PICBOT FIRMWARE

Documentation for PicBot Project using
the Motorola PIC 16F877 microcontroller

by Miki R. Marshall
2018.05.29

Describes the design of the Microchip PIC programming, the usage of the pins, how it is setup, the protocol used to communicate with it, and all that good stuff. See other documents for the physical hardware layouts and schematics of the circuit design, although this does have a close correlation with the firmware shown here.

Table of Contents

PicBoot

INTRODUCTION

PicBoot is separate from PicBot in that it is not loaded at the same time and resides in a dedicated space at the end of Page3. PicBoot is used to load the PicBot program. It then can run the PicBot program, just as PicBot can jump to PicBoot to have a new PicBot hex file downloaded into it. The two programs do not interact otherwise.

PicBoot is loaded (hopefully) once upon acquiring a new chip, currently using the 2-Bit parallel port program. Once this is complete, bootloading the PicBot program is a simple matter of connecting the SCI to the serial port of a PC and sending the hex file as a text file using a simple terminal program, or the downloader I've built into the MainBot (Qbasic) module.

PROTOCOL

This is how to interact with the PIC while in "BootLoad" mode.

Reset Responses
(A4 pin high on reset, or after a “PB” command)

"PR" - PicBot is running (Reset just occurred).
"PB" - PicBoot mode activated.

This means that if DosBot suddenly receives a "PB", it knows the PIC is in Boot Mode and can switch to that mode itself, prompting the user for a file, command, etc. To cause a reboot within PicBot, GOTO address 1FFF, which contains a GOTO instruction for the beginning of PicBoot code.

PicBoot Commands
(Commands recognized in the PicBoot mode)

":" - Start of each record in a valid HEX file. Starts the loading of a new record.
"R" - Entered at the "PB" prompt, this activates a check to ensure that user code exists, and if so, runs it.
"{Esc}" - Aborts a load (part of original Karl Lunt code, not sure if it works because it seems to have to fall at the beginning of a record...). Returns to "PB" prompt.

PicBoot Responses
"E0" - Unexpected command character received (first character in line).
"E1" - Checksum error during upload.
"E2" - Reset vector code not found first in user code.
"E3" - Boot code overrun by user code.
"E4" - Run command failed - no user code to run.
"OK" - Load successful.

All of the status messages above immediately jump back to the PicBoot prompt “PB”.

Responses During Upload
All characters received during the upload are automatically echoed back to the sender. Since the sender (a PC) runs SO much faster than the PIC (at 20 MHz), this echoing performs a sort of protocol and error checking function. The PC should not send the next character until it receives the echo of the last character (handshaking) and checks to make sure the echoed character is identical to the one sent (error checking).

Pin Usage

Intended uses for all I/O pins on the PIC 16F877. Many may have multiple uses and are listed in order of preference or expected use (the first being the default). This is the current design and will probably change over time.

See PicBot SCI Protocol for a description for the use of each Port Pin Address.

Address	*Pin*	*Principal Use*	*Alternate Uses*	
PORT A				
A0	A0	A/D Input		
A1	A1	A/D Input		
A2	A2	A/D Input		
A3	A3	A/D Input		
T0	A4	Timer		
A4	A5	A/D Input		
PORT B				
I0	B0	Ext. Interrupt (X0)	Digital Input	
I1	B1	Digital Input		
I2	B2	Digital Input		
I3	B3	Digital Input		PGM
I4	B4	Digital Input	INT on Change (X1)	
I5	B5	Digital Input	INT on Change (X2)	
I6	B6	Digital Input	INT on Change (X3)	PGC
I7	B7	Digital Input	INT on Change (X4)	PGD
PORT C				
T1	C0	Timer	Digital Output	
S3	C1	Servo	Digital Output	
S4	C2	Servo	Digital Output	
--	C3	SPI-Clock	Digital Output	
--	C4	SPI-Data In	Digital Output	
--	C5	SPI-Data Out	Digital Output	
--	C6	SCI-Tx		
--	C7	SCI-Rx		
PORT D				
O0	D0	Digital Output	Motor 0, 0*	
O1	D1	Digital Output	Motor 0, 1*	
O2	D2	Digital Output	Motor 0, 2*	
O3	D3	Digital Output	Motor 0, 3*	
O4	D4	Digital Output	Motor 1, 0*	
O5	D5	Digital Output	Motor 1, 1*	
O6	D6	Digital Output	Motor 1, 2*	
O7	D7	Digital Output	Motor 1, 3*	
PORT E				
S0	E0	Servo	Digital Output	
S1	E1	Servo	Digital Output	
S2	E2	Servo	Digital Output	

