

# HANDS-ON

# Training

## 203 PRC

### 16-bit Basic Peripherals and Programming using C30





# Objectives

- **When you walk out of this class, you will know**
  - 16-bit family Architecture
  - 16-bit family Special Features
  - 16-bit family Basic Peripherals



# Agenda

- **Architectural Overview**
  - Programmers' Model
    - **Program Memory**
    - **Data Memory**
    - **C30 for 16-bit devices**
      - **LAB 1: How to use C30**
    - **Program Space Visibility (PSV)**
      - **LAB 2: PSV Initialization and its usage**
    - **Stack and Frame pointer**
  - Addressing Modes
  - Interrupt and Trap Handling
    - **LAB 3: Interrupt Service Routine and Interrupt Priority**
- **Special features**
  - Oscillators and Power Saving modes
  - Resets
  - WDT



# Agenda (*Continued..*)

- **Basic Peripherals**

- I/O Ports
- ADC
  - **LAB 4: ADC Configuration & usage**
- Comparators & Voltage Reference generator
- Timers
  - **LAB 5: Configuration of 32-bit Timers**
- Capture
- Compare & PWM
- UART
  - **LAB 6: UART configuration and usage of FIFO**
- I<sup>2</sup>C™
- SPI

# HANDS-ON

# Training

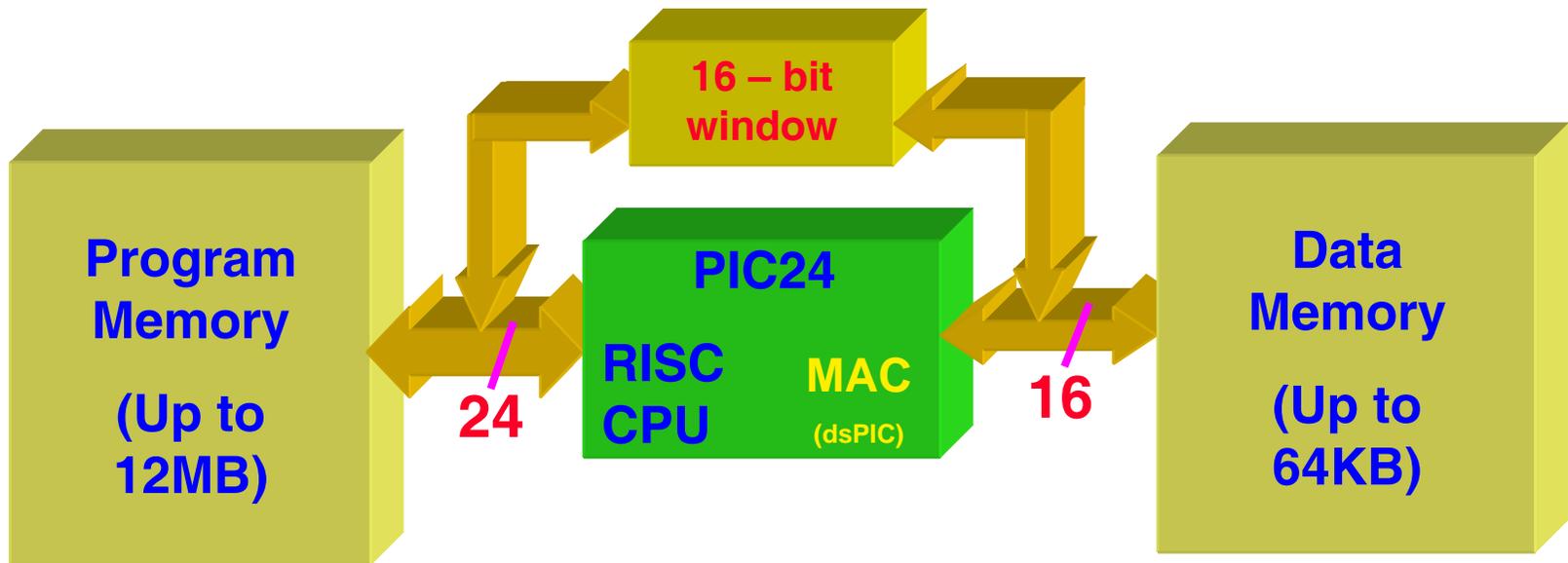
## Basic Architecture





# Harvard Architecture

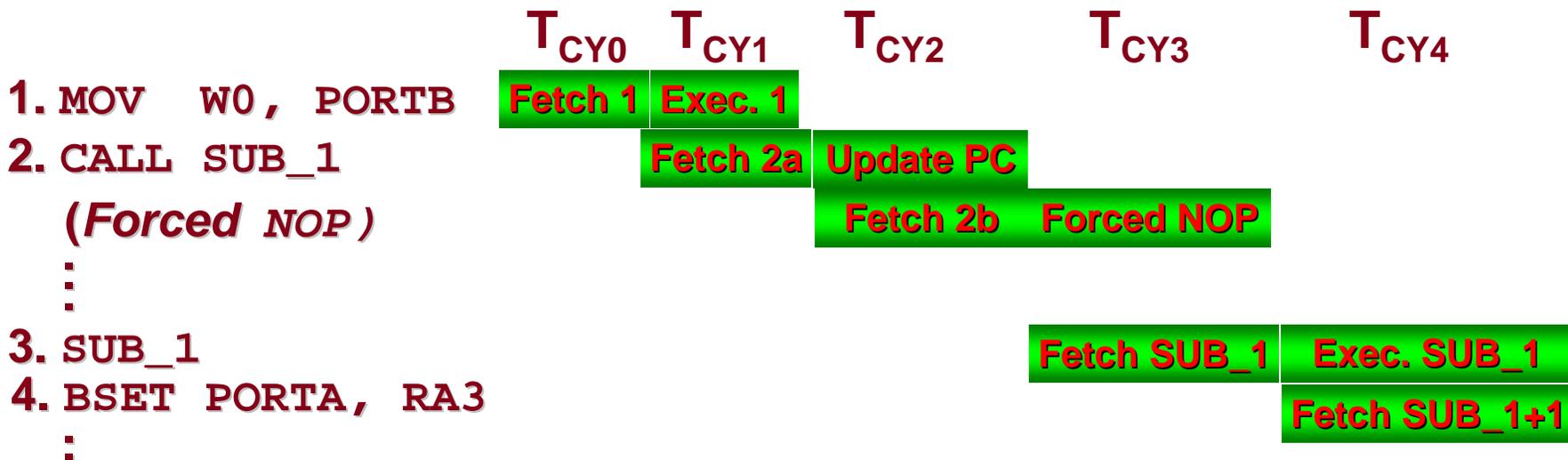
- 16-bit microcontroller
- 24-bit Instruction width
- Data Transfer Mechanism between PM and DM





# Instruction Pipeline

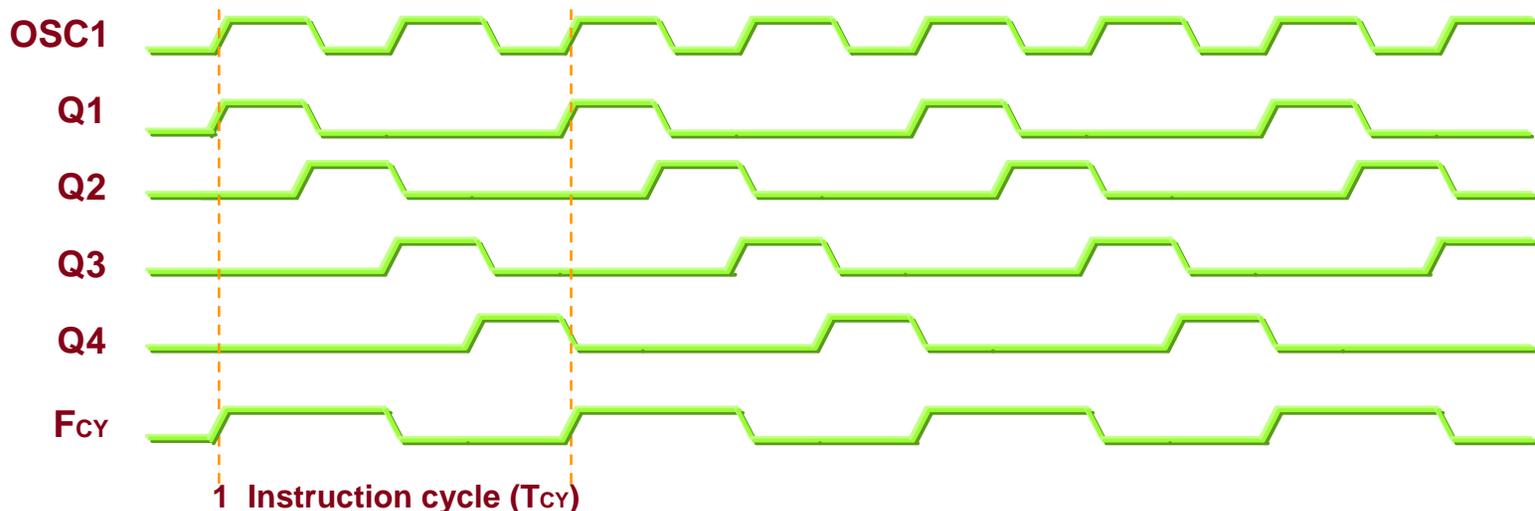
- Allows overlap of fetch and execution
- Makes single cycle execution
- Program branches (e.g. GOTO, CALL) take two cycles





# Clocking Scheme

- **Instruction frequency = 1/2 of clock input frequency**
- **PIC24F**
  - 62.5 ns Instruction cycle @ 32 MHz clock
  - 16 MIPS @ 32 MHz clock (PIC24F)
- **PIC24H/ dsPIC33F**
  - 25 ns Instruction cycle @ 80 MHz clock
  - 40 MIPS @ 80MHz clock (PIC24H)



# HANDS-ON

# Training

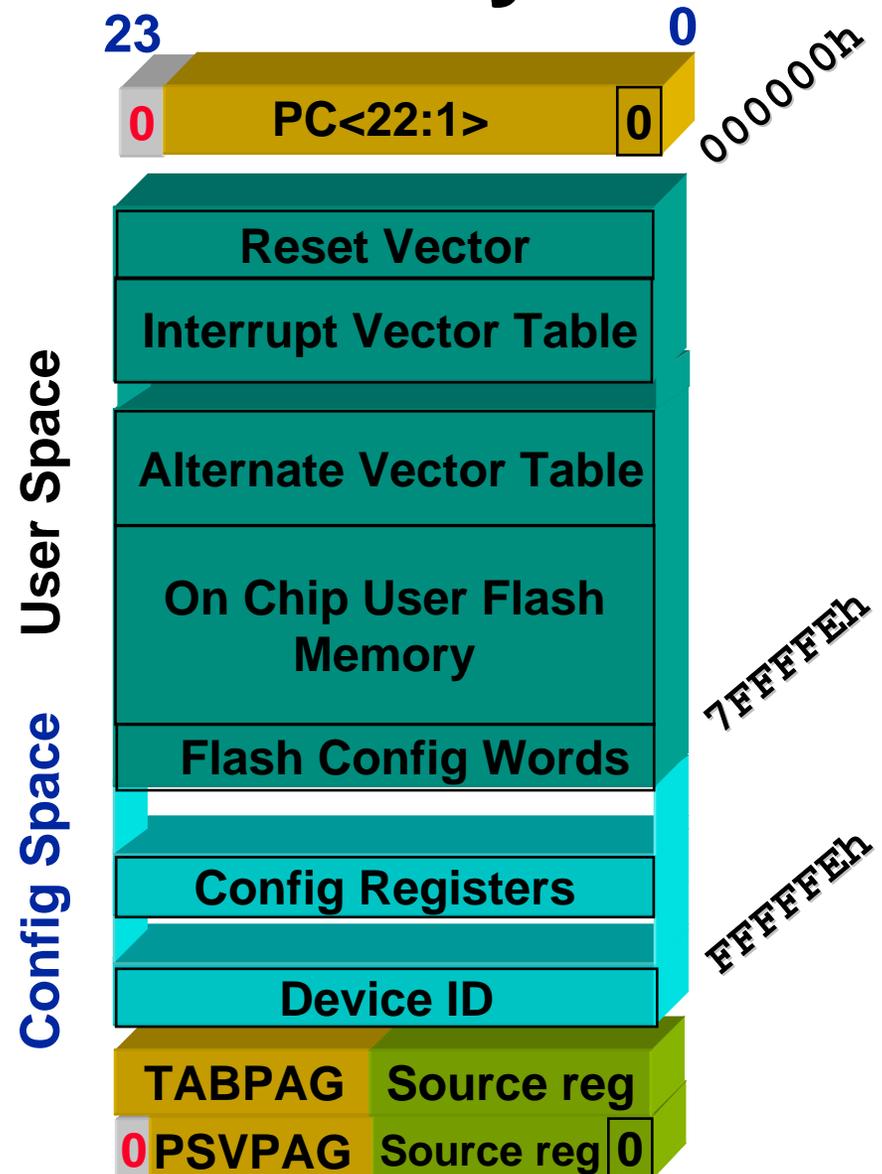
# Program Memory





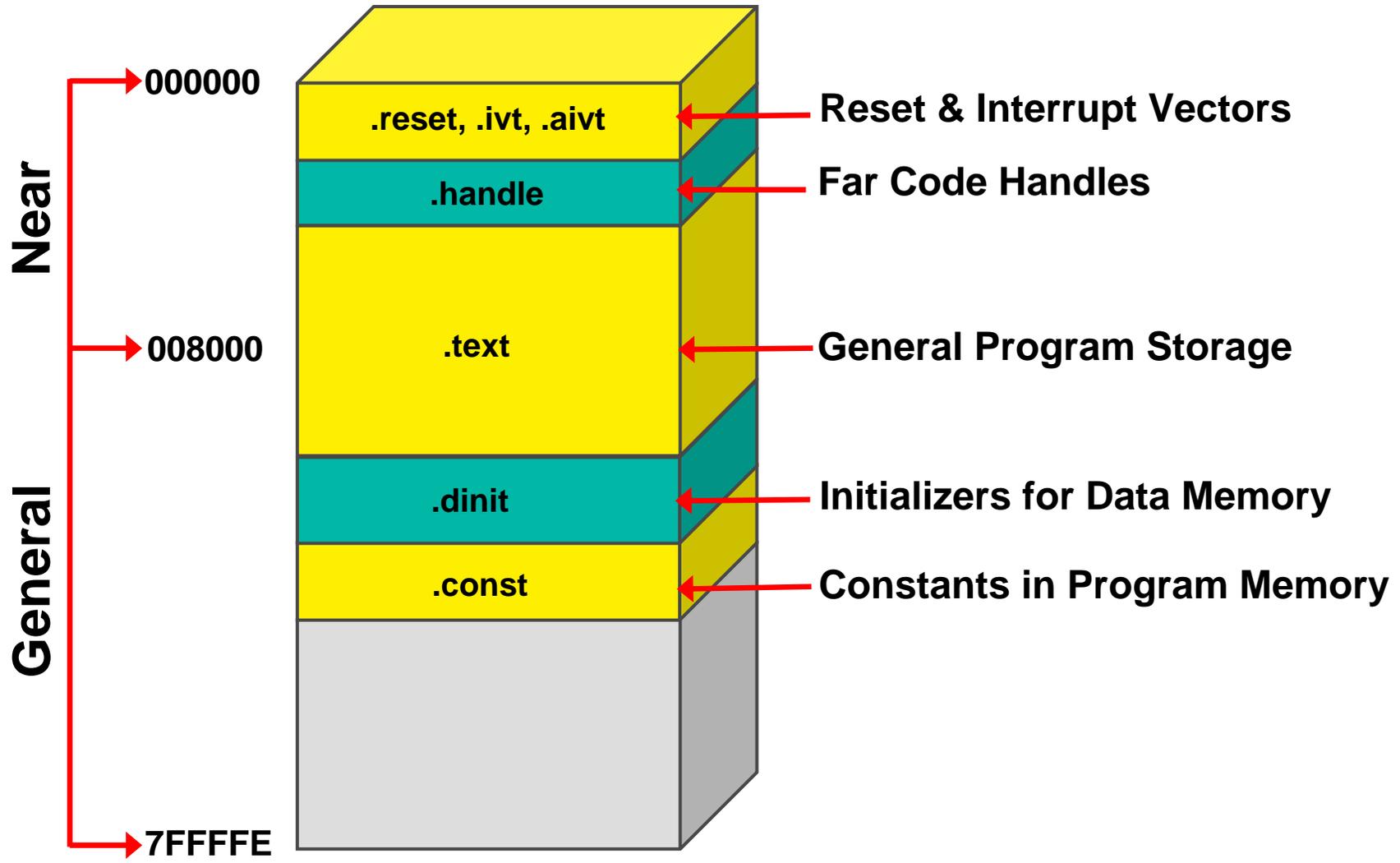
# Program Memory

- **Maximum 12MB**
  - 4MB x 24-bit
  - 23-bit PC (PCH & PCL)
- **PC increments in words (LSB always '0')**
- **Reset Vector at 0**
- **Interrupt Vector Table from 4h to FEh**
- **User Code space from 200h to 7FFFFFFEh (what ever is implemented)**



