Microchip PIC18F452 Core Hardware

CPU, Memory, Interrupts, and I/O Ports
PIC18F452 CPU

(Actually PIC18FXX2)
Harvard Architecture

- Separate instruction and data busses
- Instructions (opcodes)
  - 16-bit wide instruction bus
  - 21-bit instruction address: up to $2^{21} = 2M$ bytes FLASH program memory (where $1M = 1K \times 1K$)
- Data (operands & results)
  - 8-bit wide data bus
  - 12-bit data address: up to $2^{12} = 4K$ bytes data RAM
- 256 bytes EEPROM (not on diagram)

18F452 has

- 32K bytes FLASH (where $1K = 1024$)
- ~1.5K bytes RAM
18F4X2 Block Diagram

- 16-bit wide instruction bus (blue)
- 8-bit wide data bus (yellow)
ALU

- 8-bit ALU
- Working Register \texttt{WREG} behaves as an accumulator
- Can also source operands and write results anywhere in Data RAM, including SFRs \(\Rightarrow\) Register-to-Register architecture

- 8x8 single-cycle multiplier
- Hardware multiplier \(\sim70\) times faster than algorithm using additions
  \begin{align*}
  \text{MOV} \ F & \quad \text{ARG1, WREG} \\
  \text{MULWF} & \quad \text{ARG2}
  \end{align*}
- See code in Data Sheet for 16x16 multiply
- Also files on MxLab Server
Programmer’s Model – “Core” SFRs
# Status Register

## Status Register

### STATUS: Status (Flags) Register

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO/BO Reset</td>
<td>U-0</td>
<td>U-0</td>
<td>U-0</td>
<td>N</td>
<td>OV</td>
<td>Z</td>
<td>DC</td>
<td>C</td>
</tr>
<tr>
<td>MCLR Reset</td>
<td>U-0</td>
<td>U-0</td>
<td>U-0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

- **N**: Negative bit.
  - 1: Result was negative
  - 0: Result was positive

- **OV**: Overflow bit.
  - 1: Overflow of bit 7 occurred during signed arithmetic operation
  - 0: No overflow occurred

- **Z**: Zero bit.
  - 1: Result of operation was zero
  - 0: Result of operation was not zero

- **DC**: Digit Carry bit (for BCD).
  - 1: Carry-out from the 4th low order bit of result occurred
  - 0: No carry from the 4th low order bit of result occurred

- **C**: Carry bit (for binary).
  - 1: Carry-out from the MSB of result occurred on addition (OR No borrow occurred on subtraction)
  - 0: Carry from the MSB of result did not occur on addition (OR Borrow occurred on subtraction)

Note: **x** = unknown  
**u** = unchanged
Memory

Program Memory
Data Memory
EEPROM
Program Memory
Program Memory

- 21-bit PC (Program Counter) in RAM can address $2^{21} = 2$MB
- 18F452 has 32KB of 8-bit FLASH
- Can store 16K instructions

- Reset address is 0x000000

- High priority interrupt address is 0x000008 (not a vector!)
- Low priority interrupt address is 0x000018 (not a vector!)

- Unimplemented memory reads zero – executes as NOP
Byte Storage

- Program words are 16-bit, stored in 2 successive byte locations
- lsb of the PC always reads 0, ⇒ PC increments by 2

– but –

- Fixed data (e.g. look-up tables) can be stored 2 bytes per program word – see later
Return Address Stack

- Dedicated 31 deep x 21-bit wide hardware stack – calls and interrupts can nest 31 levels deep
- PC is pushed on CALL, RCALL or interrupt response
- PC is popped on RETURN, RETURNW or RETFIE
- Stack pointer pre-increments on push, post-decrements on pop
- TOS register contains contents of stack entry pointed to by STKPTR
- STKPTR contains STKFUL, STKUNF (underflow) bits
- Will reset on STKUNF, (address 0x0000 is popped) can reset on STKFUL if CONFIG4L<STVREN> is set
## STKPTR Register

### STKPTR: Return Stack Pointer Register

<table>
<thead>
<tr>
<th>Bit Names</th>
<th>0xFFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Names</td>
<td>R/W</td>
</tr>
<tr>
<td>Bit 7</td>
<td>STKFUL</td>
</tr>
<tr>
<td>PO/BO Reset</td>
<td>0</td>
</tr>
<tr>
<td>MCLR Reset</td>
<td>0</td>
</tr>
</tbody>
</table>

**bit 7**  
**STKFUL**: Stack Full Flag bit.  
*Note*: Can only be cleared by user software or a POR.  
- 1: Stack is full (31 entries) or has overflowed.  
- 0: Stack has not become full or has overflowed.

**bit 6**  
**STKUNF**: Stack Underflow Flag bit.  
*Note*: Can only be cleared by user software or a POR.  
- 1: Stack underflow has occurred.  
- 0: Stack has not underflowed.

**bit 5**  
Not implemented – reads as 0.

**bit 4-0**  
**STKPTR4**: Stack pointer value bits.  
This 5-bit field contains the hardware stack pointer value, between 0 and 31 inclusive.
Program Counter

- The 21-bit PC is contained in the three registers
  \[ \text{PCU : PCH : PCL} \]

- Only \text{PCL} is directly readable and writable

- \text{PCU and PCH} can be updated through the latch registers \text{PCLATU, PCLATH}
  - Operations that write \text{PCL} also copy \text{PCLATU} \to \text{PCU} and \text{PCLATH} \to \text{PCH}
  - Operations that read \text{PCL} also copy \text{PCU} \to \text{PCLATU} and \text{PCH} \to \text{PCLATH}

- \text{CALL, RCALL, GOTO and branch instructions directly load PCU : PCH : PCL}
Clocking & Instruction Cycle

- Four internal phase clocks Q1 to Q4 derived from clock input OSC1 of period $T_{OSC}$
- Fetch instruction cycle (instruction bus is active):
  - Q1: increment PC
  - Q2: write instruction address to program memory
  - Q3: fetch opcode on instruction bus
  - Q4: latch into instruction register
Clocking & Instruction Cycle

- Execute instruction cycle (data bus is active)
- Cycle varies – see instruction set. Typically:
  - Q1: decode instruction in instruction register
  - Q2: fetch operand from data memory
  - Q3: execute
  - Q4: write result to destination in data memory
- So, one instruction executes every \( T_{CY} = 4 \times T_{OSC} \) ⇒ pipelining