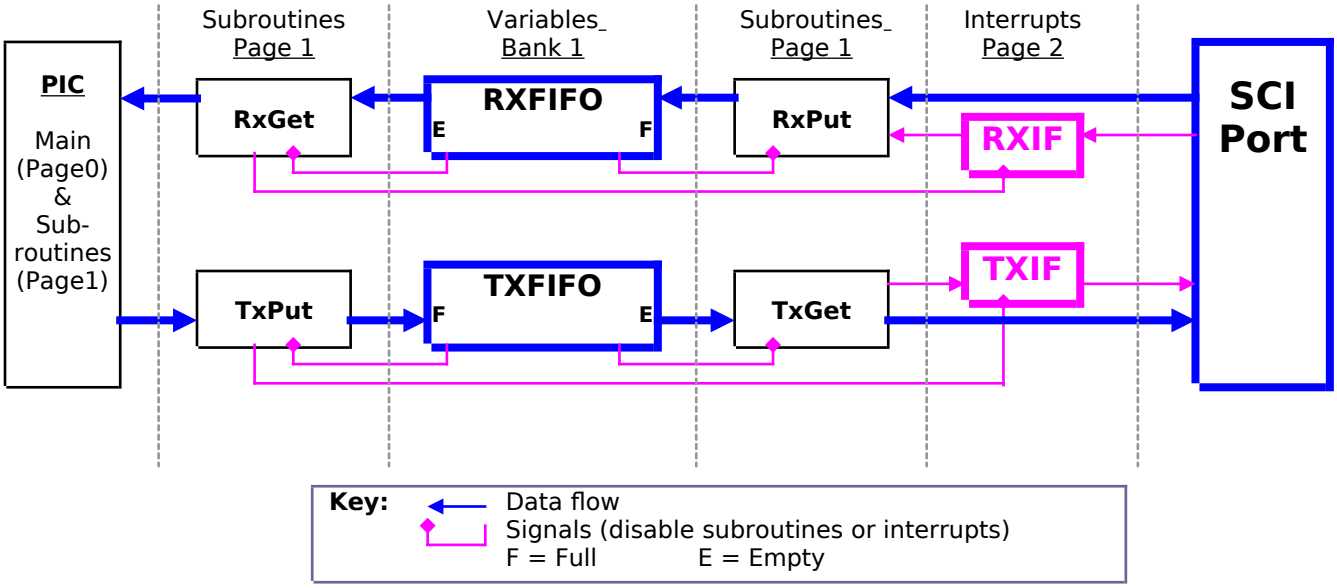
** Obsolete. Remove when code is permanently removed.*

SCI FIFO Buffer

The SCI First-In-First-Out Buffer, a well-oiled machine that has given me not a lick of trouble since I designed the thing. Here's how I did it.

DESIGN

The following graphic attempts to describe not only the dataflow and interrupts, but also the pages in which each part reside.



SCI - Protocol

SCI stands for the Serial Communication Interface (I think), which in this prototype is the interface between the PIC and the Host CPU (PC, Palm, ...). The PIC utilizes the MAX232 chip to convert TTL to Serial Port levels over a 3-wire link. See other documentation for any information I might have on that; and the Setup portion of this manual for the setup of the SCI port on the PIC.

The communication protocol between the PIC and the Host CPU has been designed specifically for the Bot application (as it stands now) and is defined below.

CURRENT COMMANDS

(PIC → PC)

(A) Analog/Digital

An	←	What value is A/D port n?
Anxx	→	Value of A/D port n is "xx".

(E) Error Status

Ea	←→	The error "a" occurred.

(I) Digital Input

I	←	What value is the Input port?
Ixx	→	Input port value is "xx".

(O) Digital Output

Oxx	←	Set Output port to value "xx".

Key:

"n" = a single decimal byte (0-9)
"a" = an alphanumeric byte (0-Z)
"x" = a single hex byte (0-F)
"xx" = a hex word (0-FF)

← = From PC to PC
→ = From PIC to PC
← → = Both ways

(P) PIC System Commands

PB	←→	Switch to Boot mode; Entering Boot mode.
PR	←→	Reset PIC; PIC has reset.
PZ	←→	Put PIC to Sleep; PIC is going to Sleep.