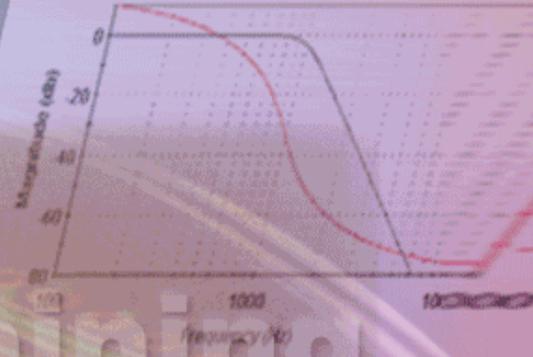


# Program Memory Space seen by C30



# HANDS-ON

# Training

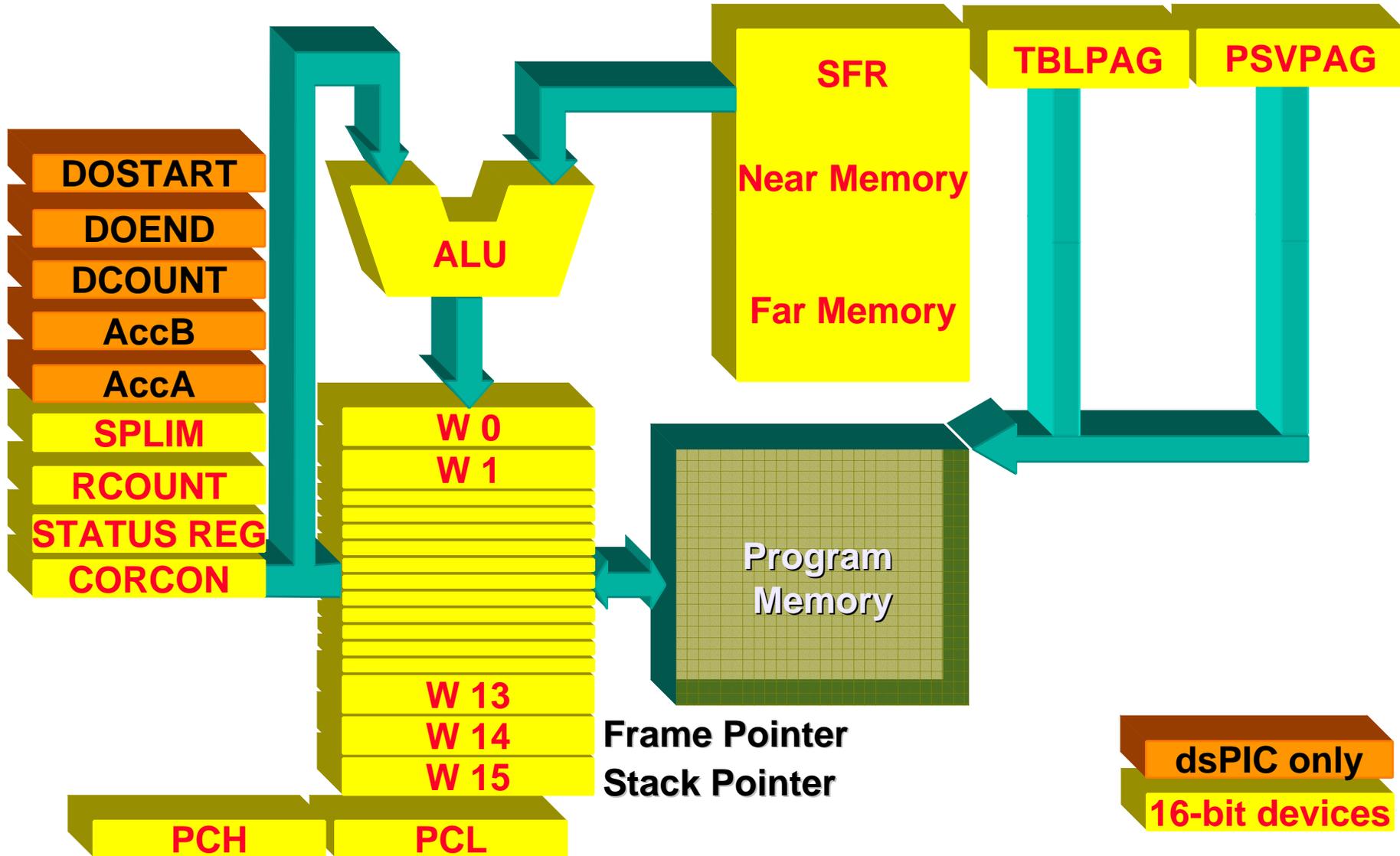


# Data Memory



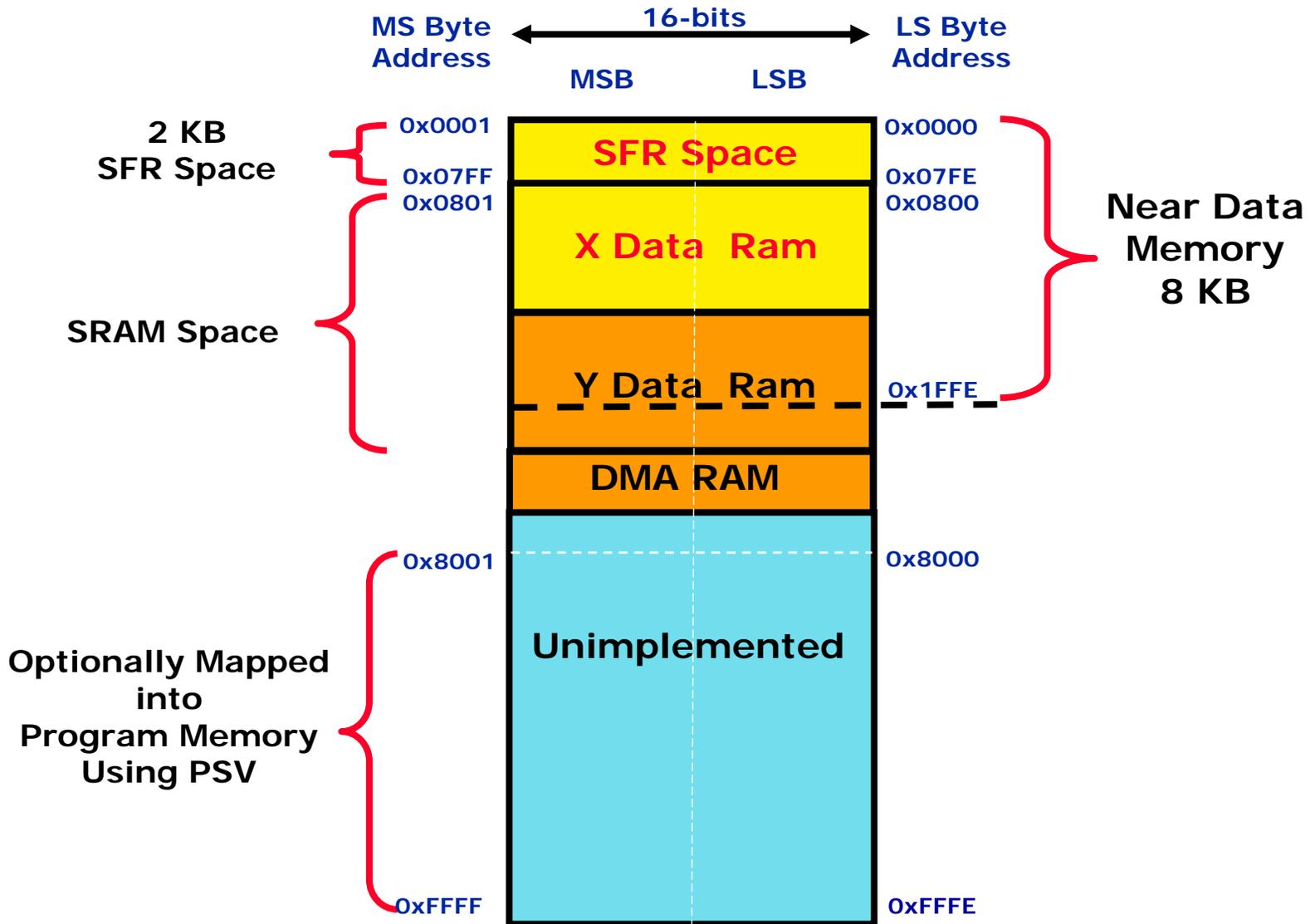


# 16-bit Architecture Programmers model



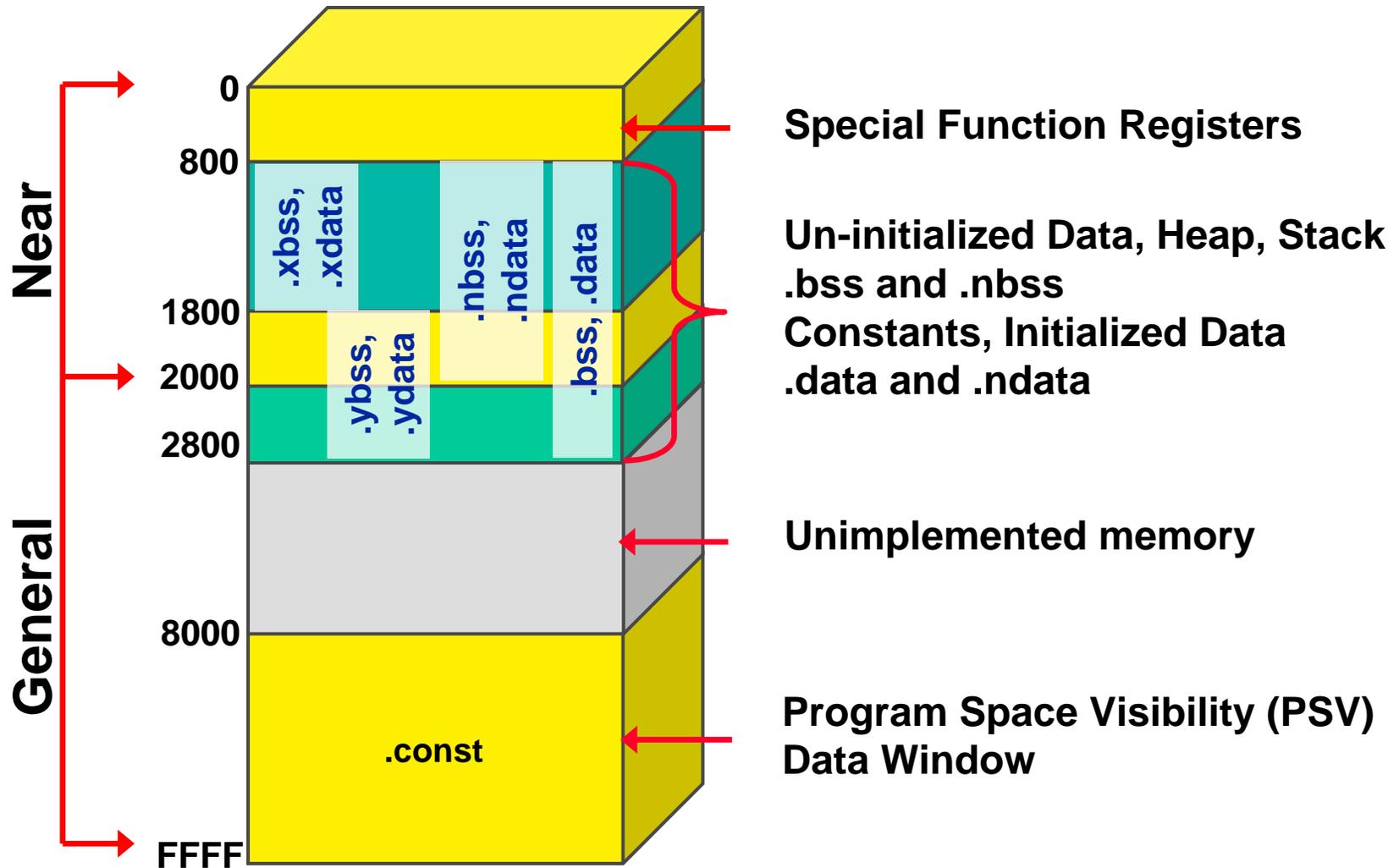


# Data Memory Organization



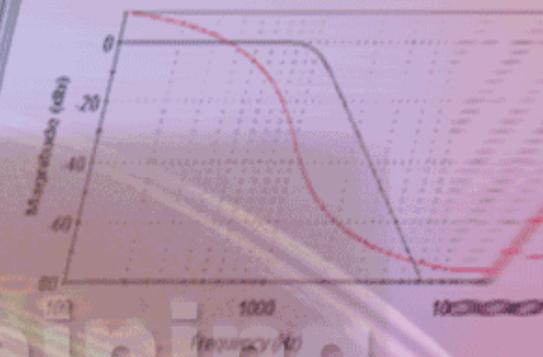


# Data Memory Space seen by C30



# HANDS-ON

# Training



# Lab 1 MPLAB<sup>®</sup> IDE





# Lab 1: Working with C30 and MPLAB IDE

- **Goals:**

- To work with C30 and MPLAB® IDE environment

- **To Do:**

- Follow the presenter

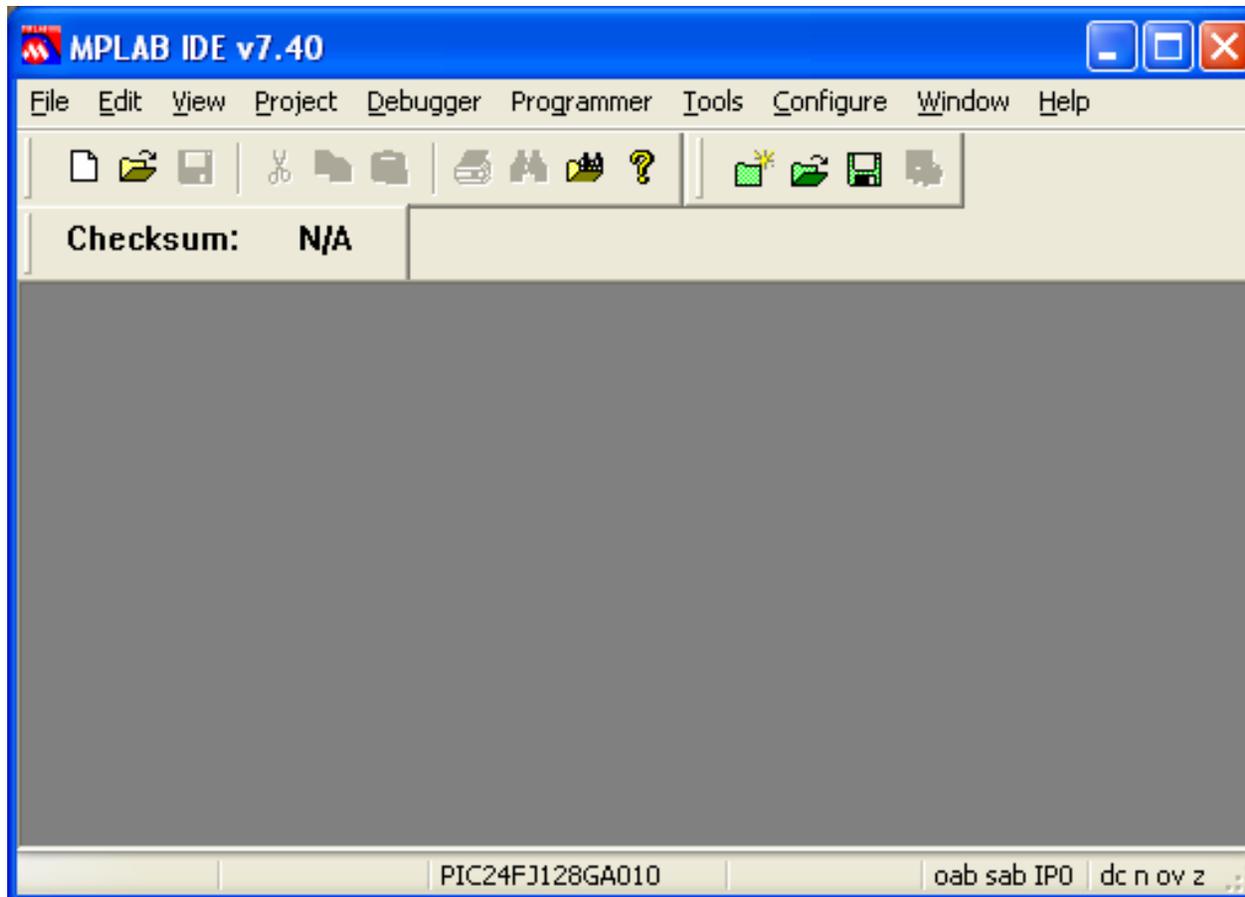
- **Expected Result:**

- Successfully build the project and program the device
- LED D3 should turn on



# 16-bit Development Tools MPLAB® IDE

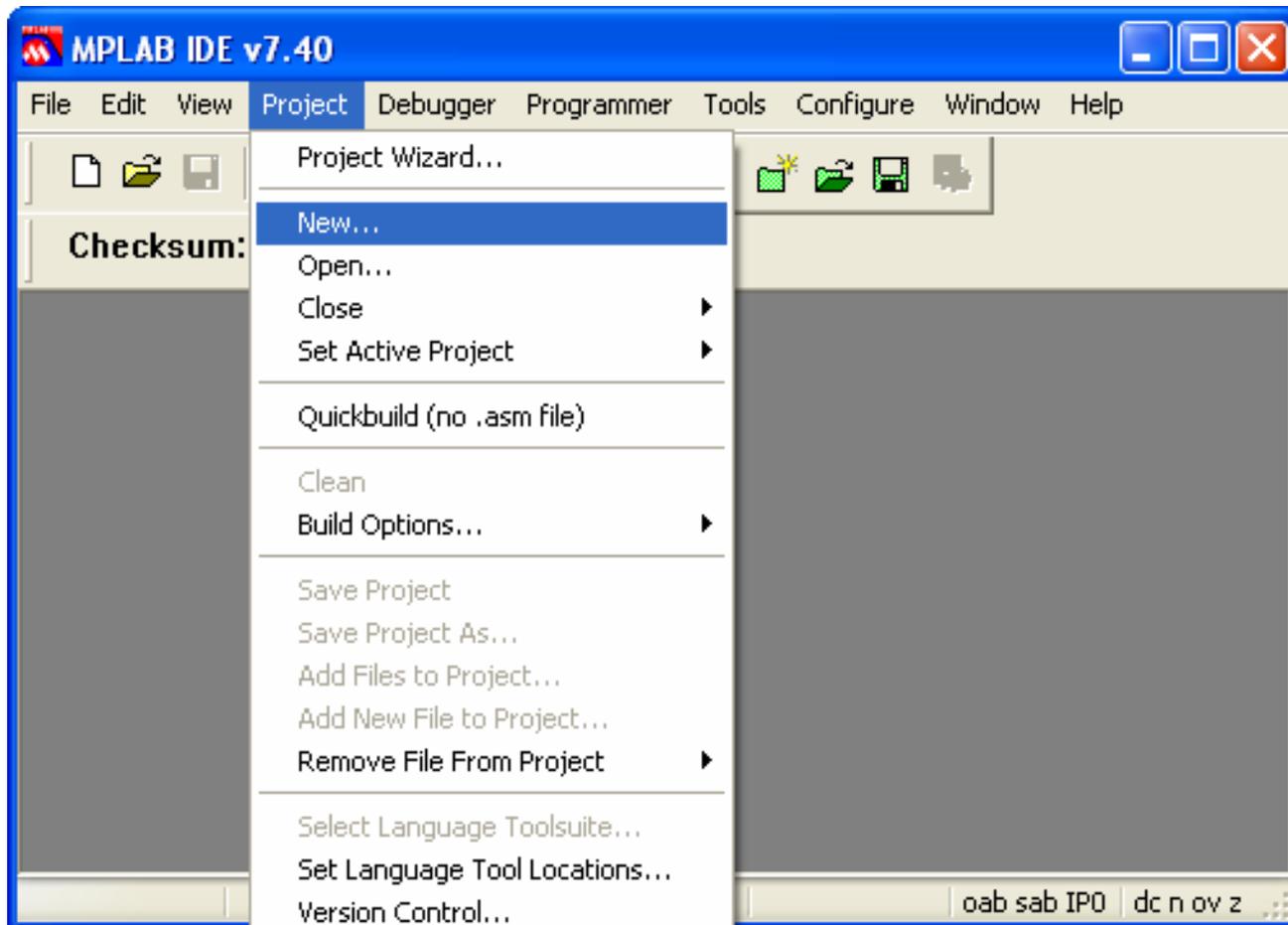
Launching the **MPLAB®** from windows environment





# 16-bit Development Tools MPLAB<sup>®</sup> IDE

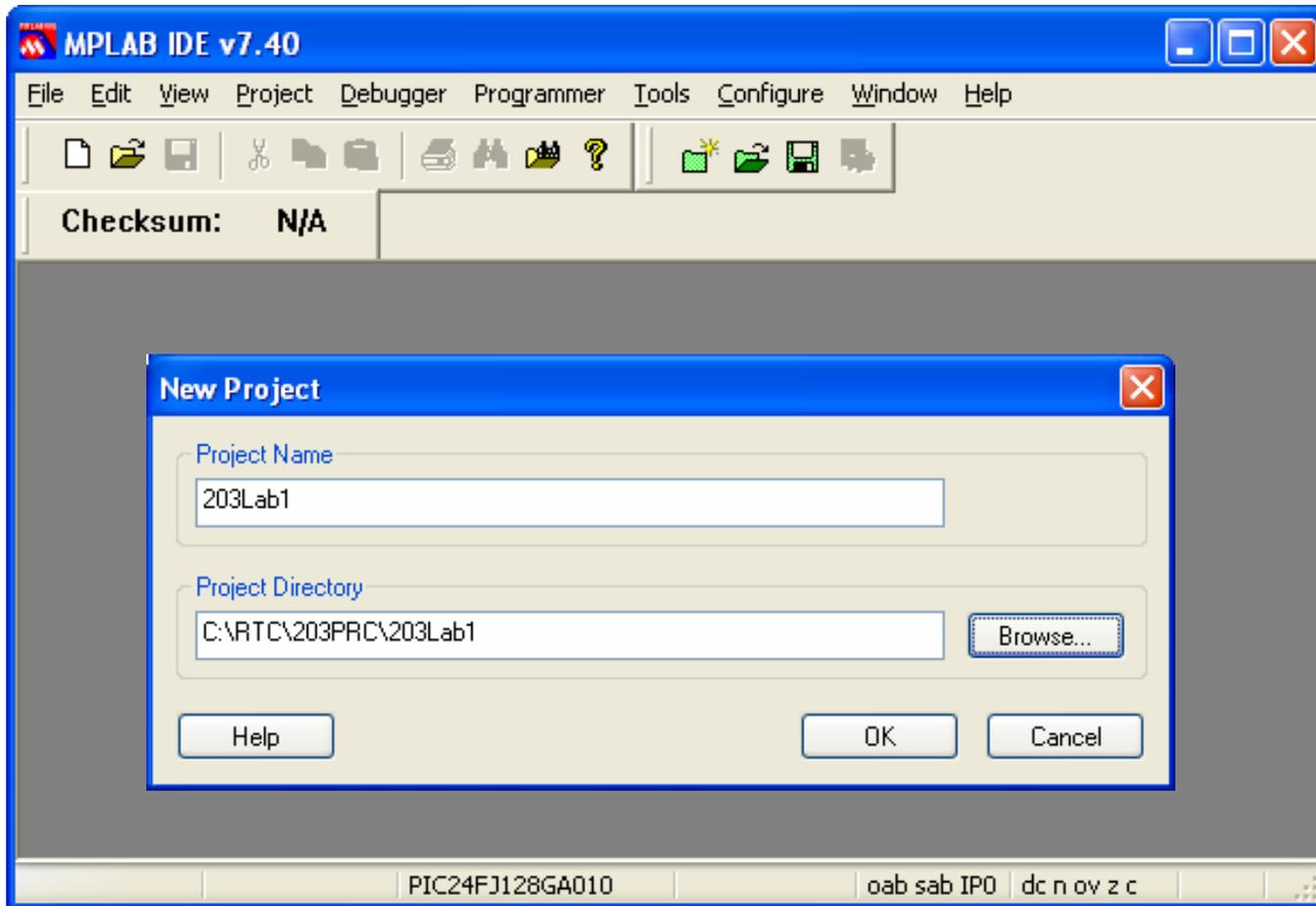
## Creating a New Project





# 16-bit Development Tools MPLAB® IDE

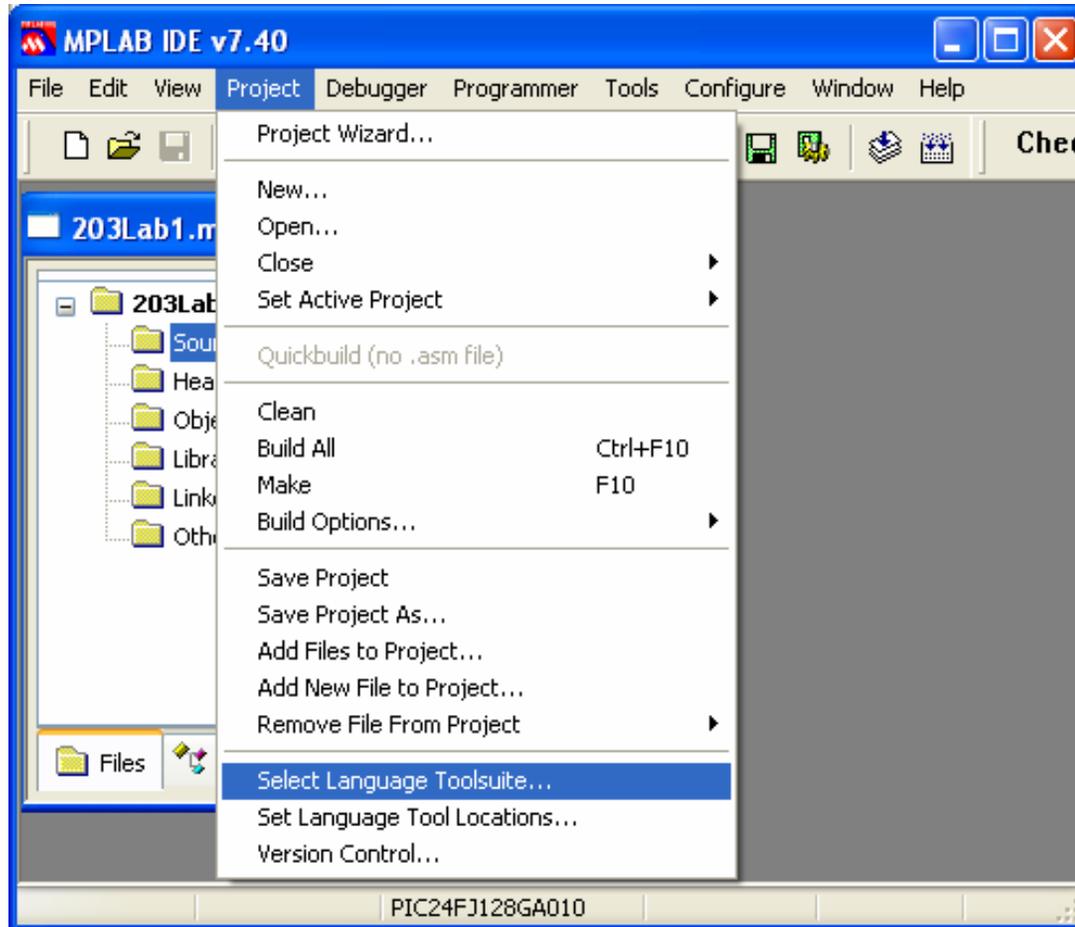
## Naming a new project





# 16-bit Development Tools MPLAB<sup>®</sup> IDE

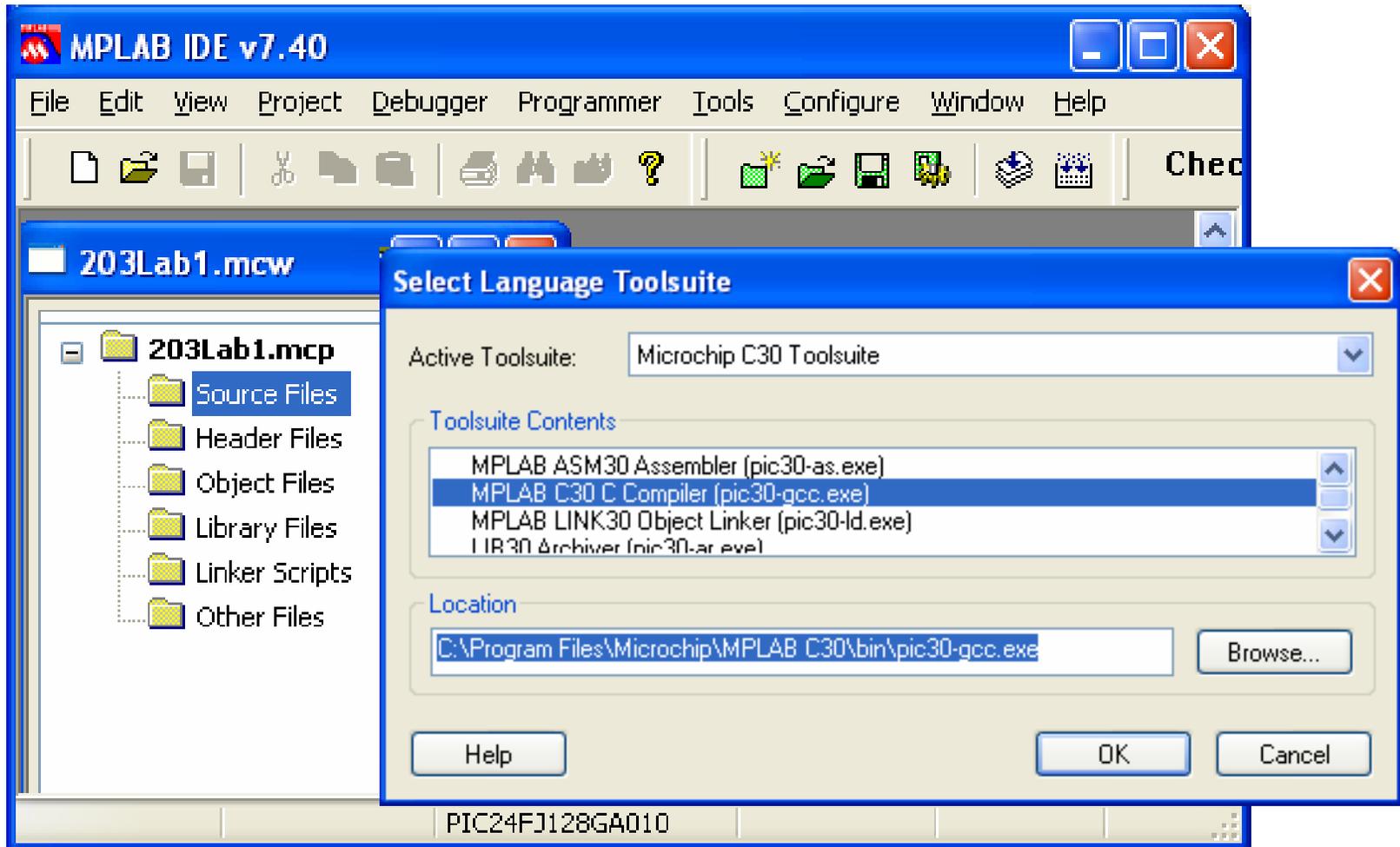
Selecting the Language tool (C30, Third party tool suit or MPASM )





# 16-bit Development Tools MPLAB® IDE

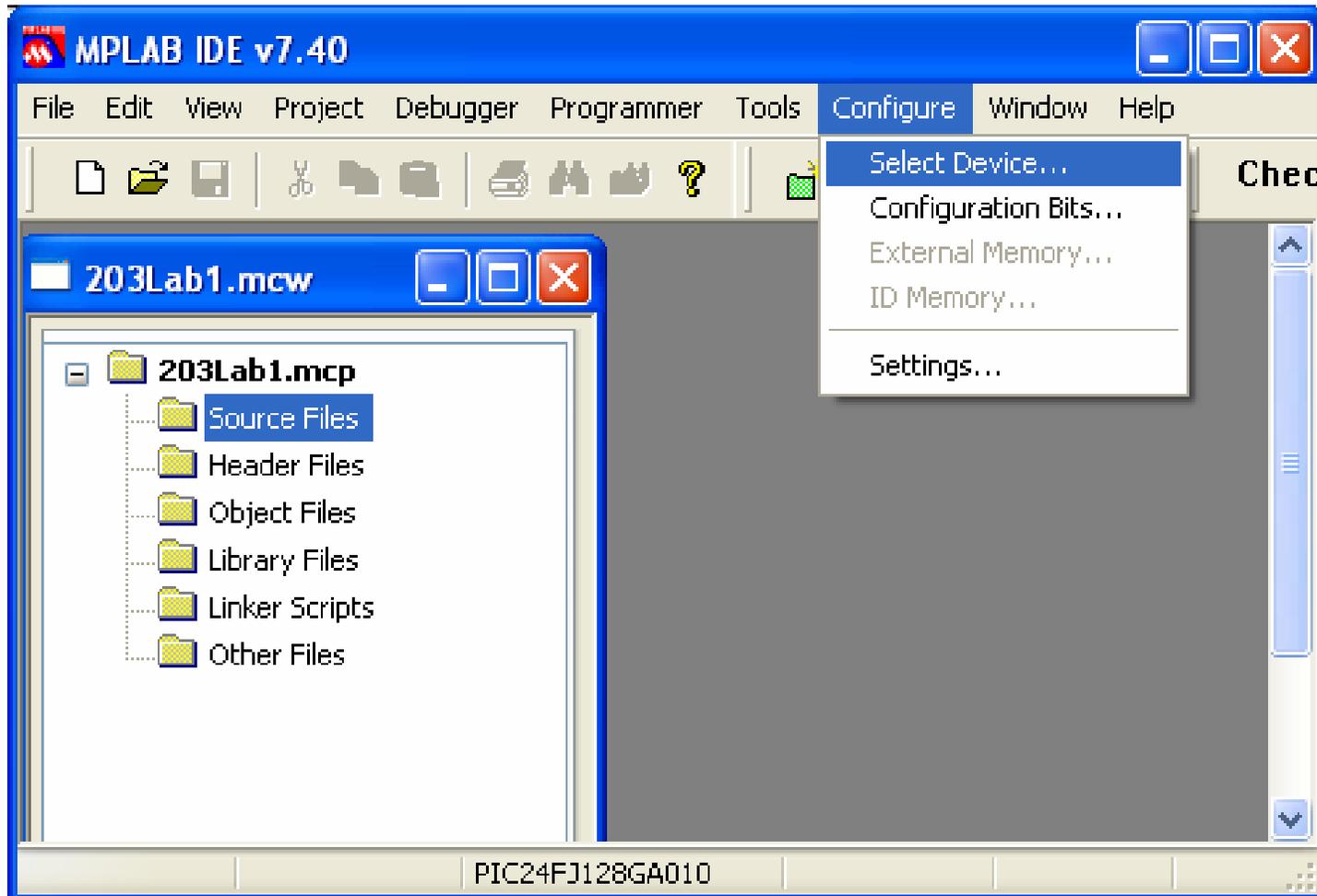
Select Microchip C30 tool suit





# 16-bit Development Tools MPLAB® IDE

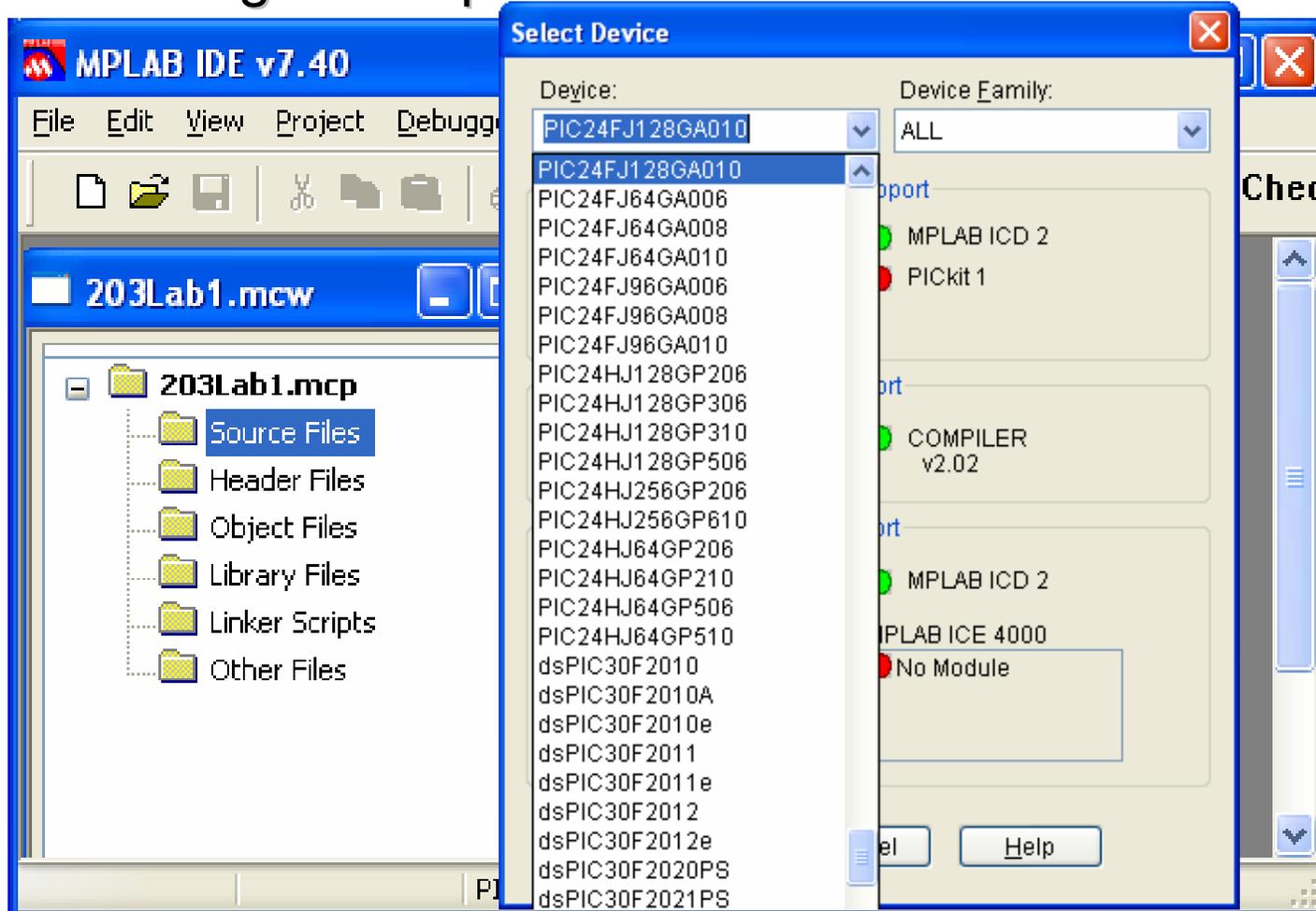
Selecting the required controller from list of devices