All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is “flushed” from the pipeline while the new instruction is being fetched and then executed.
Two-word Instructions

- There are four 2-word instructions:
  - MOVFF (Move from f to f)
  - CALL (Subroutine Call)
  - GOTO (Unconditional Branch)
  - LFSR (Load FSR)

- Each causes a *pipeline stall*, as *two* instruction fetch cycles are needed before an execute cycle can occur.
Data Memory
Data Memory

- Data memory is Static RAM (SRAM)
- 12-bit data address, can address $2^{12} = 4096$ bytes = 4K

- Data memory is divided into 16 *banks* (pages), each containing 256 bytes (8-bit addressable within a bank)

- 18F452 has 7 banks implemented: 0, 1, 2, 3, 4, 5 & 15, giving 1,664B of addressable data memory (including SFRs)

- A bank is selected using a 4-bit field in the Bank Select Register: $\text{BSR}<3:0>$
Data Memory Contents

Data memory contains

- **Special Function Registers (SFRs)**
  - Some SFRs have ‘core’ functions – WREG, STATUS, etc
  - Most SFRs used to monitor and command the peripherals
  - SFRs begin at the highest address in Bank 15 (0x000FFF) and extend downwards to 0x000F80
  - Much more detail later...

- **General Purpose Registers (GPRs) – the “Register File”**
  - Available to the programmer (or compiler) for general use
  - Store variables, intermediate results, etc.
Data Memory Addressing

Data memory can be addressed using either

- **Direct Addressing** ⇒ Operand address(es) are embedded in the opcode, and *fixed at run time*
  - PIC18 has three direct addressing methods:
    - Banked
    - Access Bank
    - Address in Opcode

- **Indirect Addressing** ⇒ Operand address(es) are separate from the opcode and *determined at run time*
  - Indirect addressing is *using a pointer* to refer to data
  - PIC18 has one indirect addressing method (although it has many flexible options)
Data Memory Map

Direct Addressing (Banked)
- A bank selected by 4 BSR bits
- A 16-bit opcode contains (only) one 8-bit address \( \Rightarrow \) full 12 bits

Direct Addressing (Access Bank)
Access Bank is comprised of
- Upper 128 bytes in Bank 15 (all of the SFRs)
- Lower 128 bytes in Bank 0 (a few of the GPRs)

- One instruction bit (called ‘a’) selects use of the Access Bank \( \Rightarrow \) 8 bit address

When \( a = 0 \), the BSR is ignored and the Access Bank is used. The first 128 bytes are General Purpose RAM (from Bank 0). The second 128 bytes are Special Function Registers (from Bank 15).

When \( a = 1 \), the BSR is used to specify the RAM location that the instruction uses.
Direct Addressing: Banked

- 4-bit Bank Select Register (BSR) chooses 1 of 16 banks
- 8-bit address from opcode selects one of the 256 registers in that bank
- $4 + 8 = 12$-bit address selects one from $2^{12} = 4096$ registers

**Note**: For register file map detail, see Table 4-1.
Direct Addressing: Access Bank

- **BSR is not used (implicitly 0 or 15)**

- **8-bit address selects one of the 256 Access Bank registers**

- **Address mapping**
  - 0x00-0x7F are in Bank 0
  - 0x80-0xFF are in Bank 15

- **High half of Bank 15 are the SFRs**

*Note 1:* For register file map detail, see Table 4-1.

*Note 2:* The access bit of the instruction can be used to force an override of the selected bank (BSR<3:0>) to the registers of the Access Bank.
Direct Addressing: Address in Opcode

- BSR is not used

- Full 12-bit address is contained in the (16-bit) opcode

- Only used in four (double length) instructions:
  - MOVFF
  - CALL
  - GOTO
  - LFSR

**Note**
1. For register file map detail, see Table 4-1.
2. The MOVFF, CALL, GOTO, LFSR instructions embed entire 12-bit addresses in the instruction.
Example 1: Direct Data Memory Addressing

Many instructions have the form

```
MNEMONIC f[, d[, a]] ; [] means optional
```

where

- **f** is an 8-bit file register (0x00 to 0xFF)
- **d** is the result destination bit: WREG (d = 0) or file register (d = 1; default)
- **a** signifies access bank (a = 0) or banked addressing (a = 1; default)
- Operands **a** and **d** are optional and both default to 1 if omitted

**Direct addressing example (banked)**

```
MOVLB   02          ; set BSR to Bank 2
ADDWF   H’55’, 0, 1 ; ADD WREG to contents of address 0x55 (f=55)
                     ; in bank 2 (a=1), result to WREG (d=0)
```

The operand is the contents of the general purpose data memory address 0x255 in Bank 2
Example 2: Direct Data Memory Addressing

Again:

```
MNEMONIC f[, d[, a]] ; [] means optional
```

where

- **f** is an 8-bit file register (0x00 to 0xFF)
- **d** is the result destination: WREG (d = 0) or file register (d = 1; default)
- **a** signifies access bank (a = 0) or banked addressing (a = 1; default)
- Operands **a** and **d** are optional and both default to 1 if omitted

- **Direct addressing example (using Access Bank)**

```
; MOVLM not required as BSR is ignored
ADDWF 0x83, 0, 0 ; ADD WREG to contents of address 0x83 (f=83)
                  ; in access bank (a=0), result to WREG (d=0)
```

The operand is the contents of general purpose data memory address 0xF83 in the Access Bank – this is PORTD
Best Practice: Use Symbolic Names

The two previous examples are hard to read. Symbolic names for SFRs and bits are predefined in `P18F452.inc`

```assembly
#include "p18F452.inc" ; defines W, BANKED, PORTD, etc
F EQU 1 ; counterpart of W (not defined in p18F452.inc)

MOVLB 02 ; set BSR to Bank 2
ADDWF 0x55, W, BANKED ; add WREG to contents of address 0x55
; in bank 2 (a=1), result to WREG (d=0)

ADDWF PORTD, F, ACCESS
```
Useful Aside: Variables in RAM

- Just as symbolic names are defined for SFRs in `P18F452.inc`, you can define symbolic names for locations in general purpose RAM – your own variables.