(S) Servo

Snxx	←	Set servo "n" (0-4) to "xx" degrees (0-B4) {0-180 in hex}.

Miscellaneous

OK	→	Valid command acknowledged.

FUTURE COMMANDS

(D) SPI Data

Dx	←	What's up with SPI device x? (Poll device for status.)
Dx"…"	←→	Send string "…" to SPI device x; Device x returns string "…".

(T) Timer

Tn	←	What is the value of Timer n?
Tn+	←	Activate Timer n.
Tn-	←	Deactivate Timer n.
Tnxx	→	The value of Timer n is xx (0=inactive, FF=Overflow).

(X) External Interrupts

Xn+	←	Activate external interrupt n.
Xn-	←	Deactivate external interrupt n.
Xn	→	External Interrupt n has occurred.

(M) Motor Control*

Mnx	←	Set motor n to x, where x-bits mean: 7 = direction (0=reverse, 1=forward) 6-0 = speed (0-127)

** Obsolete. Remove when code is permanently removed.*

Alternate PIC "Motor Controller"

This is an idea I had once coded into the main PIC but never used, as it took too many data ports to utilize. So I decided it might be best to save the idea for a future sister-PCB (interfaced via the SPI port) that only handles stepping motors. See example code for the original prototype for a fairly neat implementation that even incorporated a simplified ramp-up delay algorithm.

FUTURE SPECIALIZATIONS

I foresee that there is no way that most of the above commands will remain long past the prototyping stage. The reason for this is that the communication will naturally evolve to fit the usage of the chip and each specialization of the I/O pins.

For example, when a proximity detector is connected to pin X, none of the above will apply any longer. Data passing either way regarding pin X will report proximity data, based on timing and distances, and therefore will care less about the actual state of the pin at any given moment.

The same will go for servos, motors (if not given to another chip to handle), and a variety of other locomotion or sensor applications. This is actually a Good Thing, in that the chip will be handling a lot of the details (where it can) locally, and will only bother to send the data that is needed on the other end. Basically an embedded client-server setup.

Therefore, the above setup is meant to facilitate testing and give a coarse control until that time comes. So there.

A/D Setup Notes

Trying to get the A/D register to work as advertised, the following example finally fixes the problem by justifying-left the ADRESH register, using the 8 most-significant bits, and ignoring the least-significant ones in ADRESL.

SAMPLE CODE

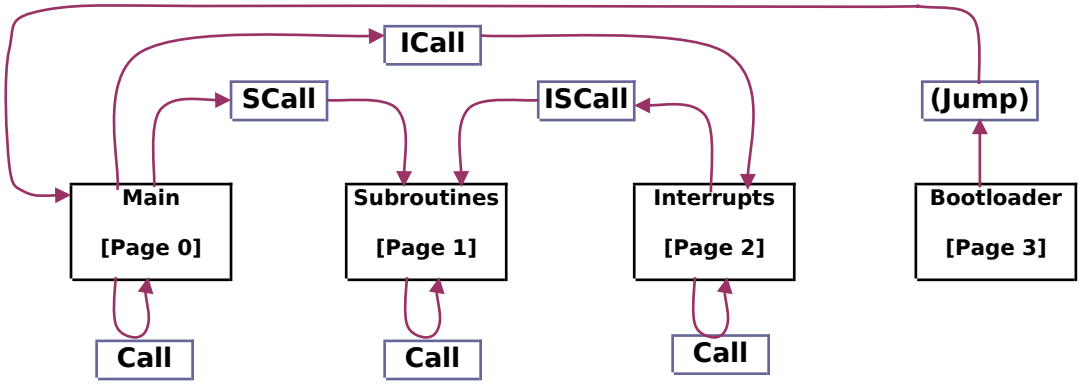
```
…
SetupAD
               bsf            STATUS,RP0             ;bank 1
               bcf            STATUS,RP1
               movlw          H'00'
               movwf          TRISC                  ;port C [7-0] output
               clrf           ADCON1                 ;left-justified, all AD inputs
               bcf            STATUS,RP0             ;bank 0
               movlw          B'01000001'            ;Fosc/8 [7-6], AD ch0 [S-3], AD ON
               movwf          ADCON0
Main
               call           AD_PortC
               goto           Main
AD_PortC                                             ;Acquisition time wait
                                                     ;(noncritical for this example)
               bsf            ADCON0,GO              ;Start AD conversion
WaitAD
               btfsc          ADCON0,GO              ;Conversion complete?
               goto           WaitAD                 ;No loop

               movf           ADRESH,W               ;Write AD result to PORT C
               movwf          PORTC                  ;for LED display
               return
               …
```