# 16-bit Development Tools MPLAB® IDE

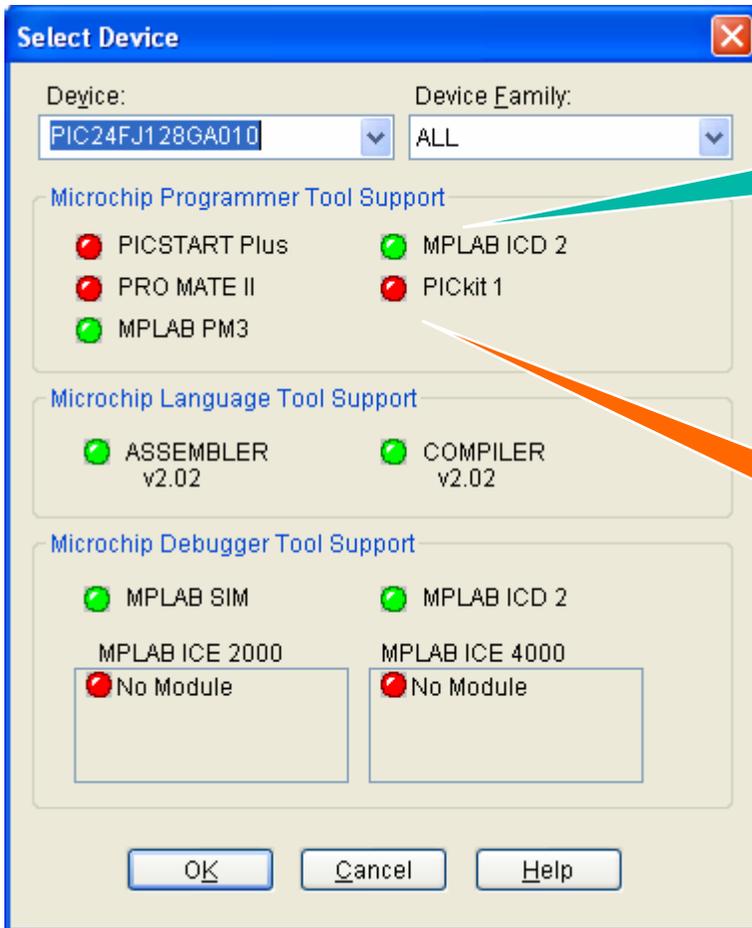
Selecting the required controller from list of devices





# 16-bit Development Tools MPLAB® IDE

Looking at supported tools by MPLAB® for the selected device



**GREEN** button indicates the tool support is validated and functional for the selected device

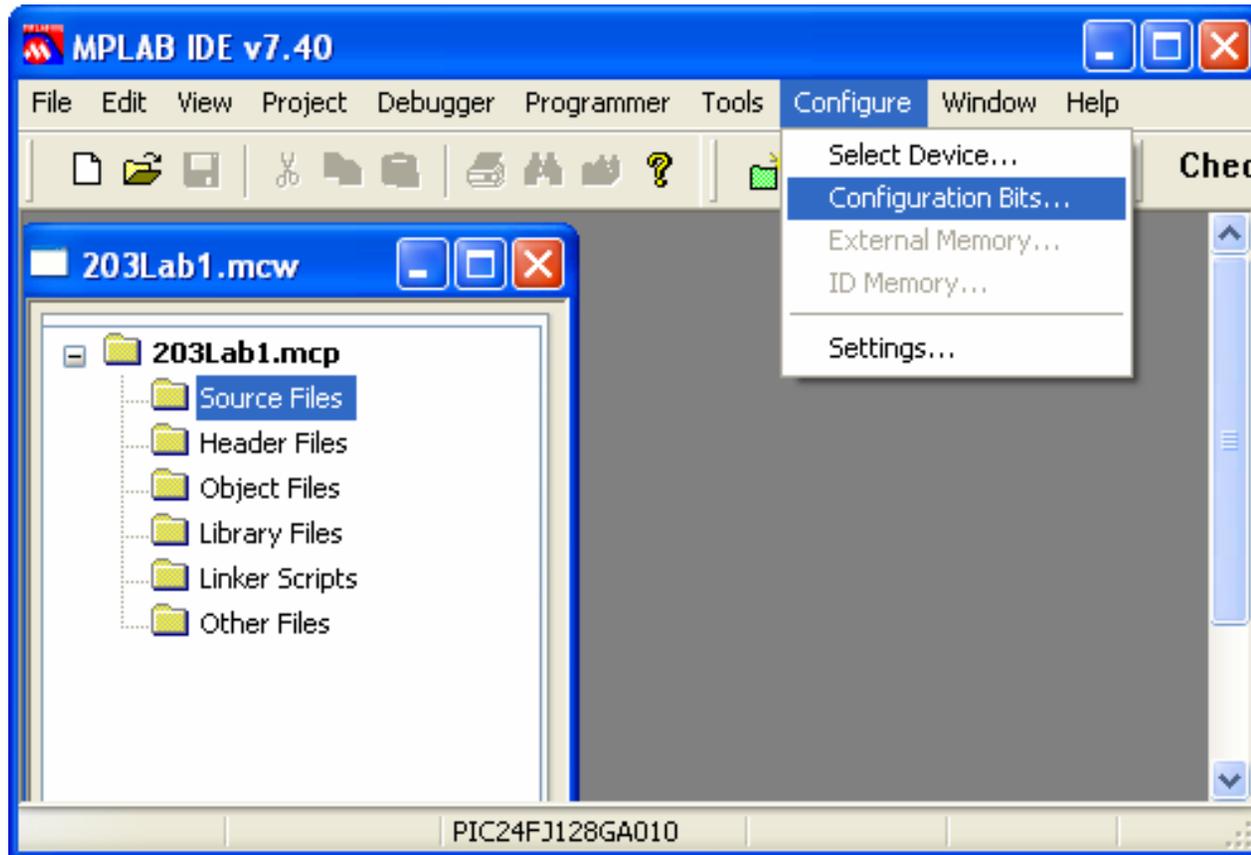
**YELLOW** button indicates the tool support is provided and functional; but not fully validated for the selected device

**Red** button indicates the tool support is not available for the selected device



# 16-bit Development Tools MPLAB<sup>®</sup> IDE

You Configure the device core from this window; alternatively you can use `_config` macro to set the configuration word in the software





# 16-bit Development Tools MPLAB® IDE

Select the config setting as per requirements.

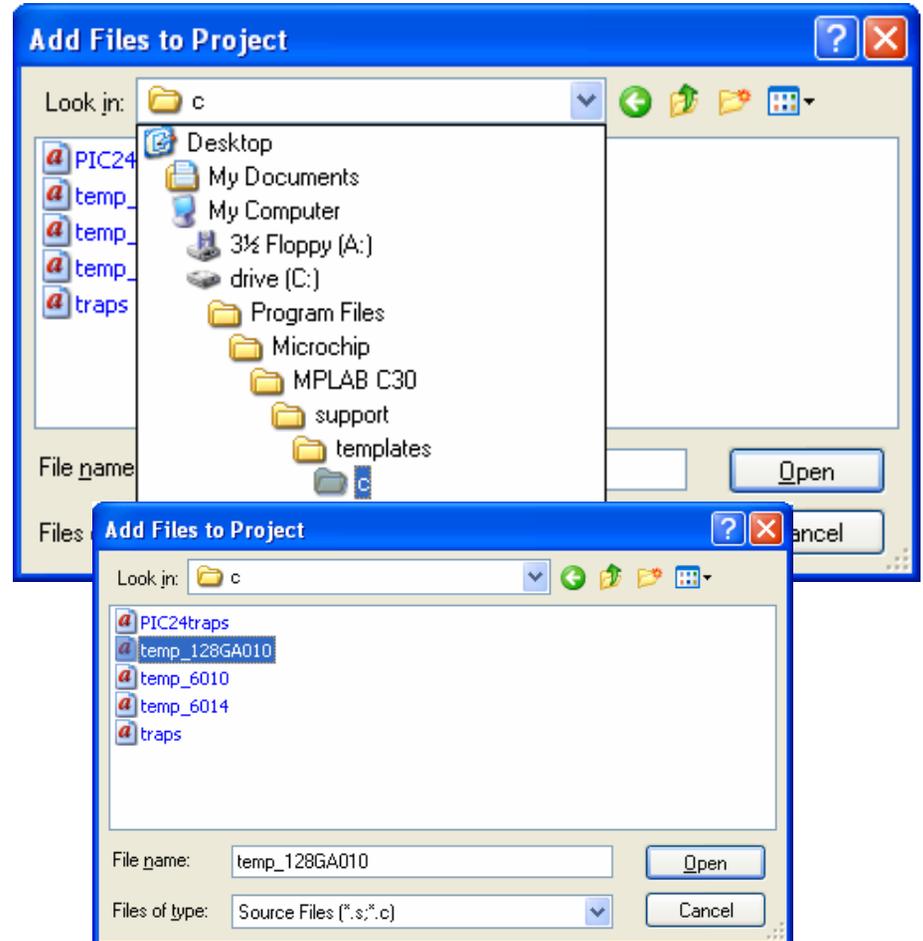
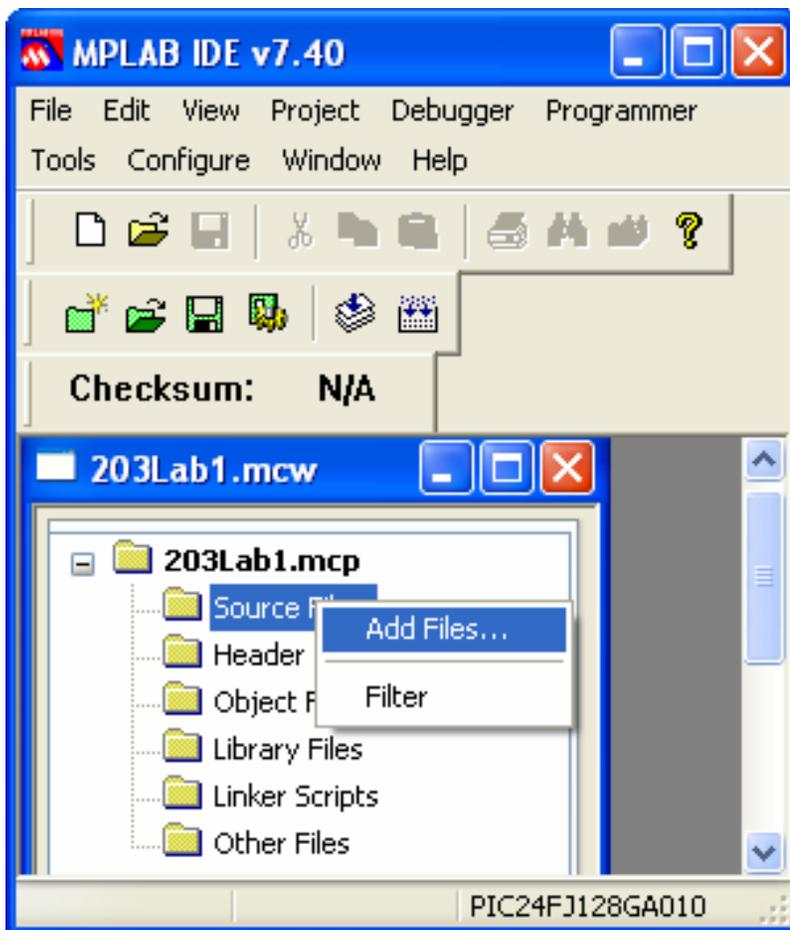
The screenshot shows the MPLAB IDE v7.40 Configuration Bits window. The window title is "MPLAB IDE v7.40" and the device selected is "PIC24FJ128GA010". The menu bar includes File, Edit, View, Project, Debugger, Programmer, Tools, Configure, Window, and Help. The toolbar contains various icons for file operations and configuration. The main area displays a table of configuration bits with columns for Address, Value, Category, and Setting. The "Checksum" is shown as "N/A".

Addr...	Value	Category	Setting
157FC	02E2	Primary Oscillator Select	HS Oscillator Enabled
		Primary Oscillator Output Function	OSCO pin has clock out function
		Clock Switching and Monitor	Sw Disabled, Mon Disabled
		Oscillator Select	Primary Oscillator (XT, HS, EC)
157FE	3F1F	Watchdog Timer Postscaler	1:32,768
		WDT Prescaler	1:128
		Watchdog Timer Enable	Disable
		Comm Channel Select	EMIC2/EMUD2 shared with PCG2/PGD2
		Set Clip On Emulation Mode	Reset Into Operational Mode
		General Code Segment Write Protect	Disabled
		General Code Segment Code Protect	Disabled
		JTAG Port Enable	Disabled



# 16-bit Development Tools MPLAB<sup>®</sup> IDE

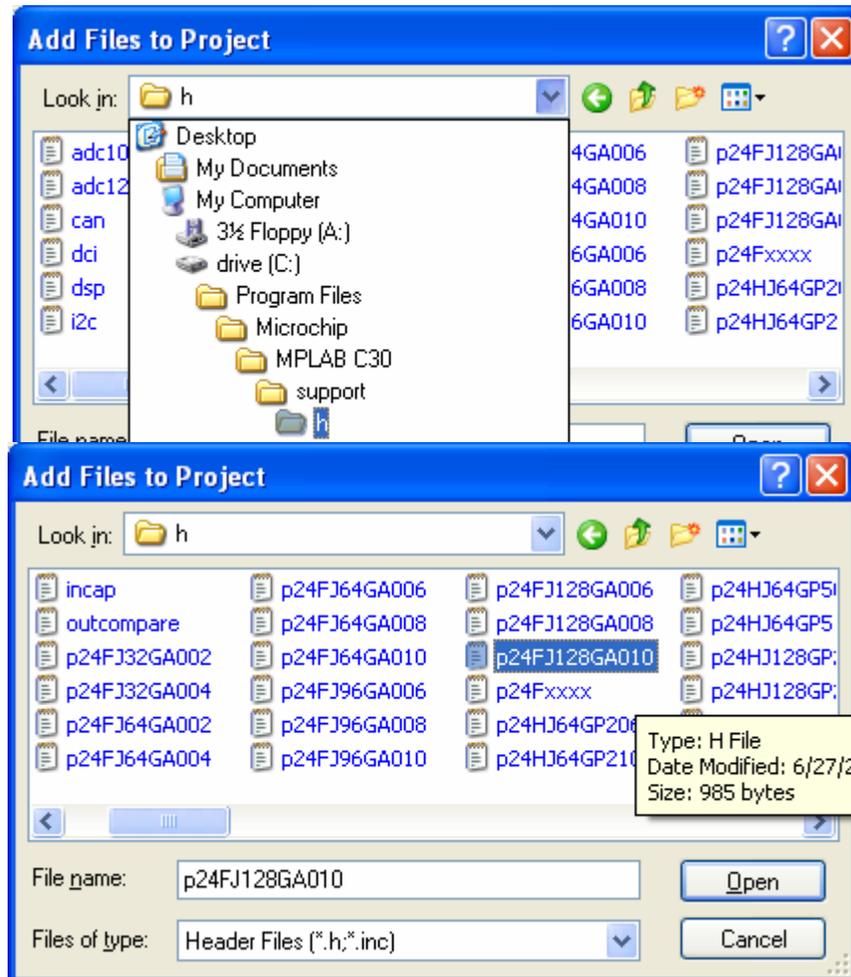
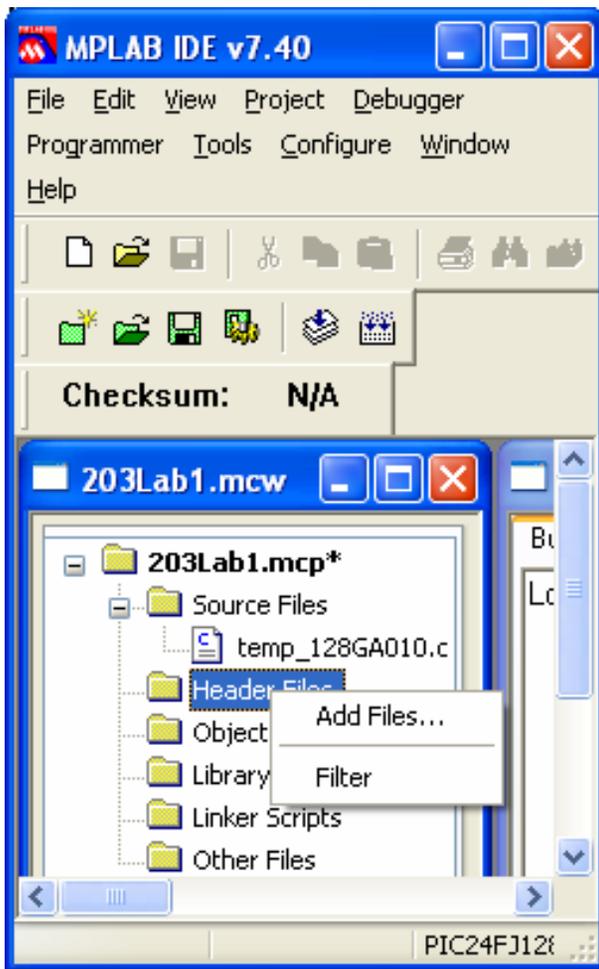
## Adding source files to the project





# 16-bit Development Tools MPLAB® IDE

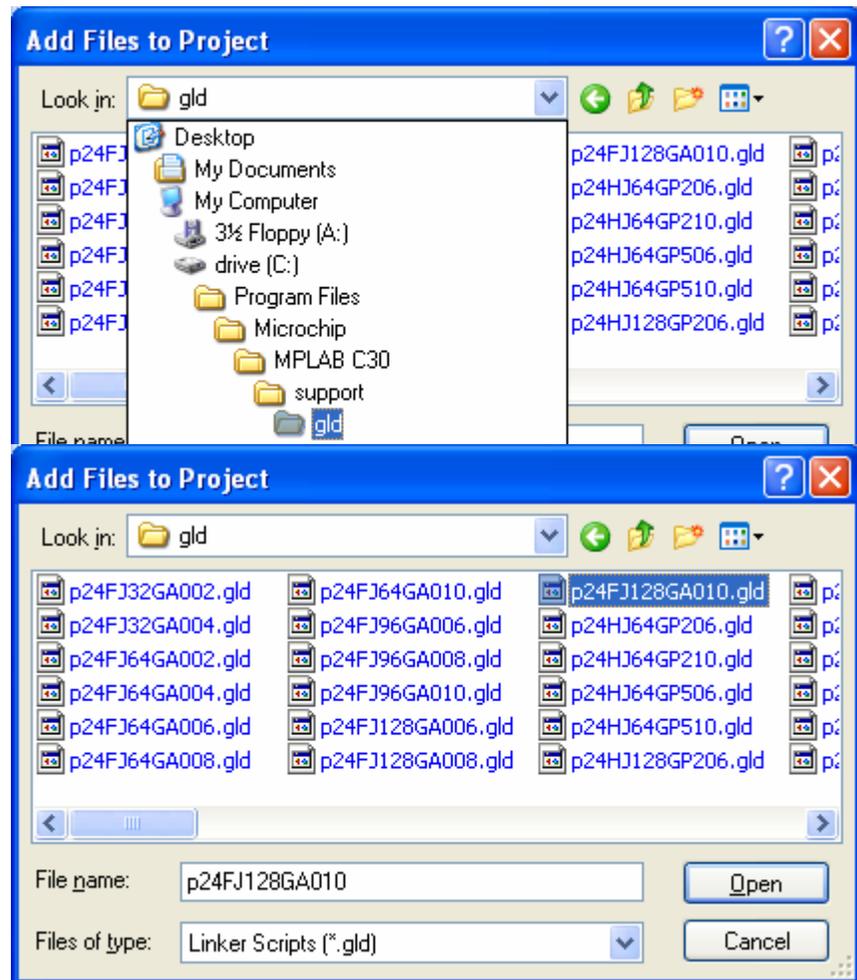
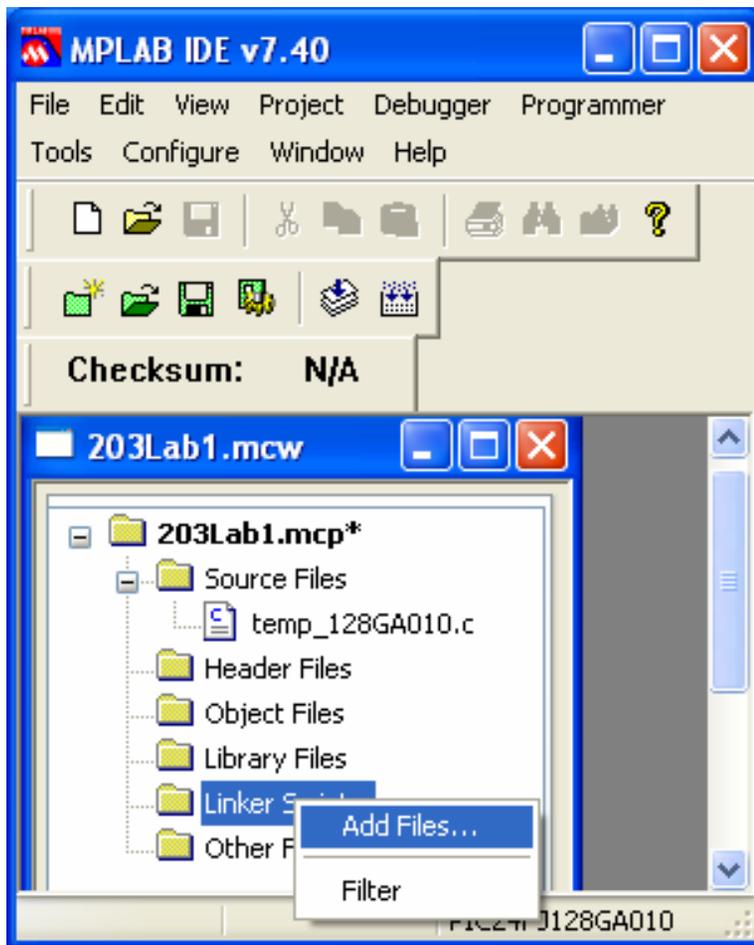
## Adding Header files to the project