- For example, to create a variable in Access RAM:

```
UDATA_ACS ; declare an uninitialised data section in access RAM
myVar  res 1 ; reserve one byte for variable “myVar”

CODE ; declare a code section

Startup:
  MOVLW  0xAA ; move literal 0xAA to WREG
  MOVWF  myVar, ACCESS ; initialise myVar
  ; etc...
```
Indirect Data Memory Addressing

- Indirect Addressing ⇔ Operand address(es) are separate from the opcode and determined at run time
  - Indirect addressing is the same as using a pointer to refer to data
  - PIC18 has one indirect addressing method (although it has many flexible options)

- Requires the use of one of the three File Select Registers FSRn, and a corresponding Indirect File Operand INDFn to hold the full 12-bit address of an operand
Indirect Data Memory Addressing

- Use any of 3 FSRs (File Select Registers) FSR0, FSR1, FSR2 to store a 12-bit pointer to an operand
- Any reference to an **INDF** register (Indirect File Operand) in an instruction actually operates on the operand pointed to by the corresponding FSR

```assembly
LFSR FSR0, ADDR    ; load FSR0 with 12-bit address ADDR
MOVWF INDF0        ; move value in WREG to file register
                   ; pointed to by FSR0
```

**INDF0** is an example of an indirect register file operand – it represents the data at the address that is stored in **FSR0**

- Cannot directly read or write any of the indirect file operands – only the FSRs are physical registers
Indirect Data Memory Addressing

File Address = access of an indirect addressing register

$2^{12} = 4K = 4096$

Register file 'F' ignored
Indirect Data Memory Addressing

- Pointer in a FSR can be post-decremented, pre- or post-incremented, etc. ⇒ step through tables etc.

- Examples

  ; All of the following move a byte from WREG to the file register
  ; pointed to by an FSR – they have different effects on the pointer

  MOVWF  INDF0, ACCESS       ; don’t change the pointer value FSR0
  MOVWF  POSTDEC1, ACCESS   ; post-decrement FSR1
  MOVWF  POSTINC2, ACCESS   ; post-increment FSR2
  MOVWF  PREINC0, ACCESS    ; pre-increment FSR0
  MOVWF  PLUSW1, ACCESS     ; offset value in FSR1 by value in WREG –
                           ; useful for indexing into a table
Reading/Writing FLASH Program Memory
Reading/Writing FLASH Program Memory

Used for

- Programming the device
  (writing executable program to FLASH)
- Modifying Program Memory
- Transferring data between
  Program Memory and Data Memory
Reading/Writing FLASH Program Memory

- During normal program execution, FLASH can be:
  - Read – one byte at a time
  - Written – one 8-byte block at a time – see data sheet
  - Erased – one 64-byte block at a time – see data sheet

- Code cannot be fetched during FLASH read/write

- Reading FLASH needed to move constants from FLASH to data memory – operands can not be in FLASH

- Writing to FLASH can alter program at run-time (!!)
TBLRD/TBLWT: Table Read/Write

- Operands cannot be in Program Memory (FLASH)
- TBLRD, TBLWT instructions move data between Program Memory (FLASH) and Data Memory

- All reads and writes are byte-wide
- Data move through an 8-bit register, TABLAT (the Table Latch register)

- The 21-bit Table Pointer in TBLPTR addresses (points to) a byte in Program Memory
- It is contained in the three registers TBLPRTU : TBLPTRH : TBLPTRL
Table Read: TBLRD *

Table Read copies a byte from Program Memory to Data Memory

- Write 21-bit table pointer value to TBLPRTU, TBLPTRH, TBLPTRL
- Executing TBLRD * instruction moves byte from Program Memory location pointed at by TBLPTR to TABLAT register in data space
- Move TABLAT into (say) WREG

Note 1: Table Pointer points to a byte in program memory.
Table Write: TBLWT *

Table Write copies a byte from Data Memory to Program Memory

- Write 21-bit table pointer value to TBLPTRU, TBLPTRH, TBLPTRL
- Move (say) WREG into the Table Latch Register TABLAT
- Executing TBLW * instruction moves byte from TABLAT to Program Memory location pointed at by TBLPTR
- See the low, high and upper assembler macros

Note 1: Table Pointer actually points to one of eight holding registers, the address of which is determined by TBLPTRL<2:0>. The process for physically writing data to the Program Memory Array is discussed in Section 5.5.
TBLRD/TBLWT: Table Read/Write

- TBLPTR can be post-decremented, pre- or post-incremented, etc. ⇒ step through tables, etc.
- The * represents the pointer (21-bit address) in TBLPTR
- The 22nd bit is set to enable access to the device config bits.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect on TBLPTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBLRD *</td>
<td>None</td>
</tr>
<tr>
<td>TBLWT *</td>
<td>None</td>
</tr>
<tr>
<td>TBLRD *+</td>
<td>Post-increment</td>
</tr>
<tr>
<td>TBLWT *+</td>
<td>Post-increment</td>
</tr>
<tr>
<td>TBLRD *-</td>
<td>Post-decrement</td>
</tr>
<tr>
<td>TBLWT *-</td>
<td>Post-decrement</td>
</tr>
<tr>
<td>TBLRD *+</td>
<td>Pre-increment</td>
</tr>
<tr>
<td>TBLWT *+</td>
<td>Pre-increment</td>
</tr>
</tbody>
</table>

TBLRD * ; read value pointed to into TABLAT, don’t change TBLPTR value
TBLRD *+ ; read value pointed to into TABLAT, then increment TBLPTR value
TBLRD *- ; read value pointed to into TABLAT, then decrement TBLPTR value
TBLWR *+ ; pre-increment TBLPTR, then write TABLAT to address pointed at
Table Read/Write Registers

- Four associated Special Function Registers:
  - **EECON1**: selects FLASH/EEPROM access, erase, EEPROM read/write, etc.
  - **EECON2**: not a physical register
  - **TABLAT**: 1-byte Table Latch register
  - **TBLPTR**: 3-byte Table Pointer register

<table>
<thead>
<tr>
<th>Bit Names</th>
<th>p. 66</th>
<th>R/W</th>
<th>R/W</th>
<th>U-0</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEPGD</td>
<td></td>
<td>R/W</td>
<td>R/W</td>
<td>U-0</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>CFGS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO/BO Reset</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>bit 0</td>
</tr>
<tr>
<td>MCLR Reset</td>
<td>u</td>
<td>u</td>
<td>-</td>
<td>0</td>
<td>u</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Data EEPROM Memory
Data EEPROM Memory