SETUP FOR SAMPLE

```
pin 11 & 32:          = 5VDC
pin 12 & 31           = 0VDC (ground)
pin 1                 = 5VDC
pin 2                 = Analog input (o - 5 VDC);  Center tap 20K var. resistor
pin 13 & 14           = 4MHz crystal with 18pf capacitors to 0VDC (ground)
pins 15 - 18, 23 - 26 = (Port C) output each to LED in series with 330 ohm to ground.
```

Program Paging

Since the PIC program memory is divided into pages and those pages are not easily modified on the fly without a serious expenditure of code, the following paging architecture was developed. Each box identifies the page each section of code resides in. Lines indicate which of the four types of subroutine calling macros are used to access one page from another page (believe it or not, this actually is necessary to keep things from getting confused when interrupts are flying around mucking things up):



Keep in mind that the PIC Stack is only 8-deep. Any calls that make more calls cannot go any deeper than 8 or the PIC will become seriously confused. The real danger here is when Main calls a Sub, and whilst in the midst of that call (or another Sub the first Sub called) an Interrupt occurs and then calls a Sub, which then calls a Sub … You get the picture.

CALL

Local calls within the same page require the standard RISK statement “Call”.

SCALL

Calls from Main (Page 0) to a subroutine (Page 1) need to save the PCLATH, otherwise a simple Return will restore the PCL without restoring the PCLATH, which handles what page we are in. SCall takes care of this.

ICALL

This call should only truly occur from the Interrupt Vector section at the beginning of the Main program. ICall does some important register saves before switching to Page 2, as there is no way of knowing at what point of the program (nor even what Page) we have interrupted *from*. And then a full restore of these registers before releasing the Interrupt and returning to wherever we got here from.

ISCALL

Interrupts calling subroutines. Basically saves the same important registers as ICall, but sets the PCLATH accordingly to get to Page 1 from Page 2. Beware of calling a subroutine that may already be called when the interrupt occurred (why Rx/Tx routines should always temporarily disable interrupts). Also, make sure that Interrupts do not share any variables that may be used in any of the subroutines, as changes will confuse the a subroutine we may have just Interrupted out of…

(BOOTLOAD) JUMP

Since the Bootloader simply does its thing and resets the PIC, all it needs to do is do a simple Jump to the beginning of Page 0 to run the Main code.

Shared Special Register Settings

00H,80H,100H,180H (IND. ADDR.)

(Setting or retrieving a value from this register automatically accesses the location pointed to in the FSR register)
[7 6 5 4 3 2 1 0]

01H,101H TMR0

(Accumulation register for Timer 0)
[7 6 5 4 3 2 1 0]

81H,181H OPTION_REG

7	_RBPU	1
6	INTEDG	1
5	T0CS	0
4	T0SE	0
3	PSA	0
2	PS2	0
1	PS1	0
0	PS0	0

02H,82H,102H,182H PCL

(Program Counter, least significant bits)
[7 6 5 4 3 2 1 0]

03H,83H,103H,183H STATUS

(Status Register, set as computations are made)

7	IRP	Indirect Bank Addressing
6	RP1	Bank Addressing
5	RP0	Bank Addressing
4	_TO	
3	_PD	
2	Z	Zero
1	DC	
0	C	Carry

04H,84H,104H,184H FSR

(Indirect data memory address pointer, used to indirectly address and access data via the Ind. Addr. register)
[7 6 5 4 3 2 1 0]

0AH,8AH,10AH,18AH PCLATH

(Sets upper five bits of Program Counter, including page addressing)
[X X X 4 3 2 1 0]

0BH,8BH,10BH,18BH INTCON

(Interrupt Controller)