# 16-bit Development Tools MPLAB® IDE

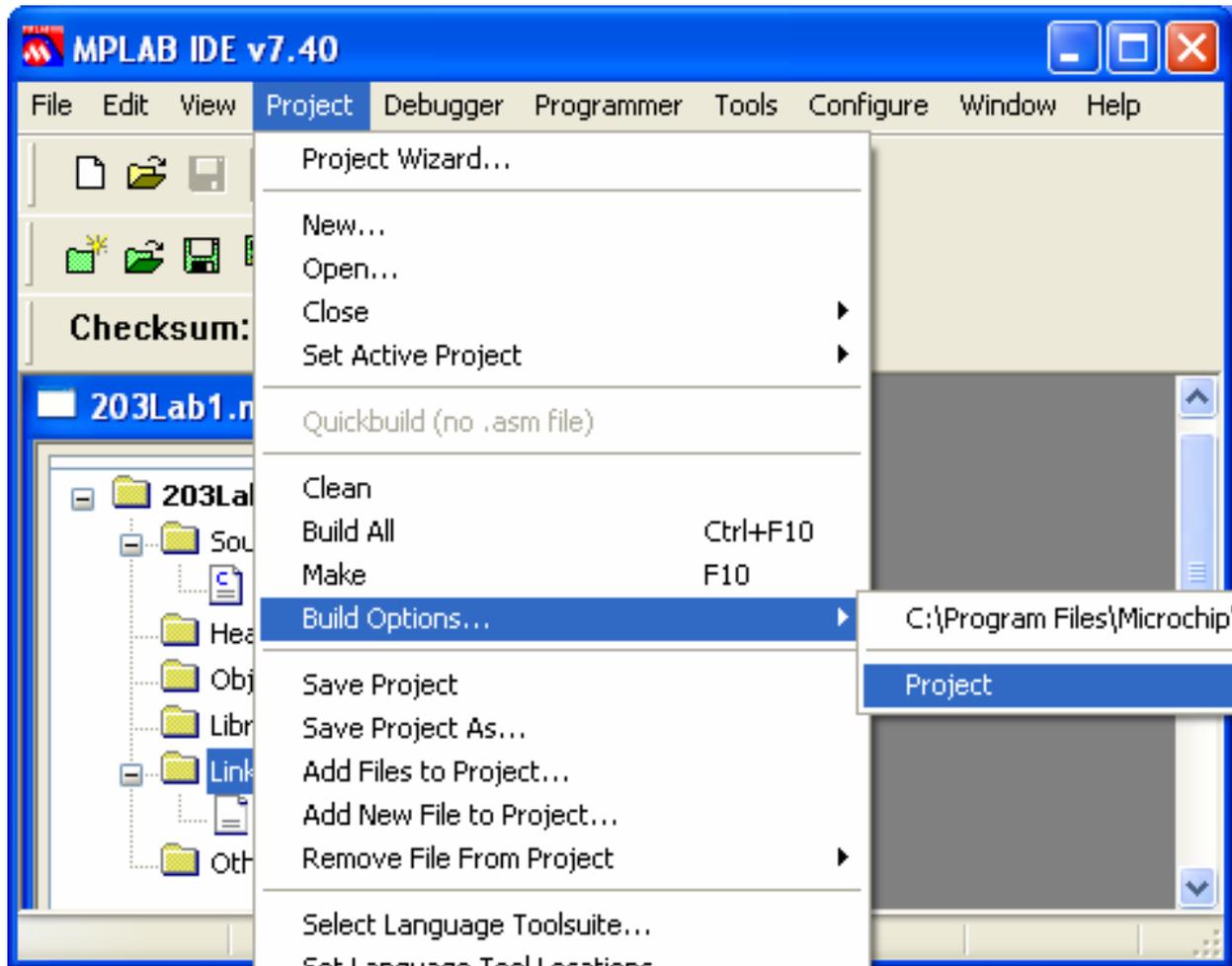
## Adding Linker file to the project





# 16-bit Development Tools MPLAB® IDE

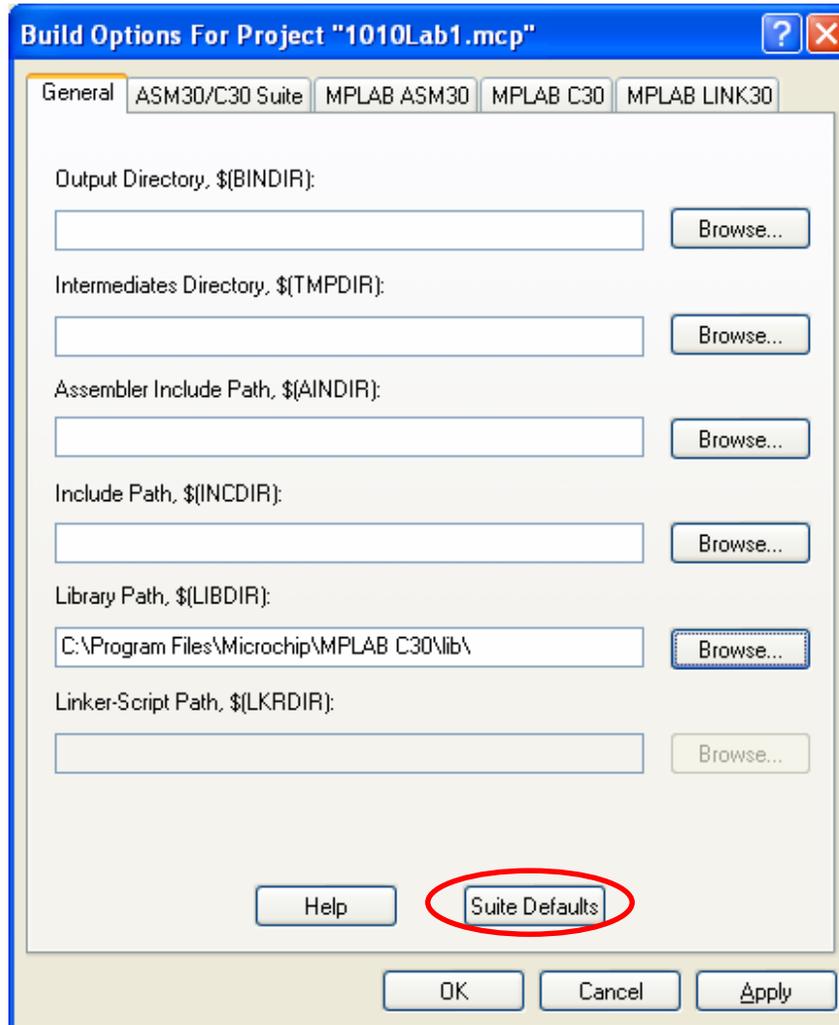
Setting the build Options for the project





# 16-bit Development Tools MPLAB<sup>®</sup> IDE

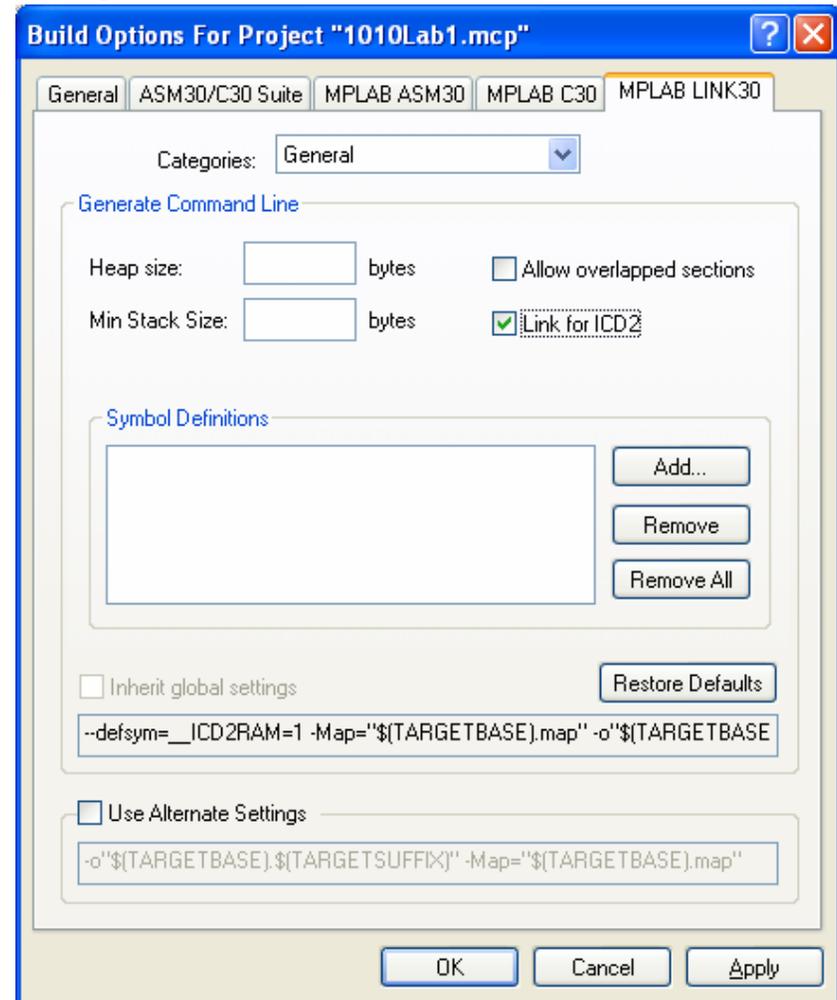
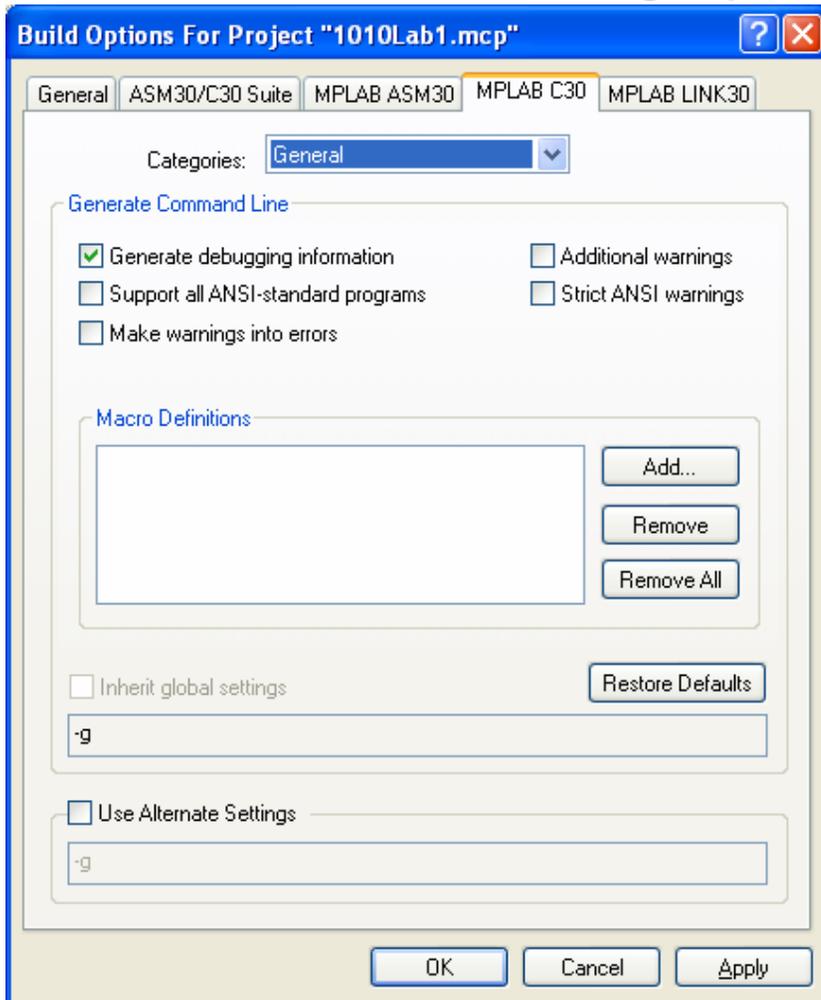
Set the Library paths...





# 16-bit Development Tools MPLAB<sup>®</sup> IDE

## Select General Category build option and Link for ICD2





# Attributes to Interrupt handlers

These are the some of the attributes to use the interrupt handlers

```
157
158 void __attribute__((__interrupt__)) _ADCInterrupt(void)
159 {
160
161
162 /* Interrupt Service Routine code goes here */
163
164 }
165
166
167 /* Interrupt Service Routine 2 */
168 /* Fast context save (using push.s and pop.s) */
169
170 void __attribute__((__interrupt__, __shadow__)) _T1Interrupt(void)
171 {
172
173 /* Interrupt Service Routine code goes here */
174
175 }
176
177
178 /* Interrupt Service Routine 3: INT0Interrupt */
179 /* Save and restore variables var1, var2, etc. */
180
181 void __attribute__((__interrupt__(__save__(variable1,variable2)))) _INT0Interrupt(void)
182 {
183
184 /* Interrupt Service Routine code goes here */
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

The predefined interrupt handler names can be found in Table 7.3 of C30 User's Guide, DS51284



# Attributes to define Variables

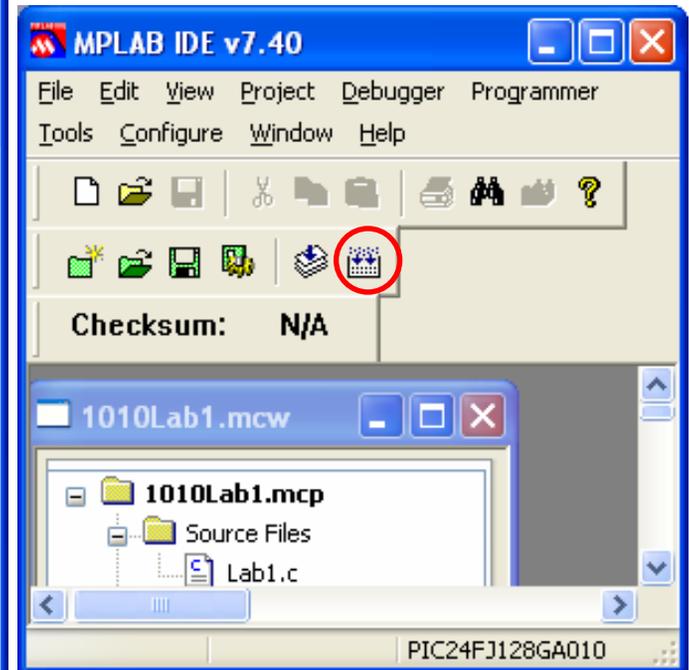
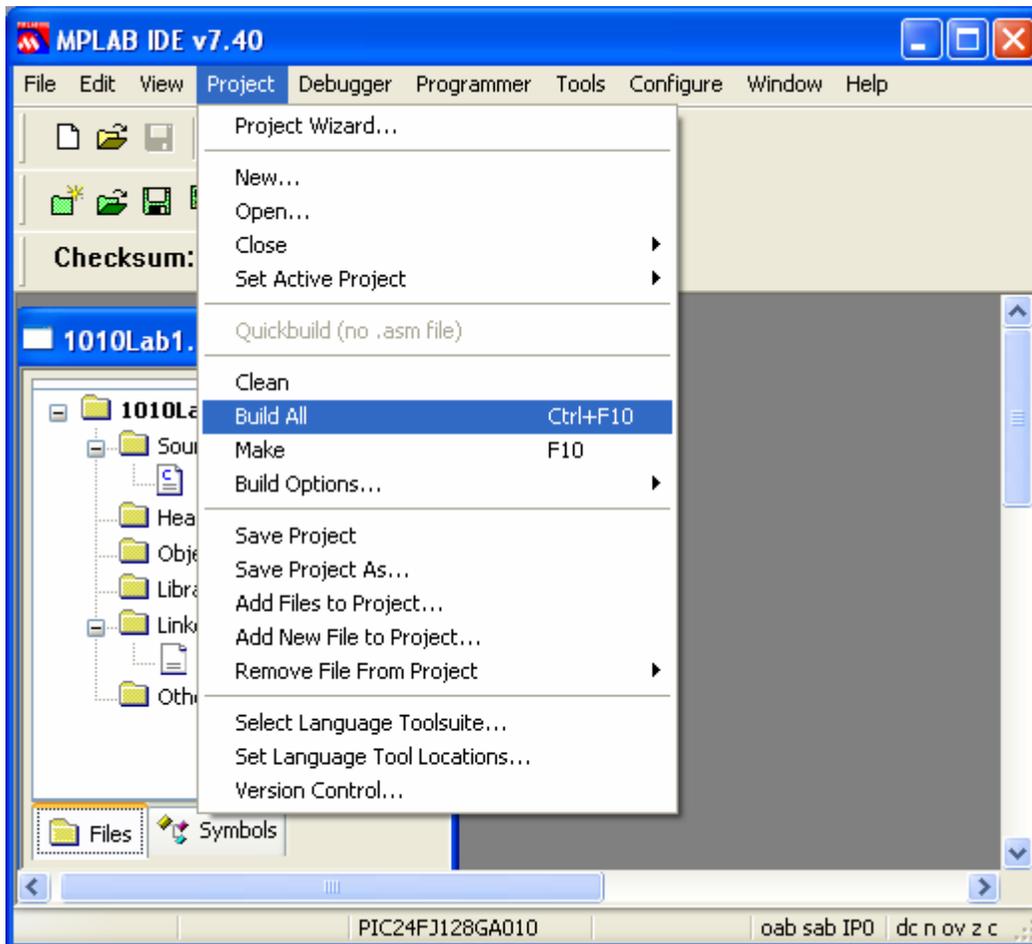
These are the some of the attributes to define variables

```
100  /****** START OF GLOBAL DEFINITIONS *****/
101
102
103  /* Define arrays: array1[], array2[], etc.
104  /* with attributes, as given below
105
106  /* either using the entire attribute
107  int array1[CONSTANT1] __attribute__((__space__(data), __aligned__(32)));
108
109  /* or using macros defined in the header file
110  int array3[CONSTANT1] _BSS(32);
111
112
113  /* Define arrays without attributes
114
115  int array5[CONSTANT2]; /* array5 is NOT an aligned buffer */
116
117  /* -----
118
119  /* Define global variables with attributes
120
121  int variable1 __attribute__((__space__(data)));
122
123  /* Define global variables without attributes
124
125  int variable3;
126
```



# PIC24: Development Tools MPLAB<sup>®</sup> IDE

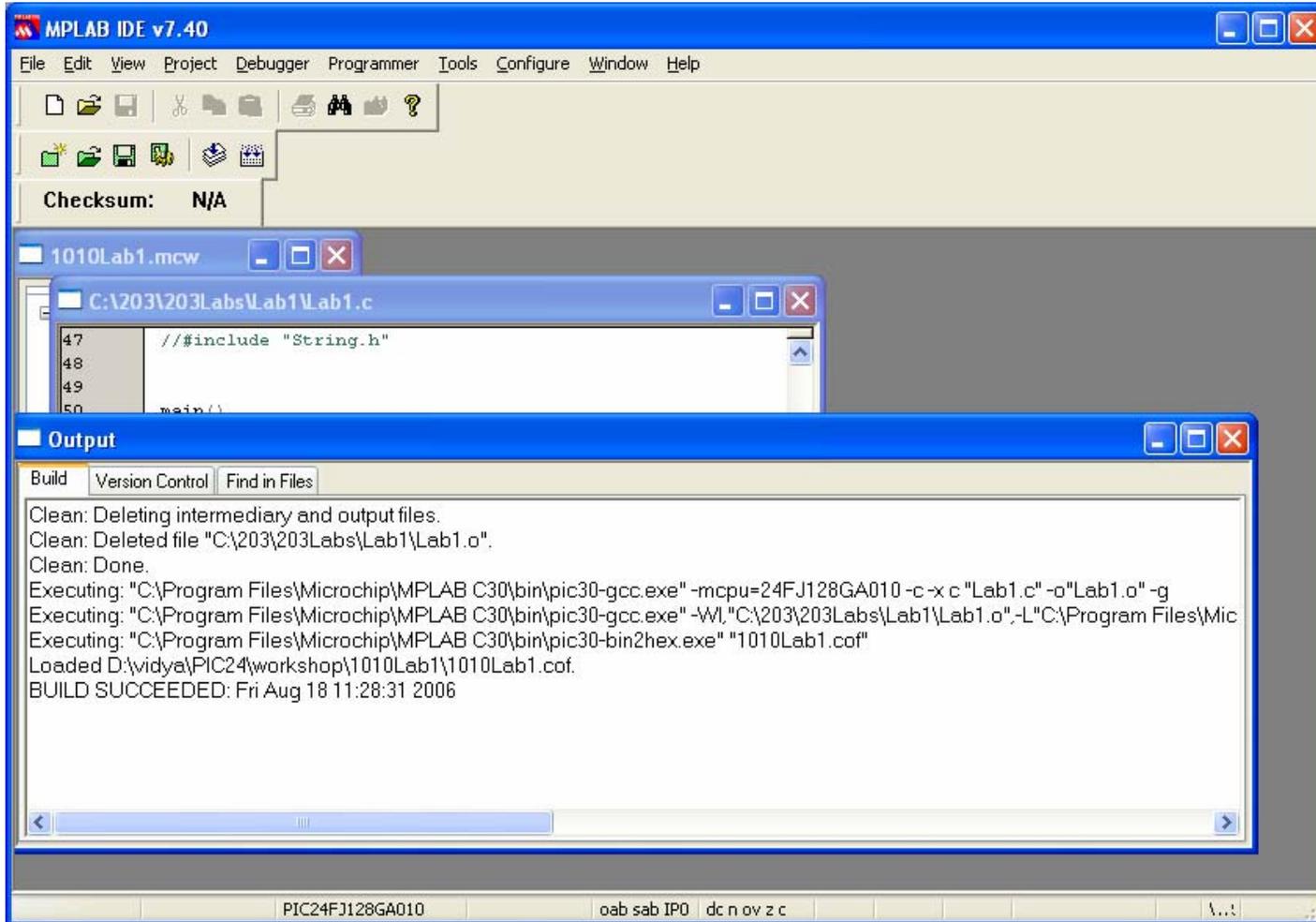
## Building the project





# PIC24: Development Tools MPLAB® IDE

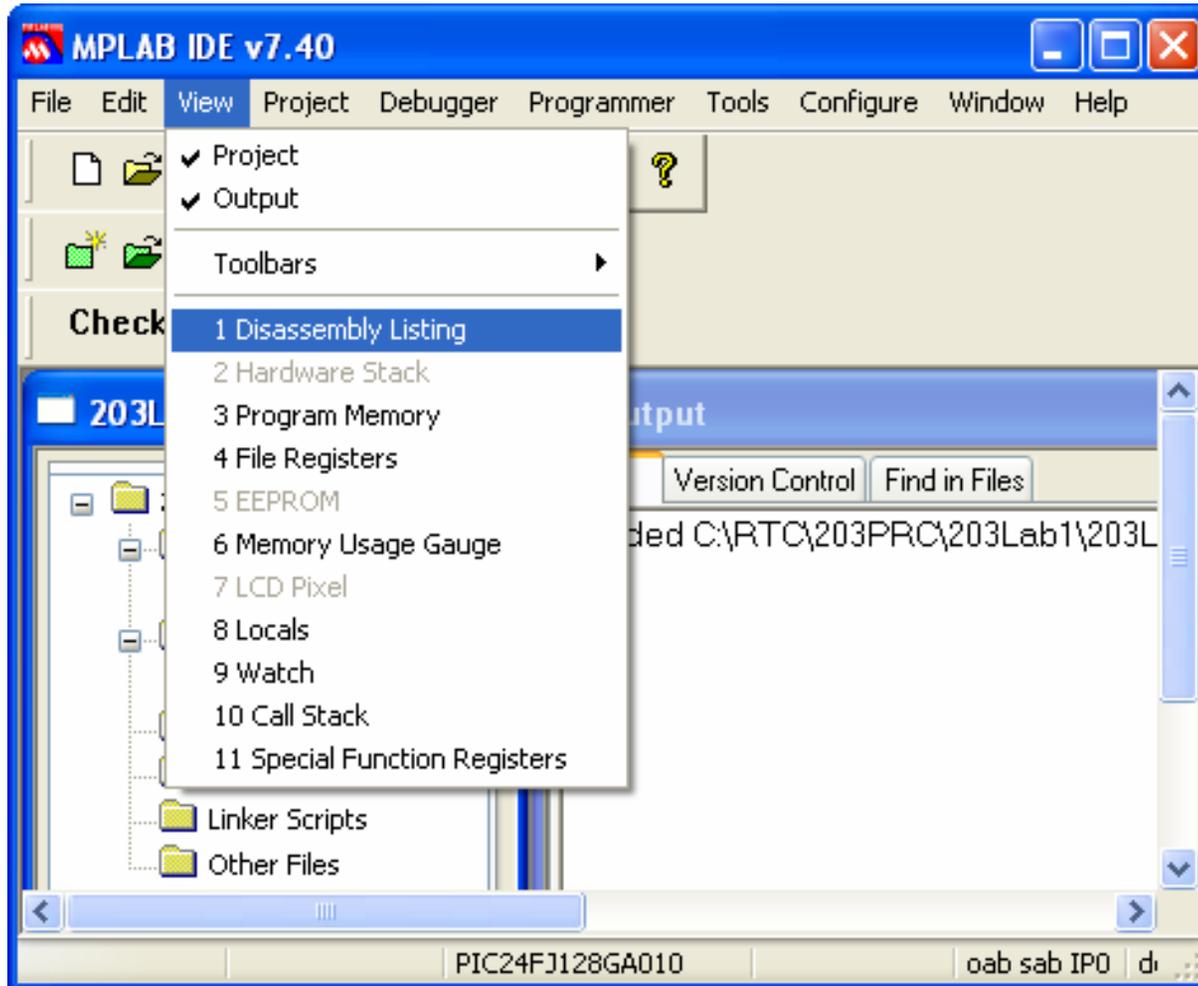
Output window after a successful build!!!





# 16-bit: Development Tools MPLAB® IDE

Viewing disassembly listing many more...





# 16-bit: Development Tools MPLAB<sup>®</sup> IDE

Snap shot of a disassembly listing

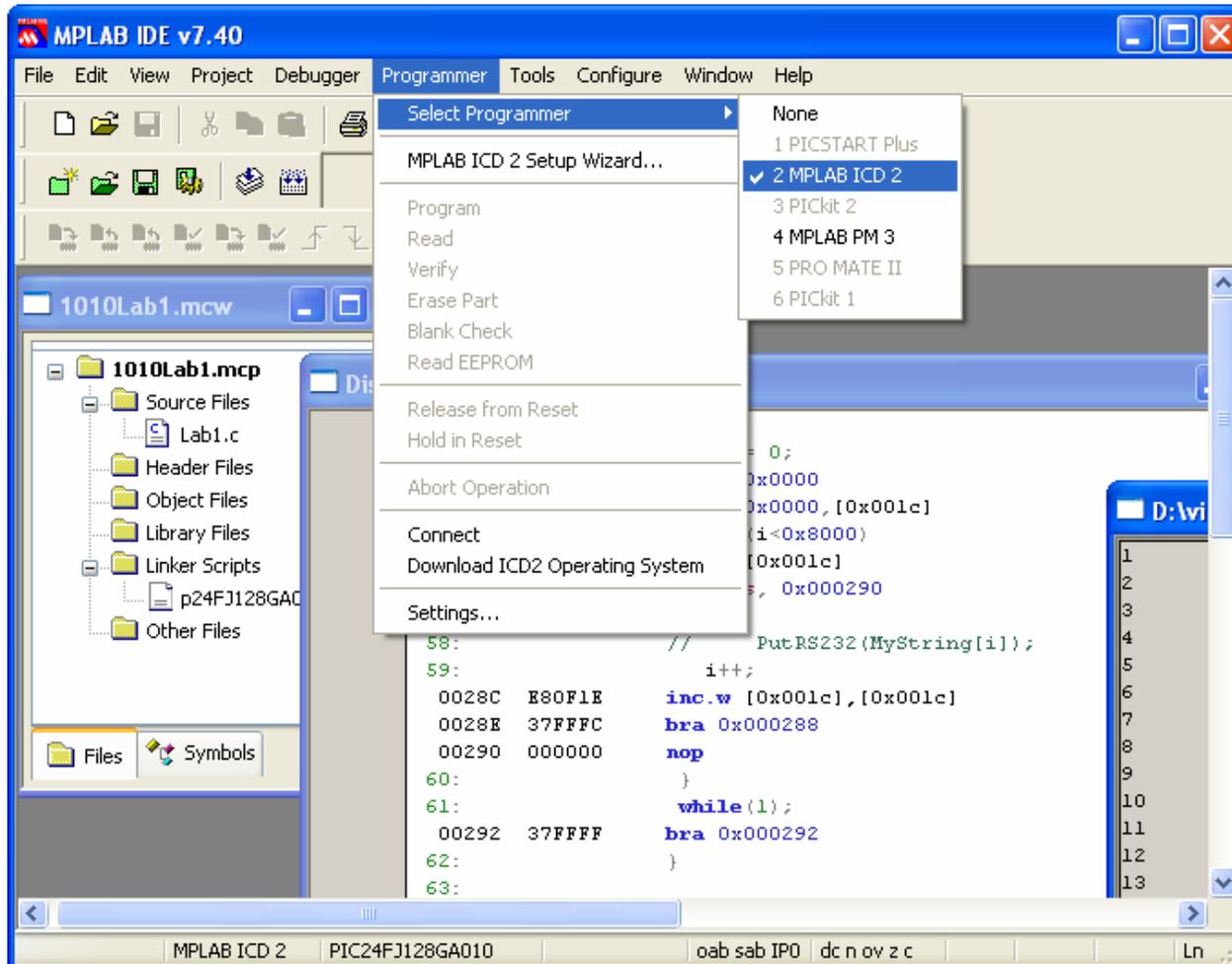
The screenshot shows the MPLAB IDE v7.40 interface. The main window displays a disassembly listing for a PIC24FJ128GA010. The listing shows assembly code with addresses, hex values, and mnemonics. The code includes a main function with various instructions like `lnk #0x2`, `rcall 0x000294`, `clr.w 0x0000`, `mov.w 0x0000, [0x001c]`, `while (i < 0x8000)`, `cp0.w [0x001c]`, `bra lts, 0x000290`, `inc.w [0x001c], [0x001c]`, `bra 0x000288`, and `nop`.