- EEPROM = Electrically-Erasable Programmable Read-only Memory
- Retains data even when powered-off
- Use to store
  - Calibration constants
  - Save context on power fail, etc

- PIC18F452 has 256 bytes of EEPROM – 8-bit address
- More EEPROM easily added via SPI or I²C expansion
Data EEPROM Registers

- Four associated Special Function Registers:
  - **EECON1**: selects FLASH/EEPROM access, erase, EEPROM read/write, etc.
  - **EECON2**: not a physical register (but used in EEPROM write cycle)
  - **EEDATA**: 8-bit data read/to be written
  - **EEADR**: 8-bit address of data read/to be written

<table>
<thead>
<tr>
<th>Bit Names</th>
<th>R/W</th>
<th>R/W</th>
<th>U-0</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>p. 66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Names</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEPGD CFGS</td>
<td>x</td>
<td>x</td>
<td>–</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MCLR Reset</td>
<td>u</td>
<td>u</td>
<td>–</td>
<td>0</td>
<td>u</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Data EEPROM Read

EEPROM Read
- Write pointer value (address) to EEADR
- Clear EECON1<EEPGD> (access EEPROM)
- Set EECON1<RD> (initiate read)

- Data is available in EEDATA the very next cycle, so
- Read that data...
Data EEPROM Write

- EEPROM Write – see Example 6-2 in data sheet
  - Write pointer value (address) to EEADR
  - Write data value to EEDATA
  - Clear EECON1<EEPGD> (access EEPROM)
  - Set EECON1<WREN> (enable EEPROM writes)
  - Clear INTCON<GIE> (disable interrupts)
  - Write 0x55 to EECON2 (required sequence)
  - Write 0xAA to EECON2 (required sequence)
  - Set EECON1<WR> (initiate write - required sequence)
  - Set INTCON<GIE> (enable interrupts)
  - Clear EECON1<WREN> (disable EEPROM writes)

- Good practise: always read back and verify each EEPROM write
Configuration Registers
Configuration Registers

- Thirteen registers in address space 0x300000 to 0x3FFFFFF
- Configure 18F452 hardware \textit{when the processor comes out of reset}
  - Oscillator (clock) configuration
  - Reset / Brownout behaviour
  - CCP2 Mux – RC1 / RB3
  - Low voltage programming / In-circuit debug
  - Watchdog timer
  - Code protection
  - Device ID

- If any of the configuration bits are changed the processor \textbf{must be reset} for the change(s) to take effect.
Setting the Configuration Registers

- In code (best practice) – see example files on server
  ```
  #include configReg.inc ; in assembler
  #include "configReg.h" // in C
  ```

- In MPLAB X: Window > PIC Memory Views > Configuration Bits
  - Set options using menus in Configuration Bits memory view
  - Generate source code to output window
  - Paste code in source file.

- See Datasheet chapter 19
Interrupt System
Interrupt System

- Multiple (18) interrupt sources

- Two interrupt response priorities: high and low
  - High priority response preempts any low priority interrupt service routine

- All interrupts execute through two fixed addresses:
  - 0x0008: High priority interrupt address
  - 0x0018: Low priority interrupt address

- ‘Compatibility mode’ (with PIC16XX) – no priorities
  - 0x0008: Interrupt address
Interrupt Sources

- Up to four external pins
  - INT0 is RB<0>
  - INT1 is RB<1>
  - INT2 is RB<2>
  - Change on RB<7:3> (RB is PORTB)
- INT0, INT1, INT2 are edge triggered – selectable as rising or falling
- Fourteen internal events

<table>
<thead>
<tr>
<th>External Pins (all on PORTB)</th>
<th>INT0 External Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INT1 External Interrupt</td>
</tr>
<tr>
<td></td>
<td>INT2 External Interrupt</td>
</tr>
<tr>
<td></td>
<td>PortB Change Detect</td>
</tr>
<tr>
<td>Timer Interrupts</td>
<td>TMR0 Overflow</td>
</tr>
<tr>
<td></td>
<td>TMR1 Overflow</td>
</tr>
<tr>
<td></td>
<td>TMR2 to PR2 Match</td>
</tr>
<tr>
<td></td>
<td>TMR3 Overflow</td>
</tr>
<tr>
<td>Capture/Compare/PWM</td>
<td>CCP1 Interrupt</td>
</tr>
<tr>
<td></td>
<td>CCP2 Interrupt</td>
</tr>
<tr>
<td>Analog Inputs</td>
<td>A/D Conversion Done</td>
</tr>
<tr>
<td>USART Port</td>
<td>USART Rx Buffer Full</td>
</tr>
<tr>
<td></td>
<td>USART Tx Buffer Empty</td>
</tr>
<tr>
<td>Synchronous Serial</td>
<td>MSSP Interrupt</td>
</tr>
<tr>
<td></td>
<td>I²C Bus Collision</td>
</tr>
<tr>
<td>Parallel</td>
<td>PSP Read/Write</td>
</tr>
<tr>
<td>Power Supply</td>
<td>Low Voltage Detect</td>
</tr>
<tr>
<td>Memory</td>
<td>EEPROM/FLASH Read/Write</td>
</tr>
</tbody>
</table>
Interrupt Registers

- Ten associated Special Function Registers:

  - **RCON<IPEN>** (bit 7): enable high/low priority feature
  - **INTCON, INTCON2, INTCON3**: these control the “core” interrupt functions - enables, priorities, flags
  - **PIR1, PIR2**: Peripheral Interrupt Request flags
  - **PIE1, PIE2**: Peripheral Interrupt Enable flags
  - **IPR1, IPR2**: Interrupt Priority flags control the “peripheral” interrupt priorities
Enabling Interrupt Response

Generally, for interrupt response to occur, requires:

- Global Interrupt Enabled TRUE
- Local (individual) Interrupt Enabled \texttt{xxxxIE} TRUE
- Individual interrupt \texttt{XXXX} requested – sets \texttt{xxxxIF}

- No other interrupt current at higher priority

- Then, control will pass to either
  - 0x0008 or 0x0018
  which must pass control to the user’s ISR – usually using a \texttt{goto} statement
Interrupt Control: Priority Mode