7	GIE	0,1
6	PEIE	1
5	T0IE	0
4	INTE	1
3	RBIE	0
2	T0IF	?
1	INTF	?
0	RBIF	?

Special Register Settings

BANK 0

00h, 01h, 02h, 03h, 04h (shared)

[7 6 5 4 3 2 1 0]

05h PORTA
(Port A data values – in/out)
[X X 5 4 3 2 1 0]

06h PORTB
(Port B data values – in/out)
[7 6 5 4 3 2 1 0]

07h PORTC
(Port C data values – in/out)

08h PORTD
(Port D data values – in/out)
[7 6 5 4 3 2 1 0]

09h PORTE
(Port E data values – in/out)
[X X X X X 2 1 0]

0Ah, 0Bh (shared)

0Ch PIR1
(Peripheral Interrupt Flags)

7	PSPIF	?
6	ADIF	?
5	RCIF	?
4	TXIF	?
3	SSPIF	?
2	CCP1IF	?
1	TMR2IF	?
0	TMR1IF	?

0Dh PIR2
(Peripheral Interrupt Flags, continued)

7	X	0
6	0	0
5	X	0
4	EEIF	?
3	BCLIF	?
2	X	0
1	X	0
0	CCP2IF	?

0Eh TMR1L
(Timer 1 least significant byte register)
[7 6 5 4 3 2 1 0]

0Fh TMR1H
(Timer 1 most significant byte register)
[7 6 5 4 3 2 1 0]

10h T1CON
(Timer 1 Controller)

7	X	0	
6	X	0	
5	T1CKPS1		0
4	T1CKPS0		0
3	T1OSCEN	0	
2	T1SYNC	0	
1	TMR1CS		0
0	TMR1ON		0/1

11h TMR2
(Timer 2 accumulation register)
[7 6 5 4 3 2 1 0]

12h T2CON
(Timer 2 Controller)

7	X	0	
6	TOUTPS3	0	
5	TOUTPS2	0	
4	TOUTPS1	0	
3	TOUTPS0	0	
2	TMR2ON		0/1
1	T2CKPS1		0
0	T2CKPS0		0

13h SSPBUF
(Synchronous Serial Port Receive Buffer/Transmit Register)
[7 6 5 4 3 2 1 0]

14h SSPCON
(SSP Controller)

7	WCOL	0
6	SSPOV	0
5	SSPEN	0/1
4	CKP	0
3	SSPM3	0
2	SSPM2	0
1	SSPM1	0
0	SSPM0	0

15h CCPR1L
(Capture/Compare/PWM register 1, LSB)
[7 6 5 4 3 2 1 0]

16h CCPR1H
(Capture/Compare/PWM register 1, MSB)
[7 6 5 4 3 2 1 0]

17h CCP1CON
(CCP1 Controller)

7	X	0	
6	X	0	
5	CCP1X	0	
4	CCP1Y	0	
3	CCP1M3		0
2	CCP1M2		0
1	CCP1M1		0
0	CCP1M0		0

18h RCSTA
(SCI port, receiver setup)

7	SPEN	1
6	RX9	0
5	SREN	0
4	CREN	1
3	ADDEN	0
2	FERR	?
1	OERR	?
0	RX9D	?

19h TXREG
(SCI Transmit Data Register)
[7 6 5 4 3 2 1 0]

1Ah RCREG
(SCI Receive Data Register)
[7 6 5 4 3 2 1 0]

1Bh CCPR2L
(Capture/Compare/PWM register 2, LSB)
[7 6 5 4 3 2 1 0]

1Ch CCPR2H
(Capture/Compare/PWM register 2, MSB)
[7 6 5 4 3 2 1 0]

1Dh CCP2CON
(CCP2 Controller)

7	X	0	
6	X	0	
5	CCP2X	0	
4	CCP2Y	0	
3	CCP2M3		0
2	CCP2M2		0
1	CCP2M1		0
0	CCP2M0		0

1Eh ADRESH
(A/D Result Register, High Byte)

1Fh ADCON0
(A/D Controller)

7	ADCS1	0
6	ADCS0	0
5	CHS2	0
4	CHS1	0
3	CHS0	0
2	GO/DONE	0/1
1	X	0
0	ADON	0/1

BANK 1

80h, 81h, 82h, 83h, 84h (shared)

85h TRISA
(Port A Data Direction Register)