```
50:      main()
51:      {
00280  FA0002  lnk #0x2
52:      //   RS232Init();           //Th
53:      PSVInit();
00282  070008  rcall 0x000294
54:
55:      int i = 0;
00284  EB0000  clr.w 0x0000
00286  780F00  mov.w 0x0000, [0x001c]
56:      while (i < 0x8000)
00288  E0001E  cp0.w [0x001c]
0028A  350002  bra lts, 0x000290
57:      {
58:      //   PutRS232(MyString[i]);
59:      i++;
0028C  E80F1E  inc.w [0x001c], [0x001c]
0028E  37FFFC  bra 0x000288
00290  000000  nop
60:      }
61:      }
```



# 16-bit: Development Tools MPLAB<sup>®</sup> IDE

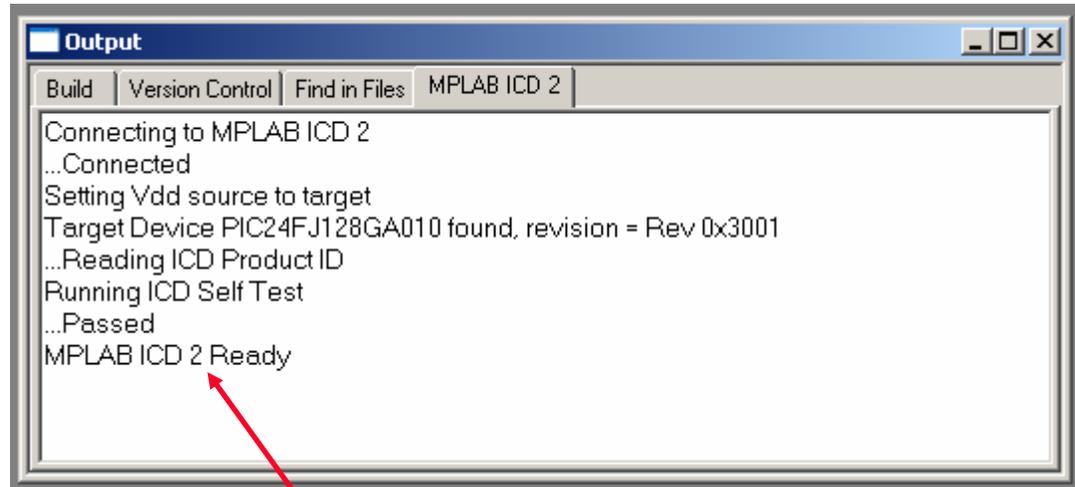
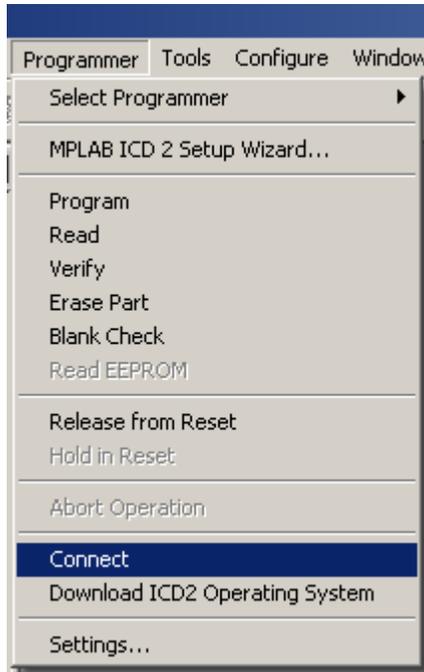
## Selecting the programmer





# 16-bit: Development Tools MPLAB<sup>®</sup> IDE

## Connect to the Programmer

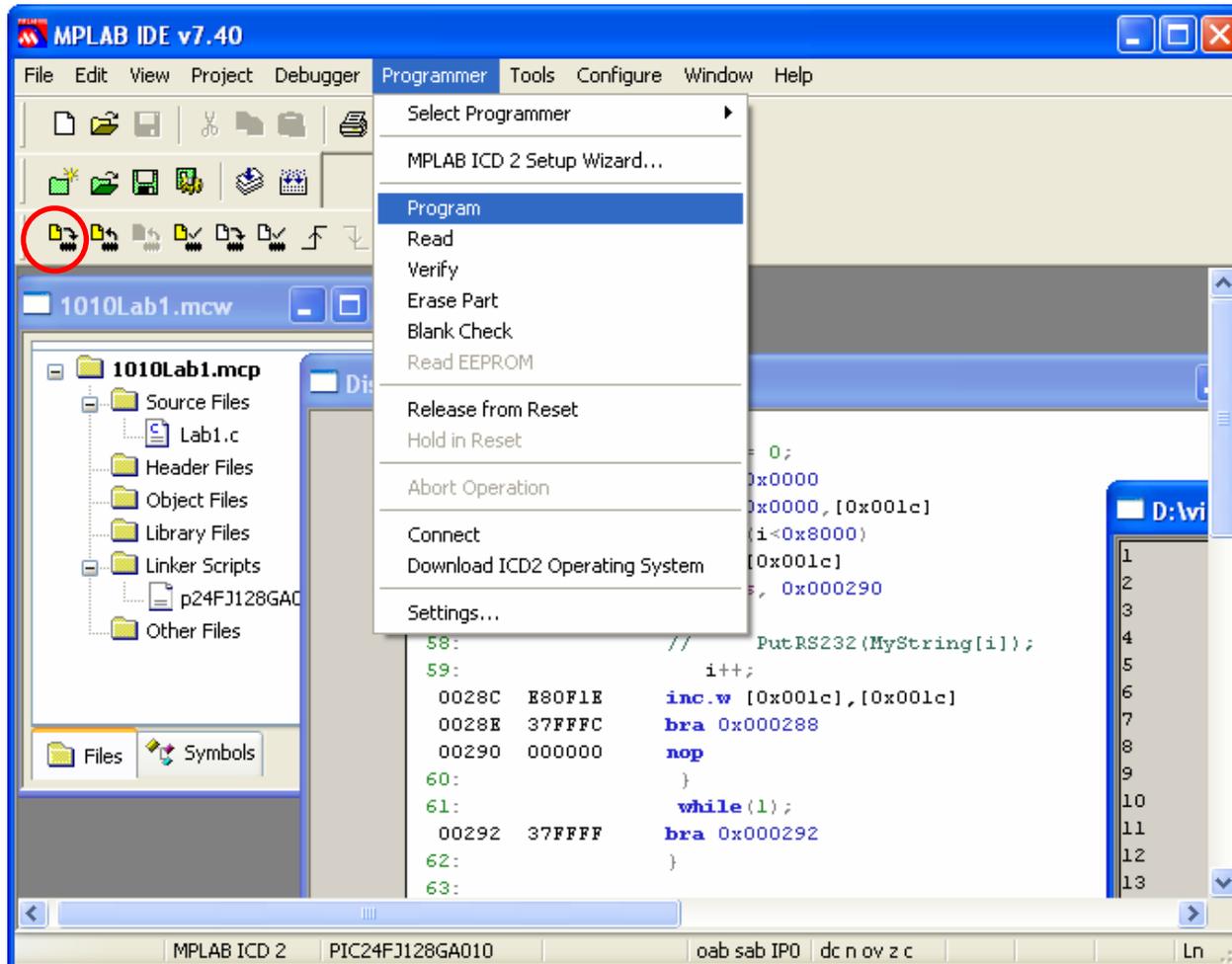


**Successful Connection**



# 16-bit: Development Tools MPLAB® IDE

Program the device



# HANDS-ON

# Training

## C30 Attributes and Built-in Functions





# C30: Data Attributes

- **Attributes used to:**
  - Place data at a specific address
  - Place into near memory
  - Place in a specific section or space
  - Align begin or end to a multiple byte boundary
  - Allow linker to fill memory gaps
  - Maintain data through reset



# C30: Data Attributes

- **Place data at a specific address**

```
int MyVar __attribute__((address(0x860)));
```

```
int __attribute__((address(0x860))) MyVar2;
```

- Be careful about overlap and auto variables

- **Place into near memory (below 8kB)**

```
char __attribute__((near)) MyVar3, MyVar4;
```



# C30: Data Attributes

- Place in a specific section or space

```
int Abc __attribute__((section("ThePlace")));  
int Def __attribute__((space(psv)));
```

- New section will be created, if needed

- Align begin or end to a multi-byte boundary

```
char __attribute__((aligned(32))) MyArray[18];  
int __attribute__((reverse(64))) EndArray[25];
```



# C30: Data Attributes

- Allow linker to fill memory gaps

```
char __attribute__((unordered)) MidArray[15];
```

- Allows linker to move variable

- Maintain data through reset