- Interrupt source can be programmed as either High or Low Priority depending on a XXXXIP bit

- High-Priority Global Int. Enable bit: INTCON<GIEH> (bit 7)
  - Enables/disables all High Priority interrupt responses

- Low-Priority Global Int. Enable bit: INTCON<GIEL> (bit 6)
  - Enables/disables all Low Priority interrupt responses

- Interrupt Enable (mask) bits: Names like XXXXIE
  - Enable/disable response to individual interrupt requests

- No difference between “Core” and “Peripheral” interrupt sources
Interrupt Control: High Priority

- Priority Enable bit: 
  \( \text{IPEN} \equiv \text{RCON<7>} \equiv 1 \)

- Global Interrupt Enable High Priority bit: 
  \( \text{GIEH} \equiv \text{INTCON<7>} \)
    - Enables/disables all high-priority interrupt responses

- Local Interrupt Enable bits \( \text{xxxxIE} \) enable/disable response to individual interrupt requests

- Interrupt Priority bits \( \text{xxxIP} \) must be 1 to set High Priority

- An active High Priority interrupt inhibits all Low Priority interrupts

Note: interrupt priority bits \( \text{xxxxIP} = 1 \)
Interrupt Control: Low Priority

- **Priority Enable bit:**
  \[ \text{IPEN} \equiv \text{RCON}<7> \equiv 1 \]

- **Global Interrupt Enable Low Priority bit:**
  \[ \text{GIEL} \equiv \text{INTCON}<6> \]
  - Enables/disables all low-priority interrupt responses

- **Local Interrupt Enable bits XXXXIE**
  enable/disable response to individual interrupt requests

- **Interrupt Priority bits XXXIP** must be 0 to set Low Priority

- All Low Priority interrupts are suppressed by an active High Priority interrupt
Interrupt Response Cycle: Priority Mode

When an interrupt request is responded to

- GIEH or GIEL is cleared, disabling further interrupts
- Return address is pushed onto the stack
- PC is loaded with a specific address: either 0x0008 (High) or 0x0018 (Low)

- User ISR must do everything else – see following

- User ISR \textit{must} end in
  - RETFIE instruction – this causes hardware to pop the return address and set GIEH or GIEL as appropriate, re-enabling interrupts
Configuring the Interrupt System

- Your code should generally do the following

```c
clear INTCON<GIEH> ; disable interrupt response
clear INTCON<GIEL>
set RCON<IPEN> ; enable priority mode
IPR1 = xxxxxxxxxx ; set interrupt priorities
IPR2 = xxxxxxxxxx
PIE1 = xxxxxxxxxx ; enable individual interrupts
PIE2 = xxxxxxxxxx
PIF1 = 00000000 ; clear all pending flag bits
PIF2 = 00000000
set INTCON<GIEH> ; enable interrupt response
set INTCON<GIEL> ; as appropriate
```

- Note that there are also some priority bits xxxxIP, enable bits xxxxIE and flag bits xxxxF in INTCON, INTCON2 and INTCON3
## Summary of Interrupt Control Bits

<table>
<thead>
<tr>
<th>Source</th>
<th>Priority Bit</th>
<th>Enable Bit</th>
<th>Flag Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT0 External Interrupt</td>
<td>High Priority Only</td>
<td>INTCON&lt;INT0IE&gt;</td>
<td>INTCON&lt;INT0IF&gt;</td>
</tr>
<tr>
<td>INT1 External Interrupt</td>
<td>INTCON3&lt;INT1IP&gt;</td>
<td>INTCON3&lt;INT1IE&gt;</td>
<td>INTCON3&lt;INT1IF&gt;</td>
</tr>
<tr>
<td>INT2 External Interrupt</td>
<td>INTCON3&lt;INT2IP&gt;</td>
<td>INTCON3&lt;INT2IE&gt;</td>
<td>INTCON3&lt;INT2IF&gt;</td>
</tr>
<tr>
<td>Change on PortB</td>
<td>INTCON2&lt;RBIP&gt;</td>
<td>INTCON&lt;RBIE&gt;</td>
<td>INTCON&lt;RBIF&gt;</td>
</tr>
<tr>
<td>TMR0 Overflow</td>
<td>INTCON2&lt;TMR0IP&gt;</td>
<td>INTCON&lt;TMR0IE&gt;</td>
<td>INTCON&lt;TMR0IF&gt;</td>
</tr>
<tr>
<td>TMR1 Overflow</td>
<td>IPR1&lt;TMR1IP&gt;</td>
<td>PIR1&lt;TMR1IE&gt;</td>
<td>PIE1&lt;TMR1IF&gt;</td>
</tr>
<tr>
<td>TMR2 to PR2 Match</td>
<td>IPR1&lt;TMR2IP&gt;</td>
<td>PIR1&lt;TMR2IE&gt;</td>
<td>PIE1&lt;TMR2IF&gt;</td>
</tr>
<tr>
<td>TMR3 Overflow</td>
<td>IPR2&lt;TMR3IP&gt;</td>
<td>PIR2&lt;TMR3IE&gt;</td>
<td>PIE2&lt;TMR3IF&gt;</td>
</tr>
<tr>
<td>CCP1 Interrupt</td>
<td>IPR1&lt;CCP1IP&gt;</td>
<td>PIR1&lt;CCP1IE&gt;</td>
<td>PIE1&lt;CCP1IF&gt;</td>
</tr>
<tr>
<td>CCP2 Interrupt</td>
<td>IPR2&lt;CCP2IP&gt;</td>
<td>PIR2&lt;CCP2IE&gt;</td>
<td>PIE2&lt;CCP2IF&gt;</td>
</tr>
<tr>
<td>A/D Conversion Done</td>
<td>IPR1&lt;ADIP&gt;</td>
<td>PIR1&lt;ADIE&gt;</td>
<td>PIE1&lt;ADIF&gt;</td>
</tr>
<tr>
<td>UART Rc Buffer Full</td>
<td>IPR1&lt;RCIP&gt;</td>
<td>PIR1&lt;RCIE&gt;</td>
<td>PIE1&lt;RCIF&gt;</td>
</tr>
<tr>
<td>UART Tx Buffer Empty</td>
<td>IPR1&lt;TXIP&gt;</td>
<td>PIR1&lt;TXIE&gt;</td>
<td>PIE1&lt;TXIF&gt;</td>
</tr>
<tr>
<td>MSSP Interrupt</td>
<td>IPR1&lt;SSPIP&gt;</td>
<td>PIR1&lt;SSPIE&gt;</td>
<td>PIE1&lt;SSPIF&gt;</td>
</tr>
<tr>
<td>PSP Read/Write</td>
<td>IPR1&lt;PSPIPR&gt;</td>
<td>PIR1&lt;PSPIE&gt;</td>
<td>PIE1&lt;PSPIF&gt;</td>
</tr>
<tr>
<td>Low Voltage Detect</td>
<td>IPR2&lt;LVDIP&gt;</td>
<td>PIR2&lt;LVDIE&gt;</td>
<td>PIE2&lt;LVDIF&gt;</td>
</tr>
<tr>
<td>Bus Collision</td>
<td>IPR2&lt;BCLIP&gt;</td>
<td>PIR2&lt;BCLIE&gt;</td>
<td>PIE2&lt;BCLIF&gt;</td>
</tr>
</tbody>
</table>
Best Practice: Configuring Interrupts