7	X	0
6	X	0
5	A5	1
4	A4	1
3	A3	1
2	A2	1
1	A1	1
0	A0	1

86h TRISB
(Port B Data Direction Register)

7	B7	1
6	B6	1
5	B5	1
4	B4	1
3	B3	1
2	B2	1
1	B1	1
0	B0	1

87h TRISC
(Port C Data Direction Register)

7	C7	1
6	C6	1
5	C5	0
4	C4	0
3	C3	0
2	C2	0
1	C1	0
0	C0	0

88h TRISD
(Port D Data Direction Register)

7	D7	0
6	D6	0
5	D5	0
4	D4	0
3	D3	0
2	D2	0
1	D1	0
0	D0	0

89h TRISE
(Port E Data Direction Register)

7	IBF	0
6	OBF	0
5	IBOV	0
4	PSPMODE	0
3	X	0
2	E2	0
1	E1	0
0	E0	0

8Ah, 8Bh (shared)

8Ch PIE1
(Peripheral Interrupt Enabler 1)

7	PSPIE	0
6	ADIE	0
5	RCIE	1
4	TXIE	0
3	SSPIE	0
2	CCP1IE	0
1	TMR2IE	0
0	TMR1IE	0

8Dh PIE2
(Peripheral Interrupt Enabler 2)

7	X	0
6	0	0
5	X	0
4	EEIE	0
3	BCLIE	0
2	X	0
1	X	0
0	CCP2IE	0

8Eh PCON
( )
[X X X X X X 1 0]

1	_POR	?
0	BOR	?

8Fh, 90h ---

91h SSPCON2
(SPI Controller 2)

7	GCEN	0
6	ACKSTAT	0
5	ACKDT	0
4	ACKEN	0
3	RCEN	0
2	PEN	0
1	RSEN	0
0	SEN	0

92h PR2
(Timer 2 Period Register)
[7 6 5 4 3 2 1 0]

93h SSPADD
(SPI Address Register)
[7 6 5 4 3 2 1 0]

94h SSPSTAT
(SPI Status Register)

7	SMP	0
6	CKE	0
5	D/A	0
4	P	0
3	S	0
2	R/_W	0
1	UA	0
0	BF	0

95h, 96h, 97h ---

98h TXSTA
(SCI Transmit Setup Register)

7	CSRC	0
6	TX9	0
5	TXEN	1
4	SYNC	0
3	X	0
2	BRGH	1
1	TRMT	1
0	TX9D	0

99h SPBRG
(SCI Baud Rate Generator Register)
[7 6 5 4 3 2 1 0]
d'129' = 2400 baud, when TXSTA:BRGH = 0
d'129' = 9600 baud, when TXSTA:BRGH = 1

9Ah, 9Bh, 9Ch, 9Dh ---

9Eh ADRESL
(A/D Result Register, Low Byte)
[7 6 5 4 3 2 1 0]

9Fh ADCON1
(A/D Controller Register)

7	ADFM	1
6	X	0
5	X	0
4	X	0
3	PCFG3	0
2	PCFG2	0
1	PCFG1	1
0	PCFG0	0

BANK 2

100h, 101h, 102h, 103h, 104h (shared)

105h ---

106h PORTB
(Same as 06h)

107h, 108h, 109h ---

10Ah, 10Bh (shared)

10Ch EEDATA
(EEPROM Data Register)
[7 6 5 4 3 2 1 0]

10Dh EEADR
(EEPROM Address Register)
[7 6 5 4 3 2 1 0]

10Eh EEDATH
(EEPROM Data Register, High Byte)
[X X 5 4 3 2 1 0]

10Fh EEADRH
(EEPROM Address Register, High Byte)
[X X X 4 3 2 1 0]

BANK 3

100h, 101h, 102h, 103h, 104h (shared)

105h ---

106h TRISB
(Same as 86h)

107h, 108h, 109h ---

18Ah, 18Bh (shared)

18Ch EECON1
(EEPROM Controller Register 1)

7	EEPGD	?
6	X	?
5	X	?
4	X	?
3	WRERR	?
2	WREN	?

1	WR	?	[7 6 5 4 3 2 1 0]
0	RD	?	

18Dh EECON2
(EEPROM Controller Register 2 – Not a physical register)

18Eh, 18Fh (reserved)

General Purpose Registers

(See source code for current general purpose register settings and use.)