```
int __attribute__((persistent)) PrevData[5];
```

- C30 startup code will not initialize to zero
- Can maintain state information through reset
- Not for power cycle



# C30: Built-in Functions

- **Use special hardware features**

- Multiply `__builtin_mulss(const signed int p0, const signed int p1);`
- Divide `__builtin_divsd(const long num, const int den);`
- Table pointer `__builtin_tblpage(const void *p);`  
`__builtin_tbloffset(const void *p);`
- PSV `__builtin_psvpage(const void *p);`  
`__builtin_psvoffset(const void *p);`
- Bit toggle `__builtin_btg(unsigned int *, unsigned int 0xn);`

- **More efficient code**

- Faster
- Smaller

# HANDS-ON

# Training

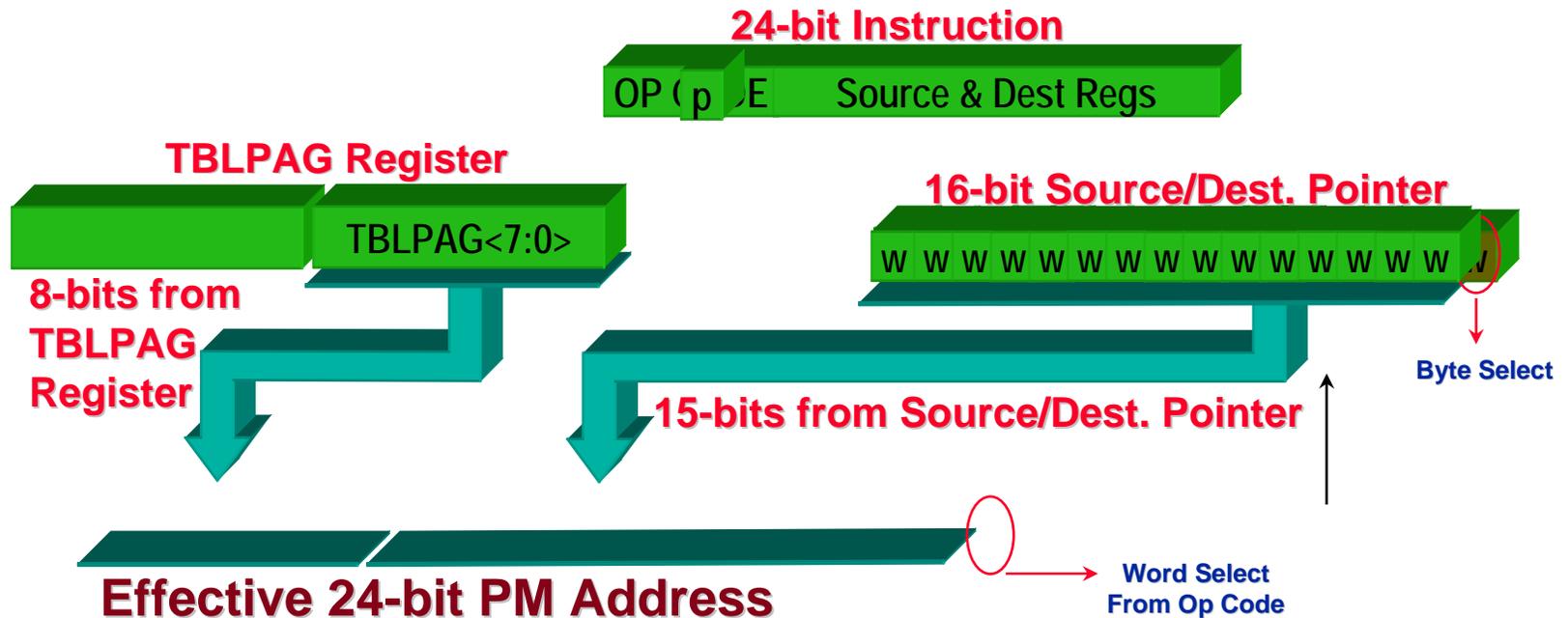
# Program Memory Data Memory Interface





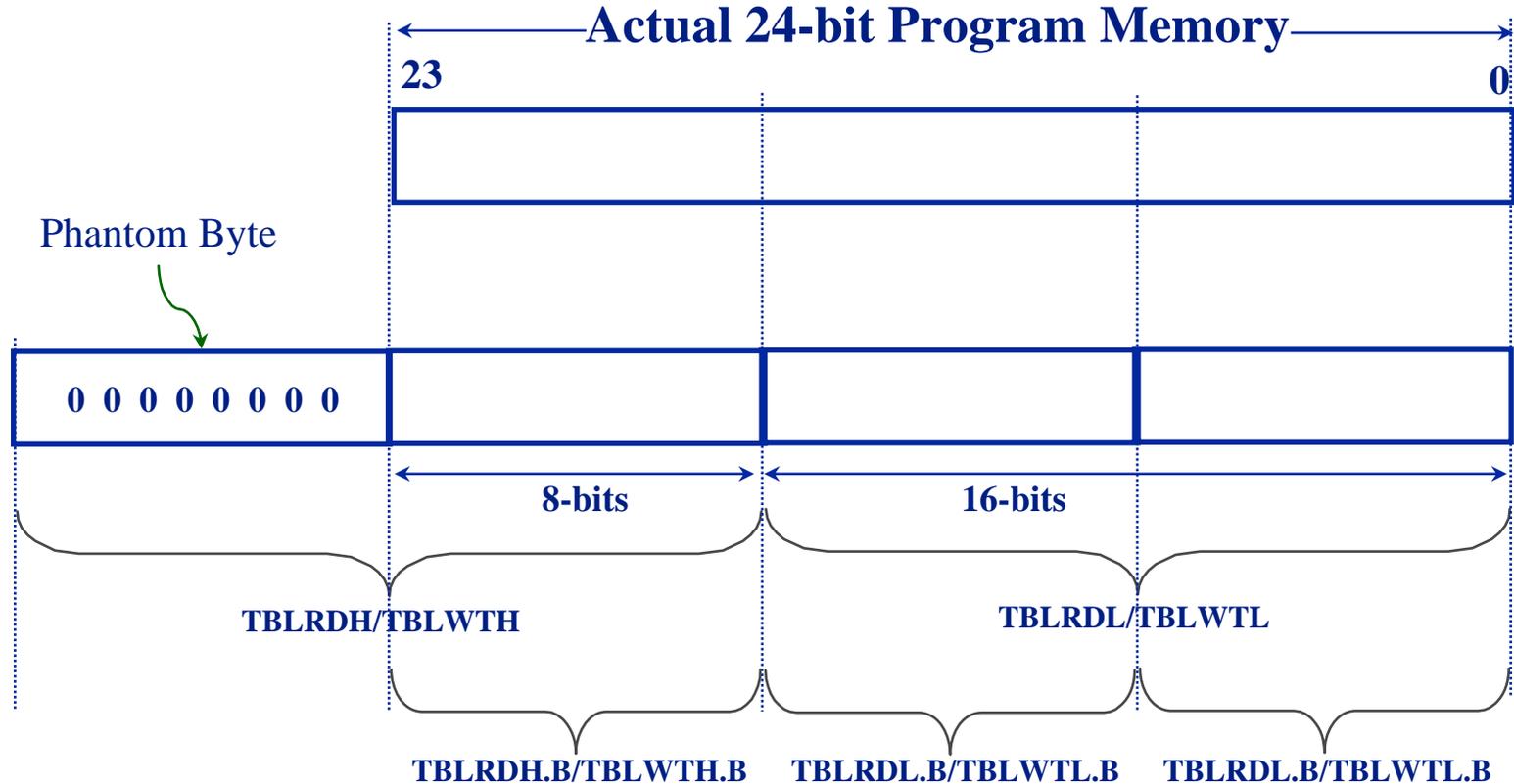
# Constant Data Access from PM Using **Table Instructions**

- **24-bit Effective Address(EA) formation**
  - Configuration space is accessed by setting TBLPAG<7> (i.e. EA<23>)
    - Only means of accessing configuration space





# Constant Data Access from PM Using **Table Instructions**

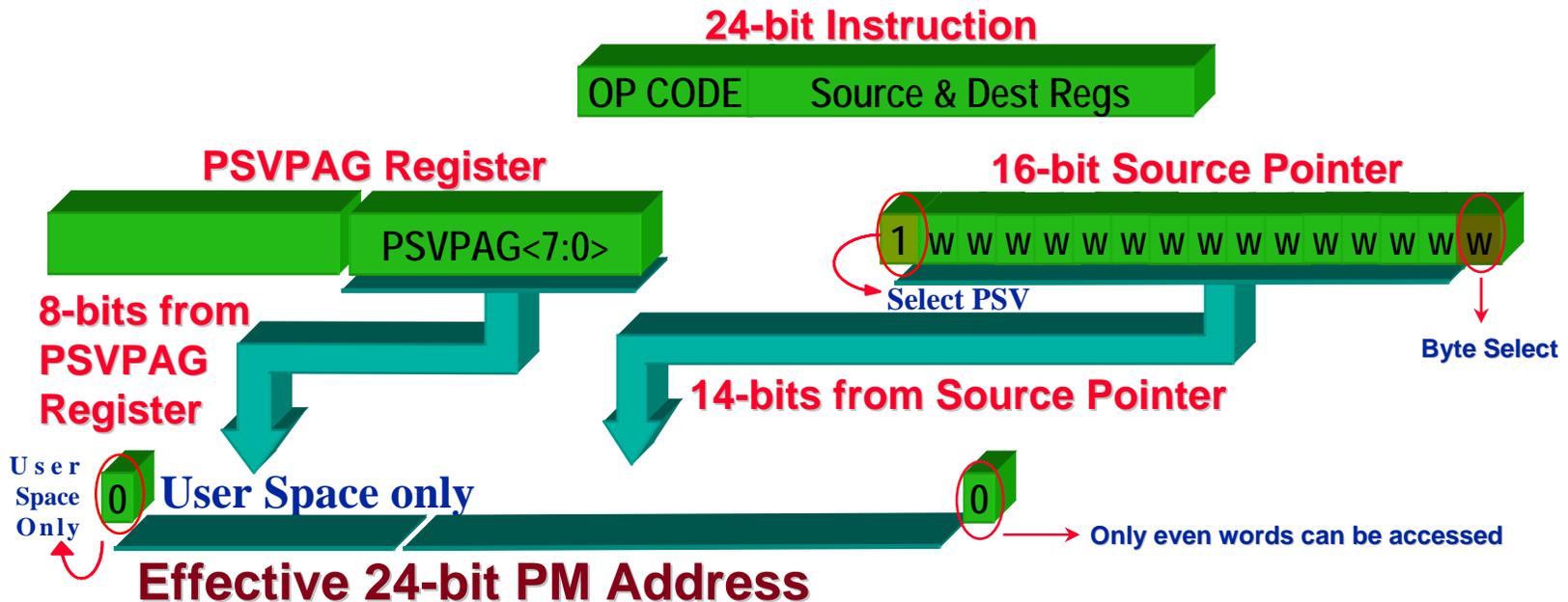


**Table Instruction View of Program Memory**



# Constant Data Access from PM Using Program Space Visibility

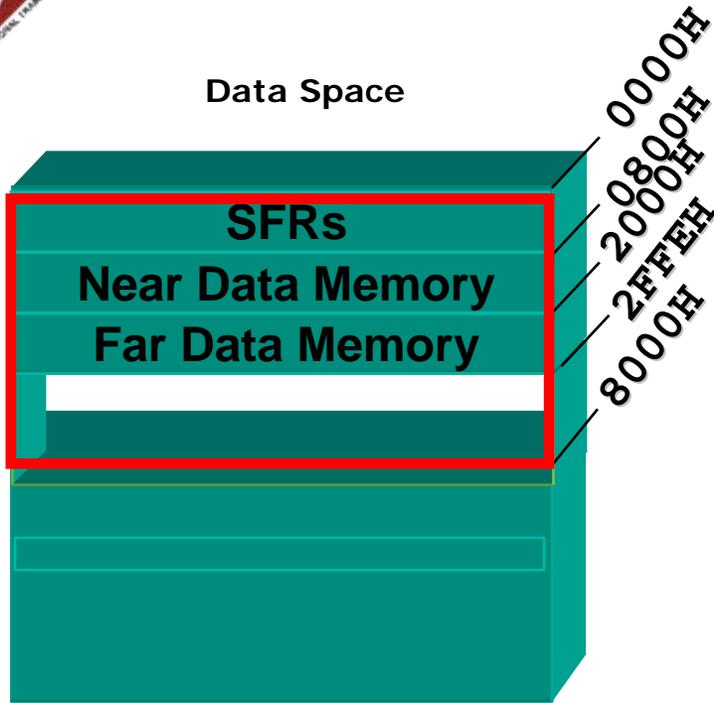
- **32 KB segment of PM may be mapped into the data memory address space**
  - If PSV bit (CORCON<2>) is set and if SrcPointer (DM EA)<15> is '1' then data is accessed from PM





# PSV Addressing Example

Data Space

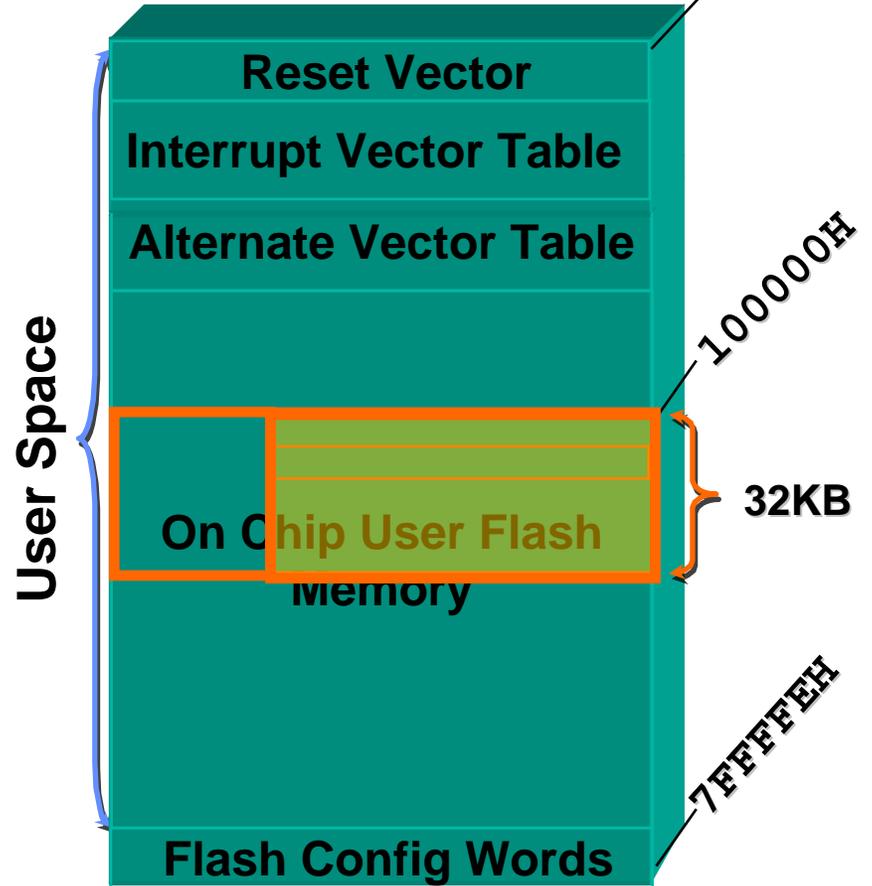


CORCON<PSV> = 1

PSVPAG = 0x20

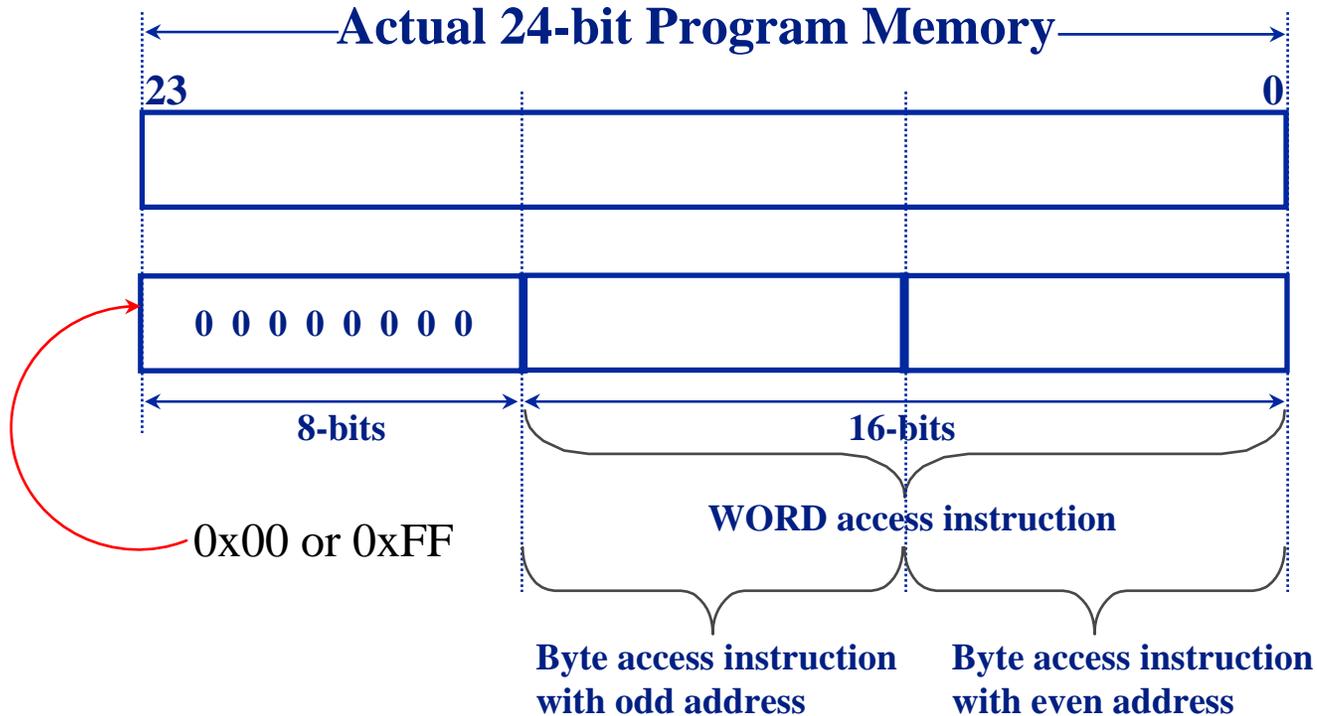
DataEA = 0x9000

ProgEA = 0x101000





# Data Access from PM Using PSV



- Only lower 16-bits of PM location can be accessed via mapping

- Upper 8 bits should be programmed for a NOP



# Defining Constants for PSV

## **;**Define constants in PM

- `char ConstData[] __attribute__((space(psv))) = {1, 2, 3};`

## What is the “**space(psv)**” attribute?

- Read only section dedicated for use in PSV window
- Linker will not allow read-only section to cross PSVPAG boundary
  - **Boundary = 32 KB**

## When the linker assigns PSV space, how to load PSVPAG

- `PSVPAG = __builtin_psvpage(ConstData);`



# Advantages of PSV

- **PSV allows very large tables of data to be stored and accessed quickly & efficiently**
- **PSV provides a bridge to a common data/program address space (Von Neumann) but only when needed**
- **Example:**
  - Large constant data for display
  - Sine Table

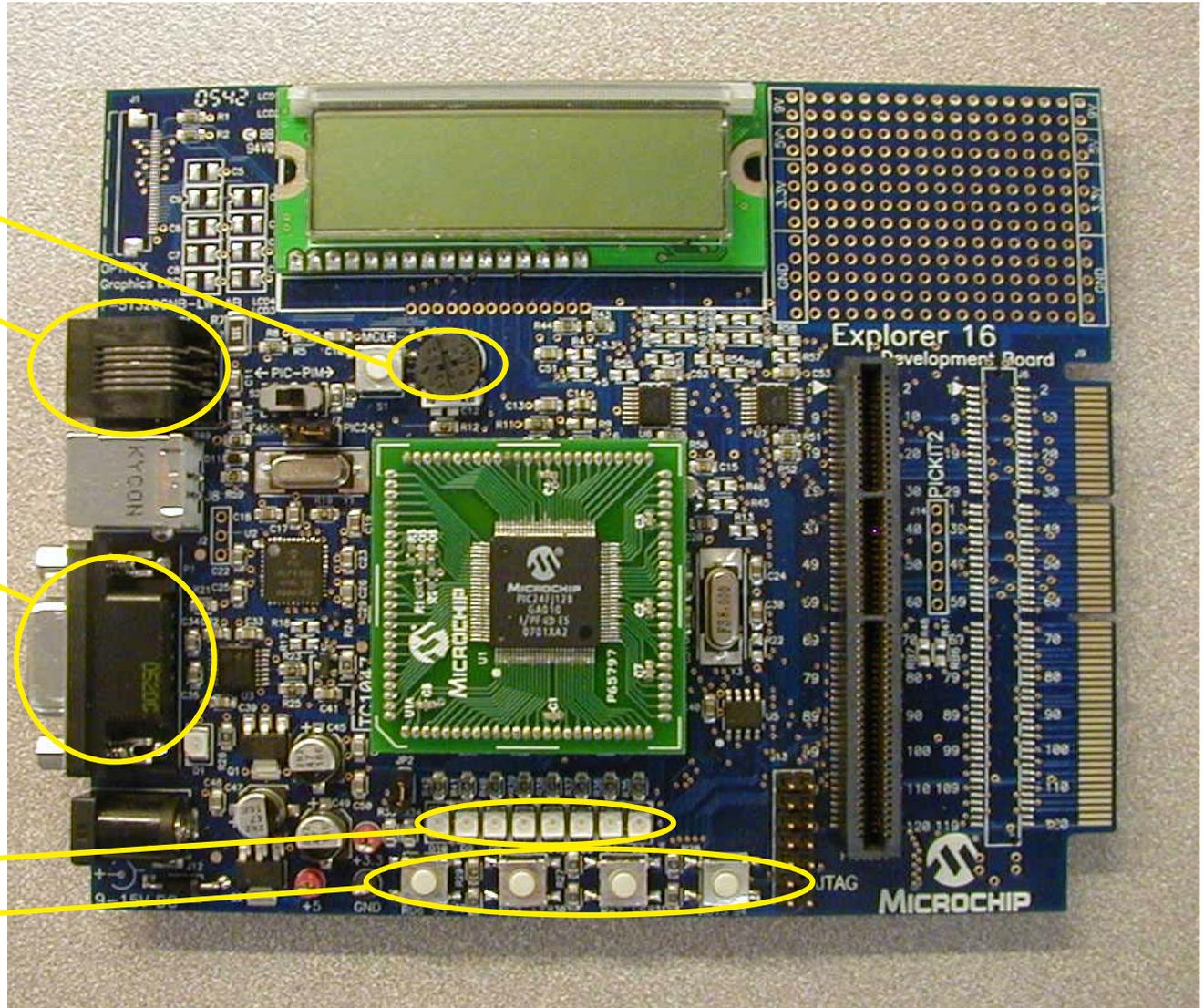


# Data Access Overhead Using PSV

- **PSV data fetch overhead:**
  - Outside a REPEAT loop:
    - **Data move ops, overhead = 1 cycle**
    - **ALU based ops, overhead = 2 cycles**
  - Within a REPEAT loop:
    - **Data pipelined, so overhead for all ops = 0 cycles**
    - **First & last iteration - data pipeline fill/flush**
      - Data move ops, overhead = 1 cycle
      - ALU based ops, overhead = 2 cycles



# Lab 2: Working with PSV



POT

ICD2 connector

RS232 connector

LEDs

Switches



# Lab 2: Working with PSV

- **Goals:**

- To initialize PSV
- To store a Hello World string in PSV space
- To read and transmit this string

- **To Do:**

- Look into the Hand out provided

- **Expected Result:**

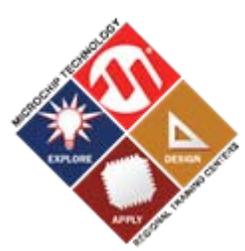
- The text file captured using HyperTerminal should match the array defined

# HANDS-ON

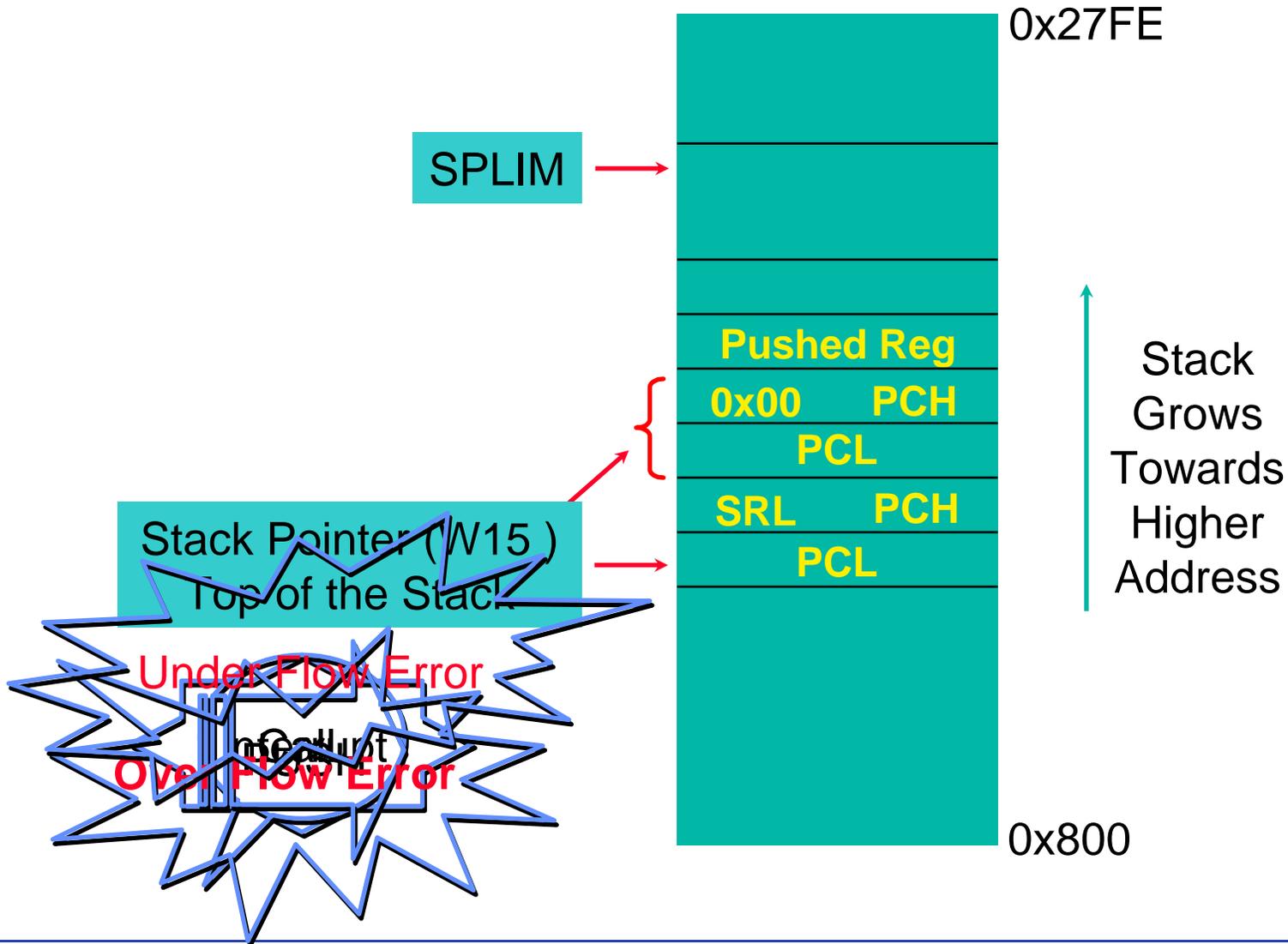
# Training

# Software Stack



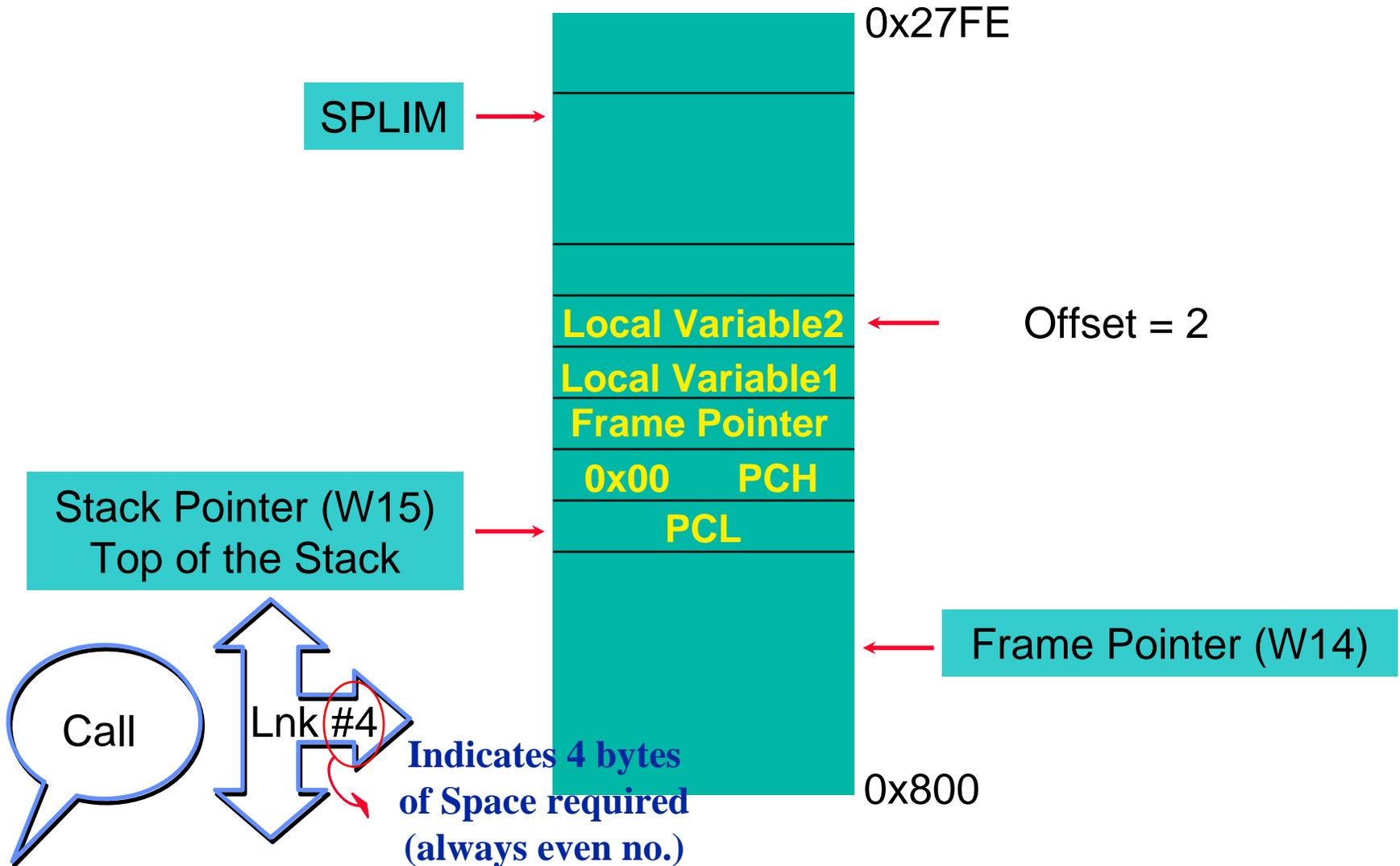


# Software Stack in Data RAM





# Frame Pointer



# HANDS-ON

# Training

## Addressing Modes





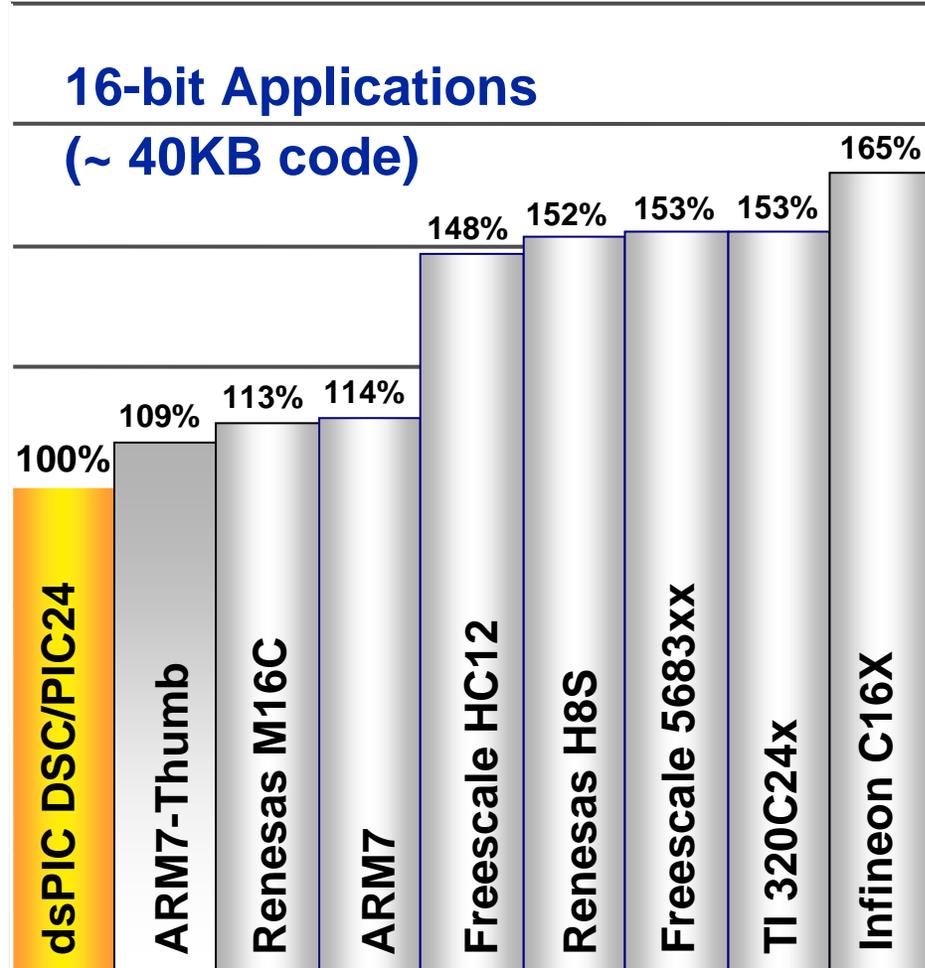
# Highly Optimized C Compiler

EEMBC industry std. Benchmarks, Automotive Suite

## Relative Code Size

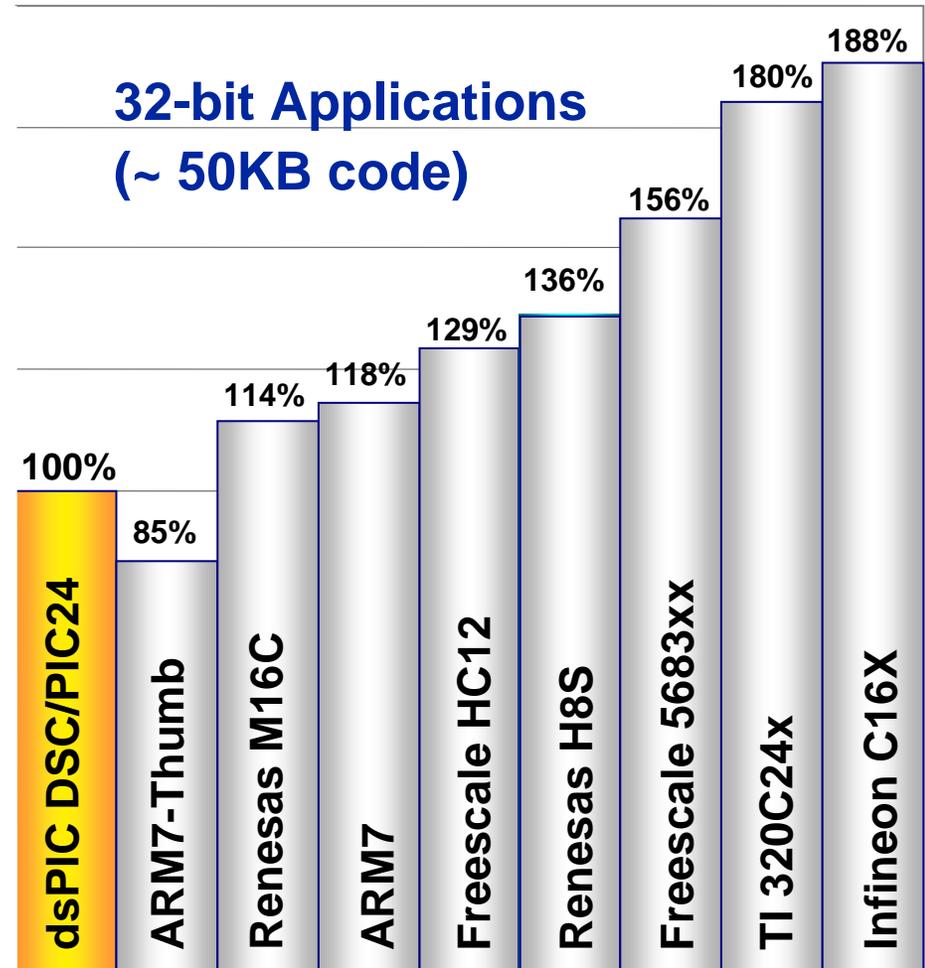
### 16-bit Applications

(~ 40KB code)



### 32-bit Applications

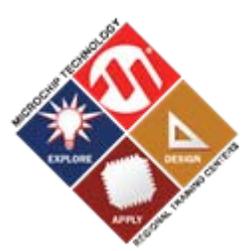
(~ 50KB code)





# File Register Addressing

- **“File Register” is a location in Data Memory**
- **Memory address operand (13-bit) encoded in the instruction**
  - Access **“Near” (first 8 KB)** of Data Space
- **Useful to access Special Function Registers (SFRs)**



# File Register Addressing

- **Example:**

```
main( )
```

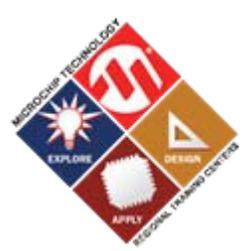
```
{
```

```
    U1STA = xyz;
```

```
}
```

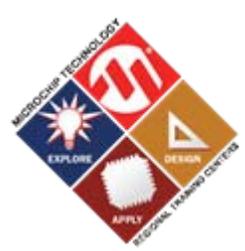
```
mov xyz, W0
```

```
mov W0, U1STA
```



# Register Direct Addressing

- **Accesses the contents of W0 - W15**
  - Supports both byte and word accesses
    - **Byte accesses will use the lower byte of the specified W register**
  - Useful when data already resides in W-array
  - This addressing mode available to all instructions



# Register Direct Addressing

- **Example:**

```
int x, y, z;
```

```
main()
```

```
{
```

```
    x = y + z;
```

```
}
```

```
mov y, w1
```

```
mov z, w0
```

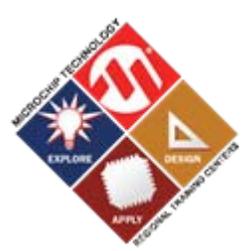
```
add w0, w1, w0
```

```
mov w0, x
```



# Register Indirect Addressing

- **W register used as a pointer to memory**
  - The contents of a working register form an Effective Address (**EA**)
  - Provides accessibility to entire 64 KB of addressable memory
  - Supported by most instructions
    - Allow indirect byte-wide accesses
  - The source operand and the destination can be indirectly addressed in an instruction



# Register Indirect Addressing

- **Example:**

```
int *p, *q, r;
```

```
main()
```

```
{
```

```
    *p = *q + r;
```

```
}
```

```
mov p, W2
```

```
mov q, W1
```

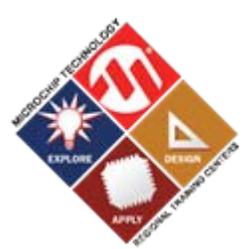
```
mov r, W0
```

```
add W0, [W1], [W2]
```



# Register Indirect Addressing with Post/Pre-Modification

- **Instruction accesses data first and then updates the Pointer, in the same cycle**
  - Effective Address (EA) is the contents of the W register used as a pointer
  - Post-modifies the address pointer by
    - 1 in byte-mode instructions
    - 2 in word-mode instructions
  - Post/Pre-modification can be applied to both source and destination pointers



# Register Indirect Addressing with Post/Pre-Modification

- **Example:**