- Subroutine for configuring the interrupt for a hardware module should
  - Set priority (if applicable)
  - Unmask interrupt for that module
- It should not
  - Assume that any config registers are in their reset states
  - Make any assumptions regarding the configuration of other interrupts, nor change their state
  - Enable/disable interrupt priorities
  - Enable/disable global interrupt response
  - (Both should be done in mainline code where clearly visible)
Best Practice: Configuring Interrupts

- Example: configuring the USART transmit (Tx) interrupt

```assembly
bcf INTCON, GIEH ; disable interrupts
bcf INTCON, GIEL
bsf RCON, IPEN ; enable priorities
call setupTxInt ; subroutine call
bsf INTCON, GIEL ; enable low interrupt

setupTxInt:
  bcf IPR1, TXIP ; low priority
  bsf PIR1, TXIF ; unmask Tx interrupt
return
```
User Interrupt Service Routines

- Only two interrupts ⇒ only two ISRs
- Inside an ISR, user code must

```c
save STATUS ; save context
save BSR
save WREG
if ( more than 1 source may have requested interrupt ) {
    for ( each interrupt used ) {
        if ( xxxxIF and xxxxIE ) {
            service interrupt xxxx
            clear xxxxI ; often needed in software
        }
    }
}
restore WREG ; restore context
restore BSR
restore STATUS ; last as REG ops. change flags!!
RETFIE
```
Installing Interrupt Service Routines

- When response to an interrupt request occurs, PC is loaded with either 0x08 or 0x18
- User must arrange to place appropriate executable code at one or both of these addresses. Good practice to *always* have at least a null ISR that just returns
- Installation depends on language (and software tools):

```assembly
; in assembler
goToLowIsr: CODE 0x0018
    goto lowPriorityIsr

lowPriorityIsr:
    ; do stuff as in previous slide
    retfie
```
Installing Interrupt Service Routines

// in C
#pragma code highISR = 0x08
void goToHighISR( void )
{
    _asm
        goto highPriorityISR
    _endasm
}

#pragma interrupt highPriorityIsr    // or interruptlow
void highPriorityIsr( void )
{ // do stuff }
FAST Interrupt Return

- Fast Return Stack (Shadow Registers)
  - Three byte-wide registers STATUS, WREGS, BSRS
  - Not directly readable or writable
  - Always loaded with STATUS, WREG, BSR in hardware during interrupt response

- User ISR *can optionally* end in RETFEI FAST – the FAST causes BSR, WREG, STATUS to be re-loaded by popping from the fast return stack

- Beware!! Will fail if FAST return is used in both High and Low Priority ISRs enabled ⇒ values from High Priority interrupt overwrite values from Low Priority interrupt

- If not used for interrupts, shadow registers can be used in a CALL FAST – then *must* use RETURN FAST
Critical Regions

- A Critical Region is a block of code (assembler or HLL) that must not be interrupted.
- Often occurs when:
  - variable is accessed in both the main code and in an ISR and
  - The access (read or write) is non-atomic (an atomic action is guaranteed to complete).

- Solution 1 (For HLL or asm)
  - Disable interrupts
  - Access the variable
  - Re-enable interrupts

- Solution 2 (For asm only)
  - Access variable using only instructions that do atomic read-modify-write

- Peatman gives an example
Critical Regions - Example

- **Poor practice**
  
  ```assembly
  MOVF PORTB, W, ACCESS ; copy PortB to WReg
  ANDLW B'11111001' ; AND literal with Wreg
  MOVWF PORTB, ACCESS ; copy result to PortB
  ```

- **Good practice**
  
  ```assembly
  MOVFLW B'11111001' ; copy literal to WReg
  ANDWF PORTB, F, ACCESS ; PortB = PortB AND WReg
  ; this is a Read-Modify-Write op.
  ```
Interrupt Control: Compatibility Mode

- Compatibility ⇒ Compatible with PIC16XXX
- Priority Enable bit: IPEN ≡ RCON<7> = 0 (priority disabled)
- Interrupt address is 0x08 in compatibility mode

- “Core” Interrupt Sources
  - Five core sources: INT0, INT1, INT2, RB change, TMR0
  - Controlled through INTCON, INTCON1, INTCON2
  - Enabled/disabled only by GIE ⇒ INTCON<7>

- “Peripheral” Interrupt Sources
  - The other 13 sources are peripheral interrupts
  - Controlled through PIE1, PIE2
  - Enabled/disabled by GIE ⇒ INTCON<7>
  - Can also be enabled/disabled by PEIE ⇒ INTCON<6>

- Individual Interrupt Enable bits XXXXIE enable/disable response to individual interrupt requests
Interrupt Control: Compatibility Mode

Note: No interrupt priority bits now

GIE = INTCON<7>

RCON = 0xxx xxxx

Interrupt to CPU Vector at 0x0008

5 'Core' Interrupts:
- TMR0IF
- TMR0IE
- RBIF
- RBIE
- INT0IF
- INT0IE
- INT1IF
- INT1IE
- INT2IF
- INT2IE

11 Peripheral Interrupts:
- TMR1IF
- TMR1IE
- xxxIF
- xxxIE
- EEIF
- EEIE

PEIE = INCON<6>
Interrupt Response Cycle: Compatibility Mode

When an interrupt request is responded to

- GIE is cleared, disabling further interrupts
- Return address is pushed onto the stack
- PC is loaded with address 0x0008

