windows programming, 578–588
  voltage comparator, 165–166
voltage reference source, 165, 259–261
VRCON register, 165
Warren, Andy, 472
Waste Instruction, 27
Watchdog Timer, 24, 33, 46, 75, 207, 221
WDTCON register, 201
web sites, PICmicro® MCU, 656–658, 659–661
WEND, PicBasic, 576
Wheatstone bridge, 636
while, C language, 591, 597
WHILE, MPLAB, 535
WHILE, PicBasic, 576
WINDOW, BASIC function, 546
Windows ASCII characters, 612, 615
Windows programming (See also Visual Basic), 578–588
WREG register, 143, 145, 172, 204
Write, 61, 101–102
WRITE, BASIC function, 548
WRITE, PicBasic, 576
XIN/XOUT, PicBasic, 577
704  Index

XOR, 30, 39, 62, 103, 114, 123, 124, 131
BASIC function, 542
bitwise, 2
C language, 593, 594, 505
PicBasic, 560

XOR gate, 633, 633
xorlw, 30, 39, 62, 103, 114, 115
xorwf, 30, 39, 62, 103, 114
XT oscillators, 213, 502–504, 503
Data in the Palm of Your Hand
PICMicro® Microcontroller Pocket Reference

Device, code, and circuit data on one of the most popular microcontrollers around—it’s in your pocket with this guide. You’ll never again get stuck in the World Wide Wait or shuffle through coffee-stained printouts, looking for the facts you need. Used alone or with Myke Predko’s Programming & Customizing PICMicro® Microcontrollers, this data source saves you from searching for:

- Complete pin-out mappings
- Device feature comparisons
- Processor instructions
- Multi-tier data tables
- Frequently used code snippets
- Clock tables
- Power consumption tables
- Debugging hints
- Common external device interfaces with sample code
- Other handy data

Short on verbiage, and long on facts, this is the ideal data tool for the experienced applications designer.

Visit us on the World Wide Web at www.books.mcgraw-hill.com

McGraw-Hill
A Division of The McGraw-Hill Companies

Cover design by 12E Design • Cover photo by PhotoDisc

ISBN 0-07-136175-8

9 780071 361750