```
long int l, m, *n;
```

```
main()
```

```
{
```

```
    l = m + *n;
```

```
}
```

```
mov n, W0
```

```
mov m, W4
```

```
mov m_high, W5
```

```
add W4, [W0++], W2
```

```
addc W5, [W0--], W3
```

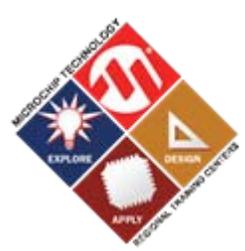
```
mov W2, l
```

```
mov W3, l_high
```



# Immediate Addressing

- **A literal operand is specified in the instruction**
  - Literal may be up to 16 bits depending on the instruction
    - **MOV instructions allow a 16-bit literal**
  - Literal may be supplied in both byte and word mode of the instructions
  - Literal operand is available to most instructions



# Immediate Addressing

- **Example:**

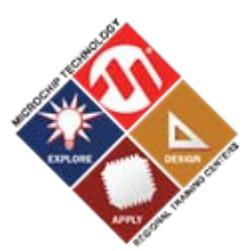
```
int x, y;  
main()  
{  
    x = y + 0x500;  
}
```

```
mov y, W0  
mov #0x500, W1  
add W0, W1, W0  
mov W0, x
```



# Register Indirect Addressing with Register Offset

- **EA is formed by adding two W registers**
  - Contents of the two W registers do not change
  - Useful in stepping through tables stored in memory



# Register Indirect Addressing with Register Offset

- **Example:**

```
const char a[];
```

```
char b, c;
```

```
main()
```

```
{
```

```
    c = a[b];
```

```
}
```

```
mov a, w1
```

```
mov.b b, w0
```

```
se.b w0, w0
```

```
mov.b [W1+W0], w0
```

```
mov.b w0, c
```



# Register Indirect Addressing with Literal Offset

- **Effective Address is formed by adding the contents of a W register with an immediate literal operand**
  - Contents of the W register does not change
  - Only allowed in the **MOV** instruction
  - The literal offset has the following range:
    - **Even values in the range [-1024, 1022], when using **MOV** instruction in word mode**
    - **Any value in the range [-512, 511] when using **MOV** instruction in byte mode**



# Register Indirect Addressing with Literal Offset

- **Example:**

```
main()  
{  
    long int l, m, *n;  
    l = m + *n;  
}
```

```
mov [W14+8], W0  
mov [W14+4], W2  
mov [W14+6], W3  
add W2, [W0++], [W14++]  
addc W3, [W0--], [W14--]
```

# HANDS-ON

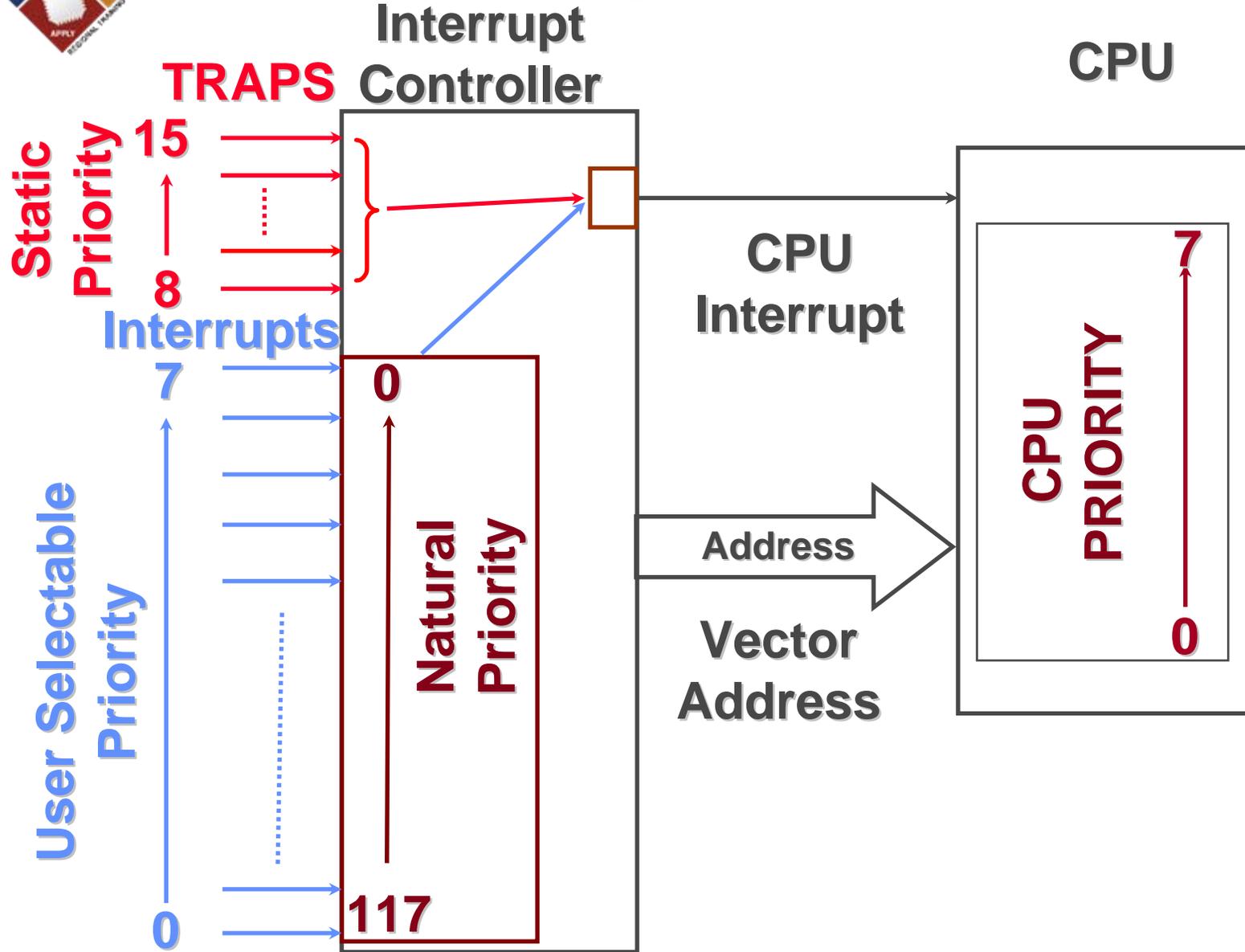
# Training

## Traps and Interrupts





# Interrupt Architecture





# IVT / AIVT

Highest  
Priority



Lowest  
Priority

0x00	RESET Vector	0x100
0x02	Also used for RESET	0x102
0x04	Reserved	0x104
0x06	Osc. Fail Trap Vector	0x106
0x08	Adrs. Err. Trap Vector	0x108
0x0A	Stack Err. Trap Vector	0x10A
0x0C	Math Err. Trap Vector	0x10C
0x0E	Reserved	0x10E
0x10	Reserved	0x110
0x12	Reserved	0x112
0x14	Interrupt 0 Vector	0x114
0x16	Interrupt 1 Vector	0x116
⋮	⋮	⋮
⋮	⋮	⋮
⋮	⋮	⋮
0xFC	Interrupt 116 Vector	0x1FC
0xFE	Interrupt 117 Vector	0x1FE



# AIVT Advantage

- IF “condition1” is true goto “application”
- Else goto “debug”

## Application\_main

Use IVT

...

.....

End

ISR1 ; add. 0x004

Process data

## Debug\_main

Use **AIVT**

...

.....

End

ISR1 ; add. 0x**1**04

Glow LED; Visual Indication

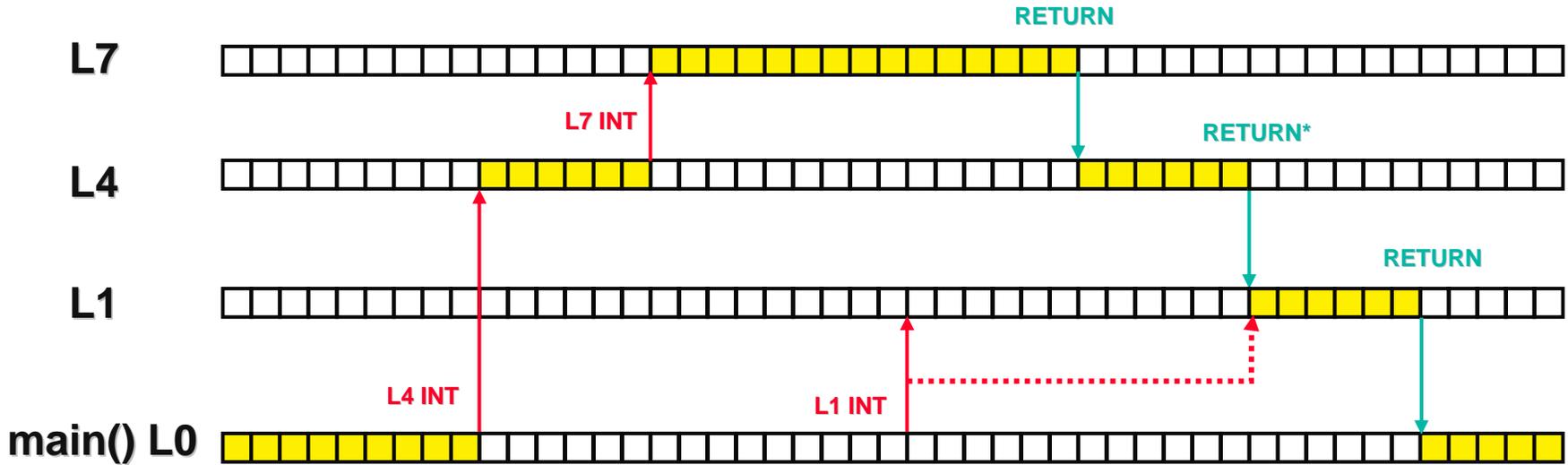


# Interrupt Priority

- CPU has 16 priority levels
  - **Level 0 is the default CPU level (main)**
  - **Level 1 - 7 for user interrupts**
  - **Level 8 -15 reserved for traps (NMI)**
- Interrupt with priority level greater than current CPU level ( $IPL<3:0>$ ) can interrupt the CPU
- Interrupts are nested by default, Nesting can be disabled by setting NSTDIS bit in INTCON1 register
- IVT has natural priority to resolve conflicts
- User assigned priority overrides natural priority

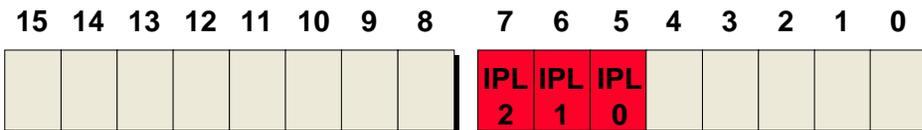


# Interrupt Nesting Example

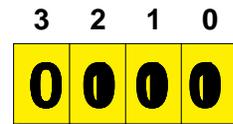


CPU EXECUTION TRACE

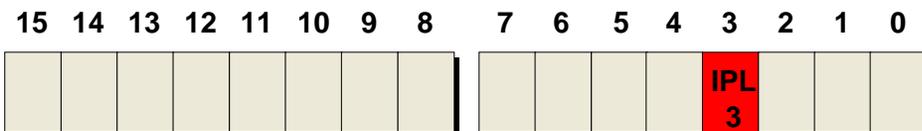
## Status Register



IPL<3:0>



## Core Control Register





# Traps

- **Soft Traps**

- Priority levels 8 through 12
- Treated as an NMI with fixed priority
- ‘Normal’ exception process is taken
- Example:
  - **Arithmetic error trap (priority level 11)**
  - **Stack error trap (priority level 12)**

- **Hard Traps**

- Priority levels 13 through 15
- CPU execution flow is immediately suspended
- Execution cannot resume until trap is acknowledged
- Example:
  - **Address error trap (level 13)**
  - **Oscillator error trap (level 14)**



# Error Traps

- What happens when a trap occurs?
- Branches to address 0 by default
- Better to branch to a `reset` instruction
  - Linker adds a `reset` instruction and vector
- Best to add your own error trap handler

Program Memory				
	Address	Opcode	Label	
	0051E	000000		<code>nop</code>
	00520	000000		<code>nop</code>
	00522	FE0000	<code>_DefaultInterrupt</code>	<code>reset</code>
	00524	FFFFFF		<code>nopr</code>
	00526	FFFFFF		<code>nopr</code>

Opcode Hex   Machine   Symbolic   PSV Mixed   PSV Data

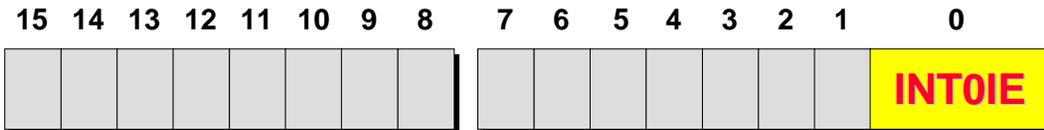
Program Memory		
	Address	Disassembly
	00000	<code>goto _reset</code>
	00002	<code>nop</code>
	00004	<code>_DefaultInterrupt</code>
	00006	<code>_OscillatorFail</code>
	00008	<code>_AddressError</code>
	0000A	<code>_StackError</code>
	0000C	<code>_MathError</code>
	0000E	<code>_DefaultInterrupt</code>
	00010	<code>_DefaultInterrupt</code>
	00012	<code>_DefaultInterrupt</code>
	00014	<code>_DefaultInterrupt</code>

Opcode Hex   Machine   Symbolic   PSV Mixed

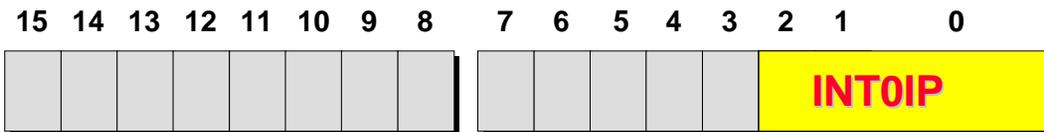


# Configuring Interrupts

**IEC0: Interrupt Enable Control Register 0 : Enable Interrupt**

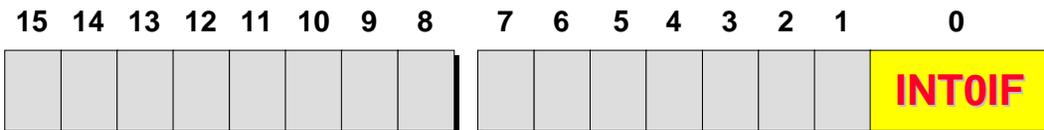


**IPC0: Interrupt Priority Control Register 0 : Assign Priority**



- 000: Disabled
- 001: Priority 1 Interrupt
- 010: Priority 2 Interrupt
- 011: Priority 3 Interrupt
- 100: Priority 4 Interrupt
- 101: Priority 5 Interrupt
- 110: Priority 6 Interrupt
- 111: Priority 7 Interrupt

**IFS0: Interrupt Flag Control Register 0 : Clear Interrupt in the ISR**





# Interrupt Service Routines in C

- **Compiler saves / restores the ISR context**
  - Hardware saves return address and SR
- **Fast ISR saves **W0-W3** in one cycle**
  - Uses `push.s` and `pop.s` (only 1 level deep)
- **Globals used by the ISR must be volatile**
  - `volatile unsigned global_flag;`
- **ISR must clear the Interrupt Flag**
  - `IFS0bits.T1IF = 0;`
- **Making function calls not recommended**
- **Linker will populate the Vector Table**
  - Must use designated interrupt names
  - The designated interrupt names can be found in the .gld file of that device, which can be found at  
\\....\microchip\mplab c30\support\gld



# Function Attributes

- **Declare an interrupt function**

```
void __attribute__((interrupt)) _ADCInterrupt(void)
void _ISR _IC1Interrupt(void)
```

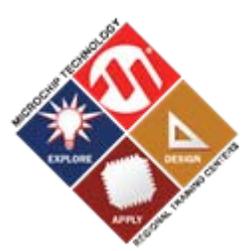
- **Make function use shadow registers**

```
void __attribute__((interrupt, shadow))
                               _ADCInterrupt(void)
```