- User ISR must do everything else
- User ISR must end in
  - RETFIE instruction – pops the return address and sets GIE

- Note: FAST return is not applicable in this mode
- Configuration is similar to priority mode
I/O Port Hardware

Using I/O Ports as Digital I/O, not as the Alternate Peripherals…
I/O Ports

- PIC18F452 has five digital I/O ports: PORTA, PORTB, PORTC, PORTD, and PORTE

- PIC18F2X2 parts omit PORTD, (the PSP) and PORTE (PSP control bits or three analog inputs)
Most I/O bits are either
- TTL-compatible I/O, outputs with totem-pole drivers
- Schmitt trigger inputs with CMOS levels

Typical voltages at $V_{SS} = 0V$ and $V_{DD} = 5.0V$ supply are
- **TTL-compatible I/O**
  - $V_{IL\text{max}} = 0.8V$
  - $V_{IH\text{min}} = 2.0V$
- **Schmitt trigger Inputs**
  - $V_{IL\text{max}} = 1.0V$
  - $V_{IH\text{min}} = 4.0V$
- Most pins have protection diodes to $V_{SS}$ and $V_{DD}$
Alternate Port Functions

Each bit (pin) on each port can be used as a general-purpose digital I/O pin, or

- **Port A:** (7 bits)
  - Analog inputs (5 channels), A/D Reference voltage inputs, Low voltage detect input
  - Crystal oscillator input
  - System (machine cycle) clock output
  - Timer0 clock input

- **Port B:** (8 bits)
  - Four external interrupt pins
  - In-circuit debug/programming pins
Alternate Port Functions (2)

- **Port C**: (8 bits)
  - USART I/O pins
  - Synchronous Serial Port (SPI or I²C) pins
  - Capture/Compare/PWM I/O pins
  - Timer1 I/O pins

- **Port D**: (8 bits) (not bonded out on 28-pin devices)
  - Simplest hardware
  - Parallel Slave Port (PSP) data bus

- **Port E**: (3 bits) (not bonded out on 28-pin devices)
  - Three analog input channels – or –
  - Parallel Slave Port control bus (bits 2:0)
I/O Port Registers

- Each port has three associated registers
  - \texttt{TRISx}: Data direction register
  - \texttt{PORTx}: Input latch – reads levels on the port pins
  - \texttt{LATx}: Output latch – useful for read-modify-write operations

\( (x \in \text{A, B, \ldots, E}) \)
Port D: PORTD, TRISD and LATD

- **PORTD** is an 8-bit bidirectional port
- **TRISD** is the Data Direction register
  - Setting (=1) a bit makes that pin an Input: 1 \(\Rightarrow\) I and makes the pin output driver high-impedance
  - Clearing (=0) a bit makes that pin an Output: 0 \(\Rightarrow\) O

- Reading **PORTD** register will read the pin status
- Writing **PORTD** register will write to the port latch

- Reading/writing **LATD** has slightly different effects, depending on if the bit is configured as input or output – see next slide.
Port D: Registers and Circuit

Simplest hardware – 8-bit bidirectional port
Not present on PIC18F2X2 (28 pin) parts

Data Direction
- On WR TRISD ↓ edge, data bit latched into TRIS latch

Digital Output Function (TRISD<n>=0)
- On WR LATD ↓ edge, data bit latched into output (data) latch
- On RD LATD high, data just written can be read back

Digital Input Function (TRISD<n>=1)
- On RD PORTD high, data bit held in transparent latch can be read on data bus
- Latch prevents changes during bus read

Note 1: I/O pins have diode protection to Vdd and Vss.
Port D Bit Definitions

**PORTD: 8-bit Bi-directional Port**

<table>
<thead>
<tr>
<th>Bit Names</th>
<th>RD7</th>
<th>RD6</th>
<th>RD5</th>
<th>RD4</th>
<th>RD3</th>
<th>RD2</th>
<th>RD1</th>
<th>RD0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSP7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSP6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSP5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSP4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSP3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSP0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alternate Pin Functions (See PIC18FXX2 Data Sheet for bit definitions)
- Parallel Slave Port (PSP) data bus (bits 7:0)
Port A: PORTA, TRISA and LATA

- PORTA is a 7-bit bidirectional port
- Direction register TRISA and latch register LATA function identically to those for Port D
Port A Bit Definitions

Alternate Pin Functions (See PIC18FXX2 Data Sheet for bit definitions)

- Analog inputs (bits 5, 3:0)
- A/D Reference voltage inputs (bits 3,2)
- Low voltage detect input (bit 5)
- OSC2 – Xtal oscillator input 2 (bit 6)  [OSC1/CLKI is not shared]
- Clock output (bit 6)
- Timer0 clock input (bit 4)
Port A I/O Circuits

Direction register TRISA and latch register LATA function identically to those for Port D

Data Direction
- On WR TRISA \(\downarrow\) edge, data bit latched into TRIS latch
- Must set (=1) TRISA for analog inputs!!

Digital Output Function (TRISA\(<n>=0\))
- On WR LATA \(\downarrow\) edge, data bit latched into output (data) latch
- Output *driven* high and low (totem pole output)
- On RD LATA high, data just written can be read back

Digital Input Function (TRISA\(<n>=1\))
- On RD PORTA high, data bit held in transparent latch can be read on data bus
- Latch prevents changes during bus read

Note 1: I/O pins have protection diodes to VDD and VSS.
Port A I/O Circuits

**RA4 (T0CLKI)**

- Data Bus
- WR LATA or PORTA
- WR TRISA
- RD PORTA
- TMRO0 Clock Input

**RA6 (OSC2/CLKO)**

- Data Bus
- WR LATA or PORTA
- WR TRISA
- RD PORTA

Note 1: I/O pin has protection diode to Vss only.

Note 1: I/O pins have protection diodes to VDD and Vss.

Note – output is open drain

Totem Pole (push-pull) outputs
Port B: PORTB, TRISB and LATB

- PORTB is an 8-bit bidirectional port
- Direction register TRISB and latch register LATB function identically to those for Port A

- Each I/O pin has a weak pull-up, active only on inputs
- All pullups enabled by clearing #RBPU ≡ INTCON2<7>
- Typical pullup current 50µA, max 450 µA at $V_{pin} = V_{SS}$

- RB2 : RB0 have alternate functions INT2 : INT0
- RB7 : RB4 have interrupt-on-change functionality
Alternate Pin Functions (See PIC18FXX2 Data Sheet for bit definitions)

- External interrupt inputs (bits 3:0)
- Alternate (non-default) Capture/Compare/PWM 2 I/O pin (bit 4) – enable via Configuration Registers
- In-circuit serial programming inputs (bits 7:5)
Port B I/O Circuits

Interrupt-on-change

- Four pins RB7:RB4 can request interrupt on state change
- Pin must be configured as input for this to be active

- Any MOVxx instruction has execute cycle with
  - Q1 – decode
  - Q2 – read data
  - Q3 – process data
  - Q4 – write result

- Schematic shows “set RBIF if the state read has changed from the previous state” – not quite correct?
Port B I/O Circuits

External interrupt pins

- Three pins can be configured as external interrupt pins: RB2 : RB0 ≡ INT2 : INT0
- Pin must be configured as input for this to be active

Note 1: I/O pins have diode protection to VDD and VSS.
Note 2: To enable weak pull-ups, set the appropriate TRIS bit(s) and clear the RBPU bit (OPTION_REG<7>).
Port C: PORTC, TRISC and LATC

- PORTC is an 8-bit bidirectional port
- Direction register TRISC and latch register LATC function identically to those for Port A
- All Port C pins have Schmitt trigger input buffers

Caution! Take care when setting the TRISC bits:
- Some peripherals force the pin to be an input
- Some peripherals force the pin to be an output
Port C Bit Definitions

**PORTC: 8-bit Bi-directional Port**

<table>
<thead>
<tr>
<th>Bit Names</th>
<th>p. 93 et seq.</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC7</td>
<td>RC6</td>
<td>RC5</td>
<td>RC4</td>
<td>RC3</td>
<td>RC2</td>
<td>RC1</td>
<td>RC0</td>
<td></td>
</tr>
<tr>
<td>RX</td>
<td>TX</td>
<td>SDO</td>
<td>SDI</td>
<td>SCK</td>
<td>–</td>
<td>T1OSI</td>
<td>T1OSO</td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td>CK</td>
<td>–</td>
<td>SDA</td>
<td>SCL</td>
<td>CCP1</td>
<td>CCP2</td>
<td>T1CKI</td>
<td></td>
</tr>
</tbody>
</table>

**Alternate Pin Functions** (See PIC18FXX2 Data Sheet for bit definitions)

- UART Asynchronous I/O (bits 7: 6)
- UART Synchronous I/O (bits 7: 6)
- Synchronous Serial Port (bits 5:3)
- Default Capture/Compare/PWM I/O pins – CCP1 & CCP2 (bits 2: 1)
- Timer1 I/O (bits 1: 0)
Port C Circuits

Note that
- Peripheral Data Out bypasses the Output Data Latch
- Peripheral Output Enable determines output buffer activity for peripheral writes (*not* TRISC)
- Peripheral Data In (RX, SDI, CCP1, CCP2) bypasses the Input Latch

Note 1: I/O pins have diode protection to VDD and VSS.
Note 2: Port Peripheral Select signal selects between port data (input) and peripheral output.
Note 3: Peripheral Output Enable is only active if peripheral select is active.
Port E: Registers and Circuit

- **PORTE** is a 3-bit bidirectional port
- Direction register **TRISE** and latch register **LATE** function identically to those for Port A
- Not present on PIC18F2X2 parts
- All Port E pins have Schmitt trigger input buffers

![Diagram of Port E registers and circuit](image)

**Note 1:** I/O pins have diode protection to VDD and VSS.
## Port E Bit Definitions

<table>
<thead>
<tr>
<th>PORTE: 3-bit Bi-directional Port</th>
<th>0xF84</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Names</td>
<td></td>
</tr>
<tr>
<td>p. 99 et seq.</td>
<td></td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>R/W</td>
<td>R/W</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>RE2</td>
<td>RE1</td>
</tr>
<tr>
<td>RE0</td>
<td></td>
</tr>
<tr>
<td>#CS</td>
<td>#WR</td>
</tr>
<tr>
<td>#RD</td>
<td></td>
</tr>
<tr>
<td>AN7</td>
<td>AN6</td>
</tr>
<tr>
<td>AN5</td>
<td></td>
</tr>
<tr>
<td>PO/BO Reset</td>
<td></td>
</tr>
<tr>
<td>MCLR Reset</td>
<td></td>
</tr>
<tr>
<td>bit 7</td>
<td></td>
</tr>
<tr>
<td>bit 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bit Names</td>
<td></td>
</tr>
</tbody>
</table>

Alternate Pin Functions (See PIC18FXX2 Data Sheet for bit definitions)
- Analog inputs (bits 2:0)
- Parallel Slave Port control bus (bits 2:0)
Parallel Slave Port (PSP)

Used to turn the PIC18 into a custom, programmable 8-bit microprocessor interface IC
Parallel Slave Port (PSP)

- Can make custom ICs that interface directly to an 8-bit microprocessor bus
- PORTD can be configured as an 8-bit PSP with control signals on PORTE
- All Port D pins have TTL buffers in PSP mode
- All Port E pins have Schmitt trigger input buffers
- Not present on PIC18F2X2 parts (only 28 pins)
Parallel Slave Port (PSP)

- PSP Enabled by setting TRISE<PSPMODE>
- Must also ensure
  - TRISE<2:0>=1
    (Port E pins are inputs)
  - ADCON1 s.t. Port E pins are digital
- Port D becomes an 8-bit data bus
- Port E provides 3 control signals:
  - #CS ≡ RE<2> – Chip Select
  - #WR ≡ RE<1> – Write
  - #RD ≡ RE<0> – Read

Note: I/O pin has protection diodes to VDD and VSS.
- Read from (output from) the PSP begins when \#CS and \#RD are first detected to be low
  - \#OBF (Output Buffer Full) set when available byte has not been read from the PSP by external device
- Both read and write can request an interrupt
PSP Write Timing

- Write to (input to) the PSP begins when #CS and #WR are first detected to be low
  - #IBF (Input Buffer Full) set when byte can be read by CPU
  - #IBOV (Input Buffer Overflow flag) set if next byte written by external device before previous was read by CPU