```
void _ISRFAST _IC1Interrupt(void)
```

```
int __attribute__((shadow)) FastFunc(int Abc)
```

- Saves W0-W3 and SR bits with `push.s` and `pop.s`



# Lab 3: Working with Interrupts

- **Goal:**

- Understand Interrupt configuration
- Understand Interrupt priority
- Writing Interrupt handler for a given Interrupt vector

- **To Do:**

- Look into the Hand out provided

- **Expected Result:**

- Look into the Hand out provided

# HANDS-ON

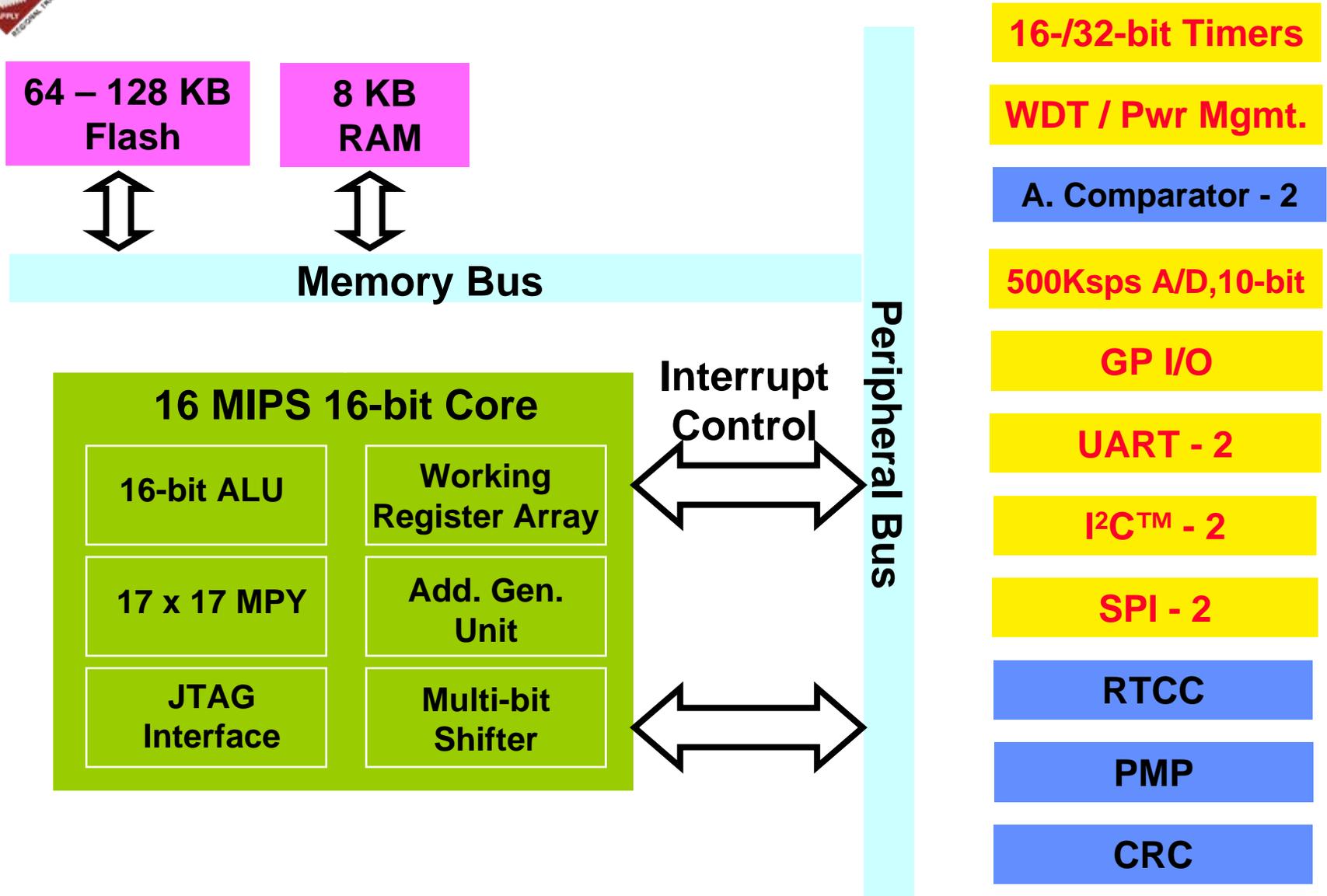
# Training

# Peripherals



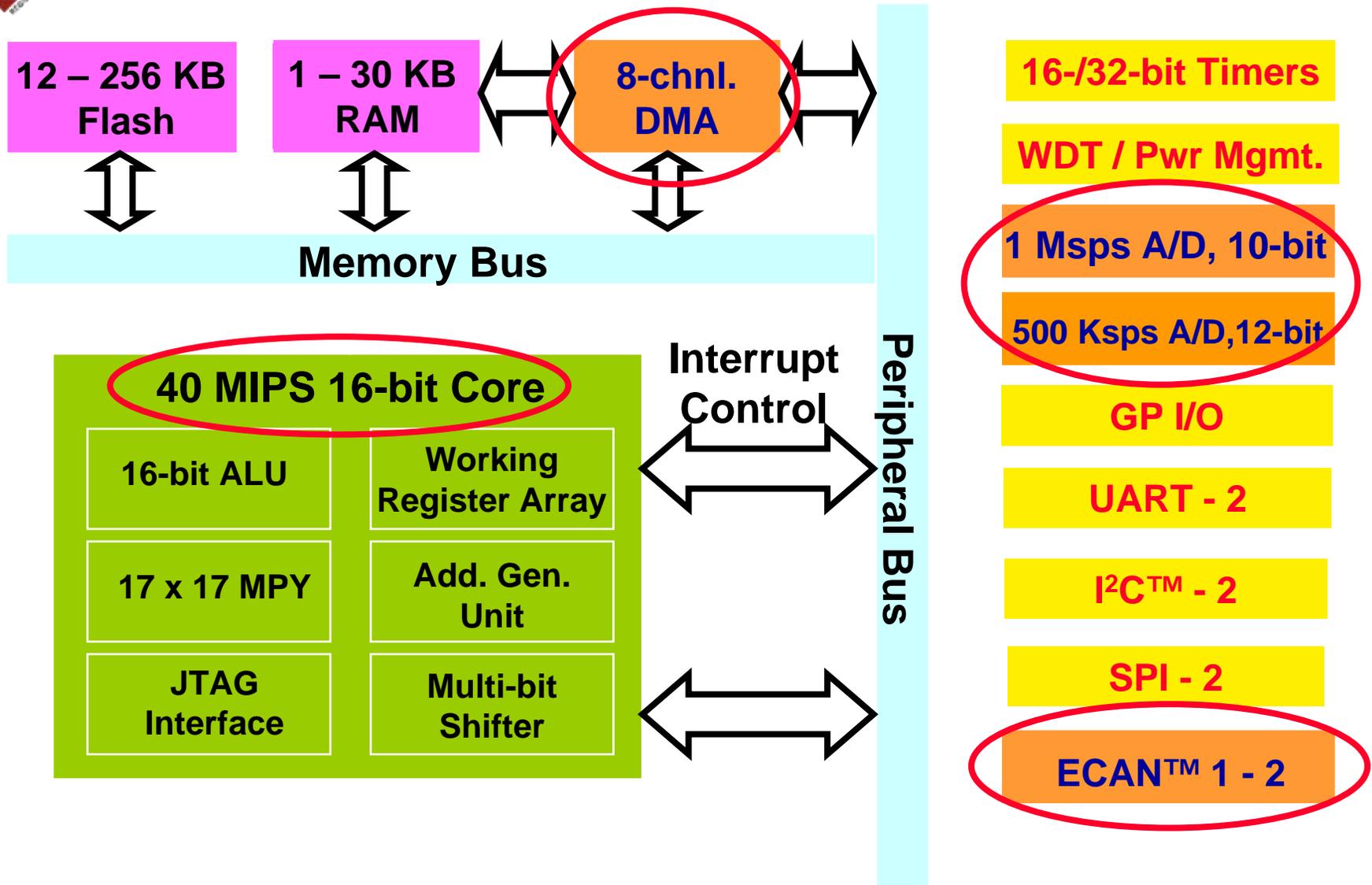


# PIC24F Family Block Diagram



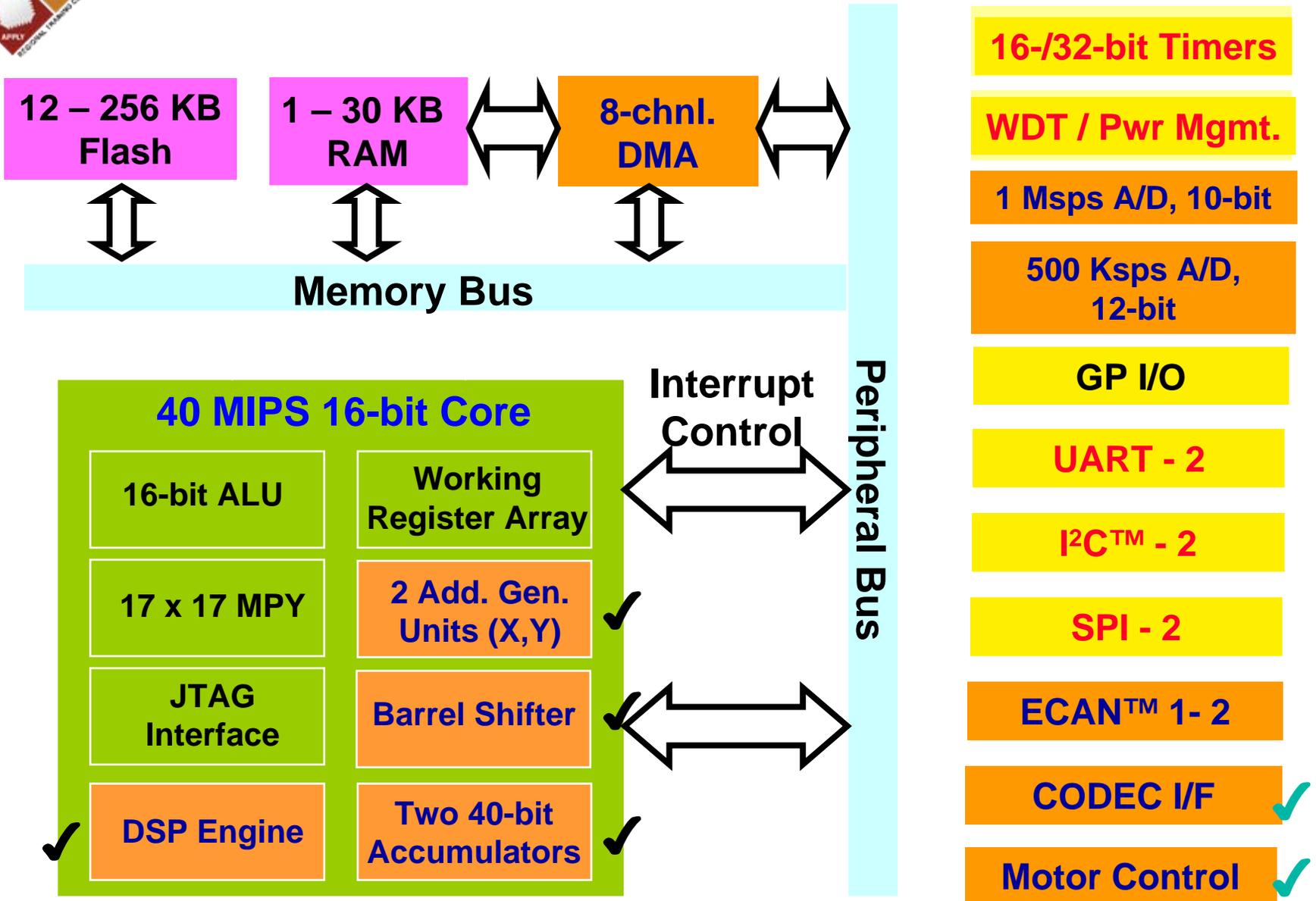


# PIC24H Family Block Diagram





# dsPIC30/33 Family Block Diagram

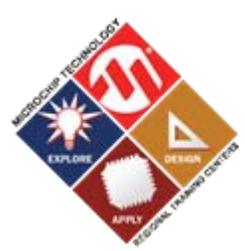


# HANDS-ON

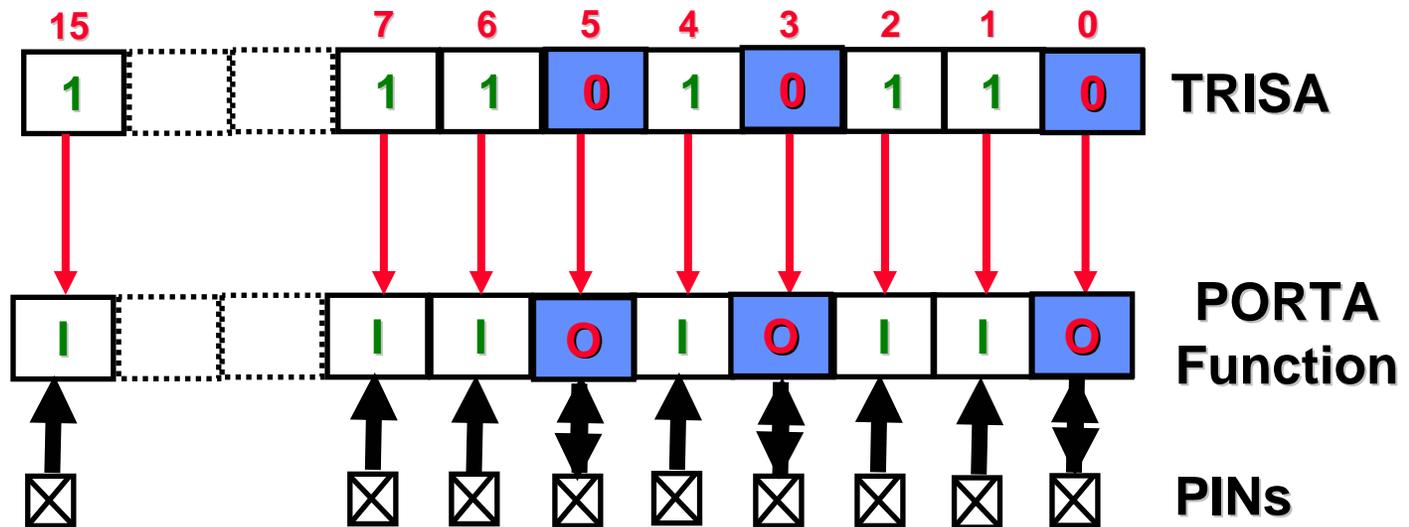
# Training

## I/O Ports



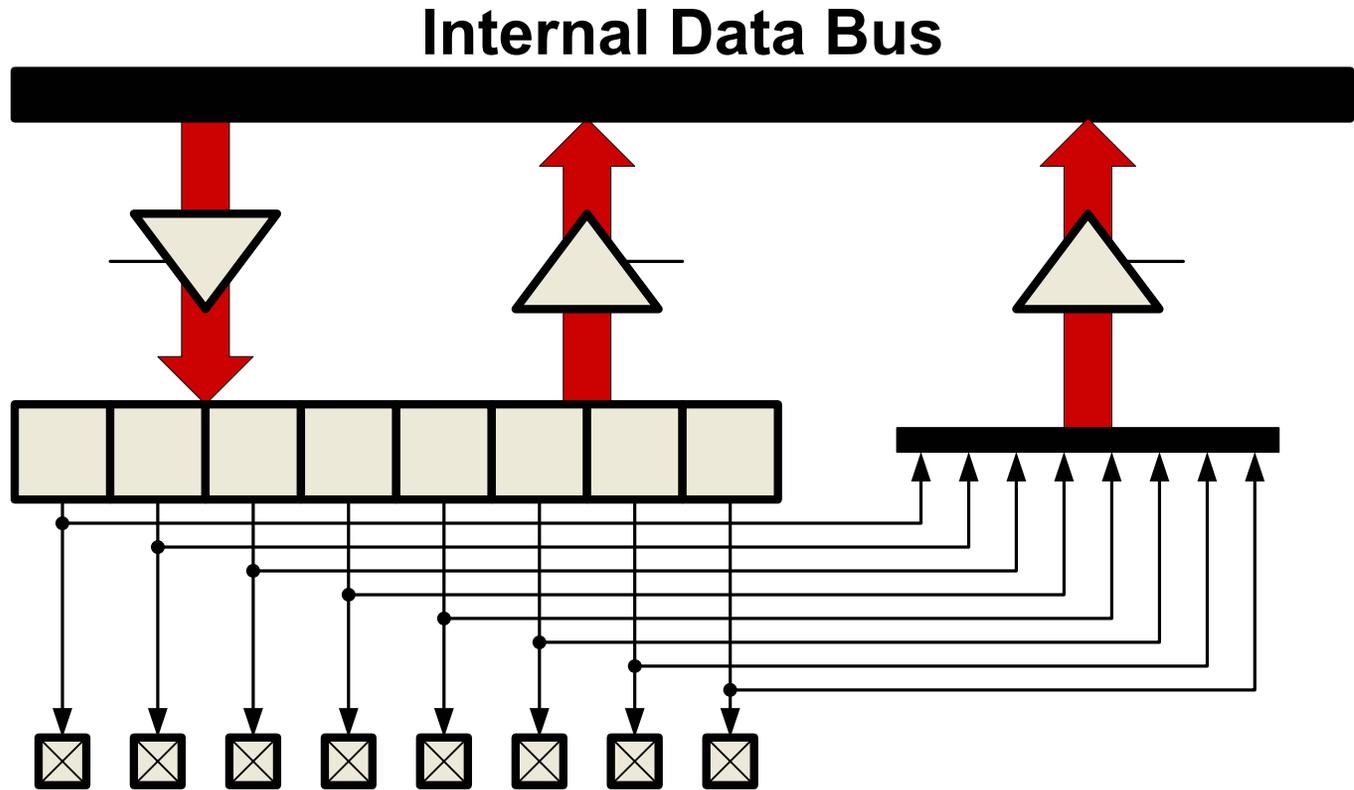


# I/O PORT





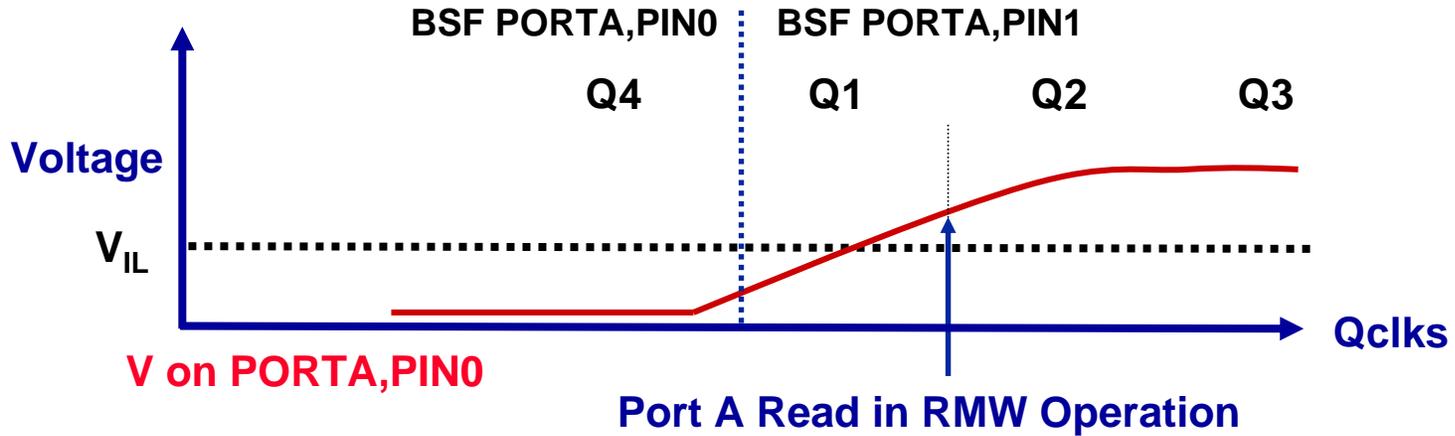
# Digital I/O Ports



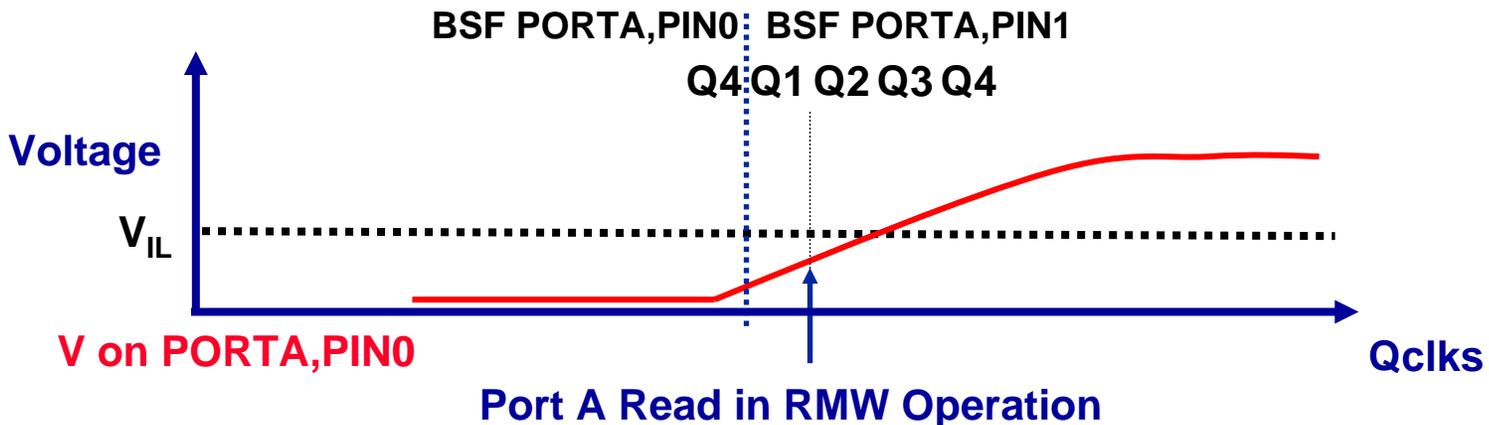


# Why do we have LAT Reg?

## At Low Frequency or Low Capacitive Loading



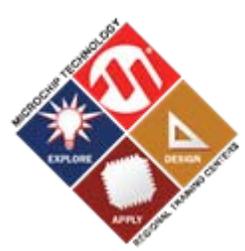
## At High Frequency or High Capacitive Loading





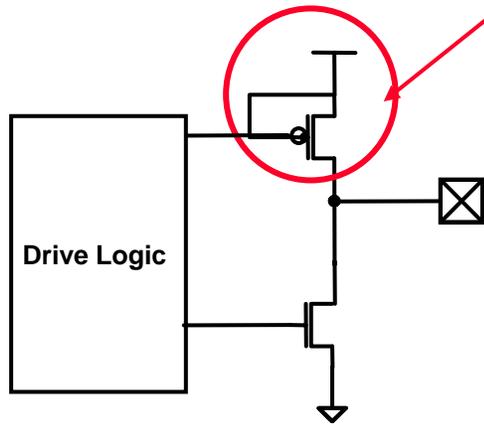
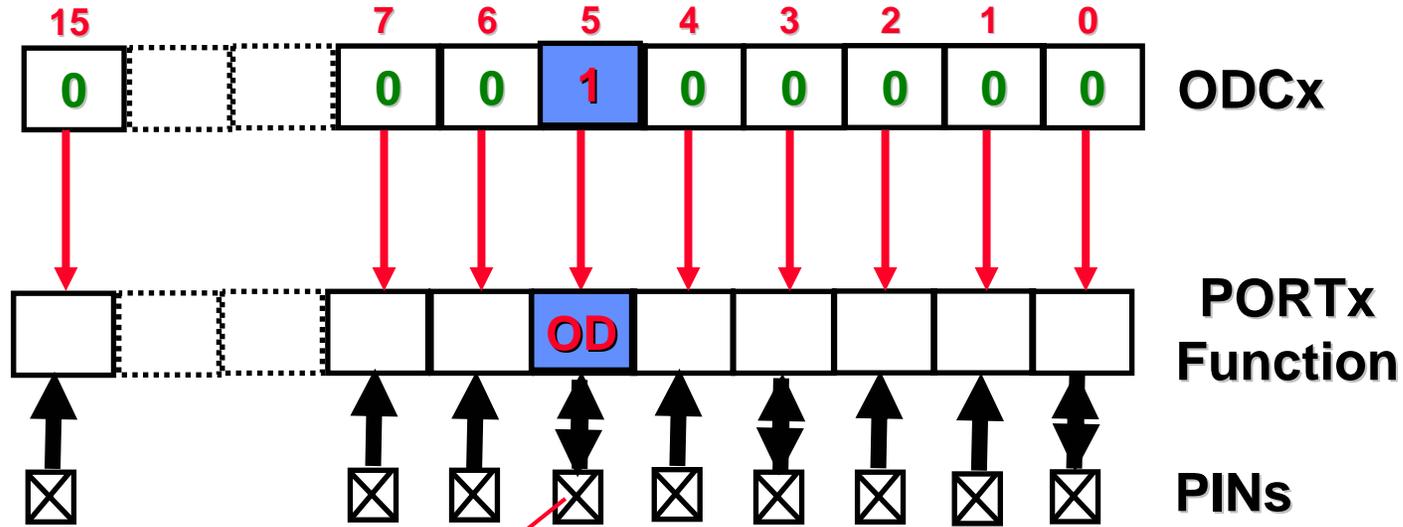
# Why do we have LAT Reg?

- **Use RMW instruction on the LATch rather than PORTx:**
  - BSF LATA,#0
  - BSF       LATA,#1



# I/O PORT

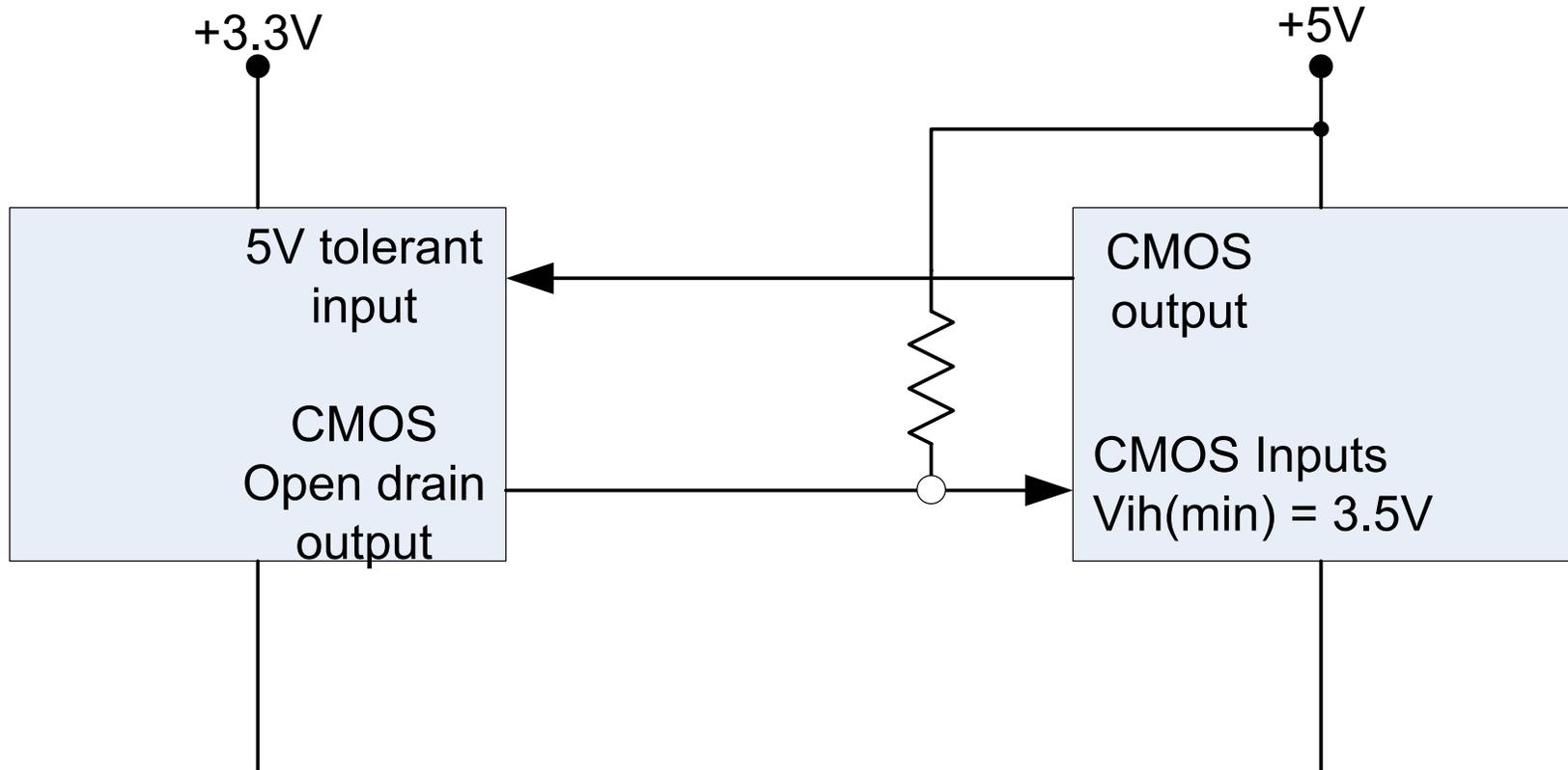
## Open Drain Control

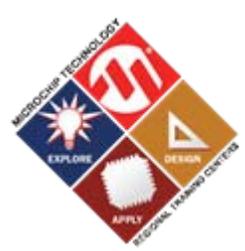




# Interfacing to 5V devices

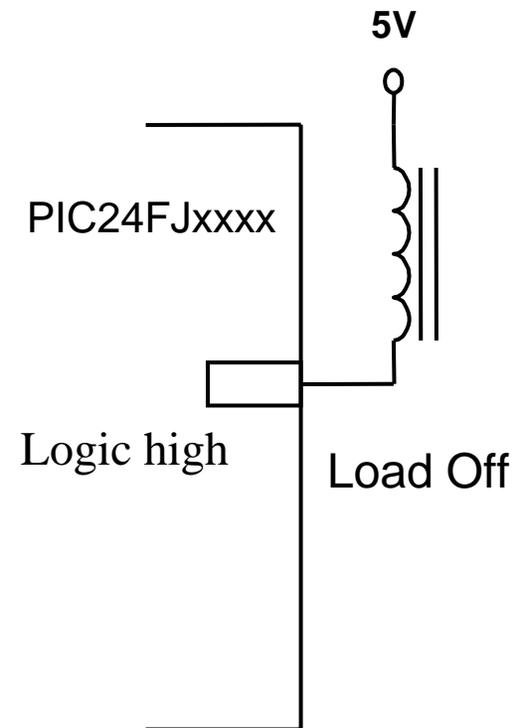
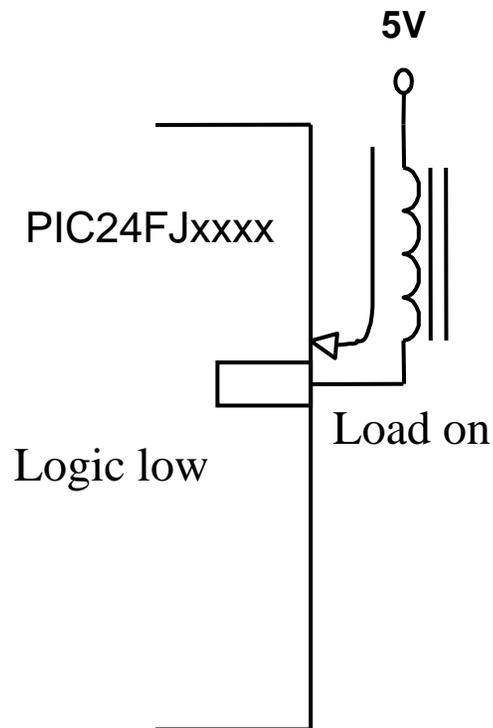
- **5V tolerant input and Open drain configuration simplifies 5V interface**





# Interfacing to 5V devices

- Interfacing to low impedance 5V load with open drain feature



# HANDS-ON

# Training

# ADC

## Analog to Digital Converter



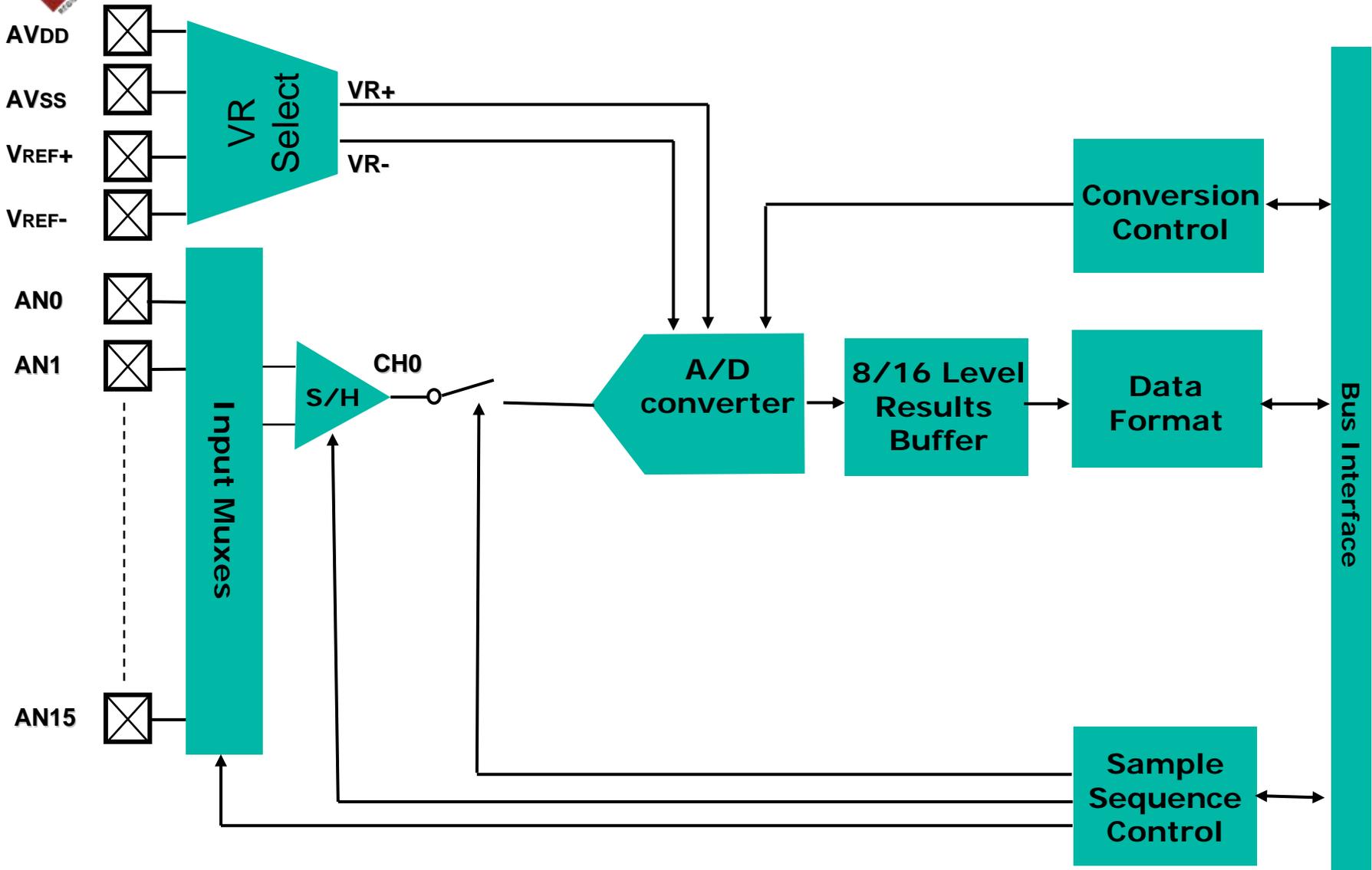


# 10 Bit ADC Features

- **Successive Approximation (SAR) conversion**
- **Conversion speeds of up to 500ksp/s**
- **Up to 16 analog input pins**
- **External Voltage reference pins**
- **Automatic Channel Scan mode**
- **Selectable conversion trigger source**
- **16-word conversion result buffer**
- **Selectable Buffer Fill modes**
- **Four result alignment options**
- **Operation during CPU Sleep and Idle modes**



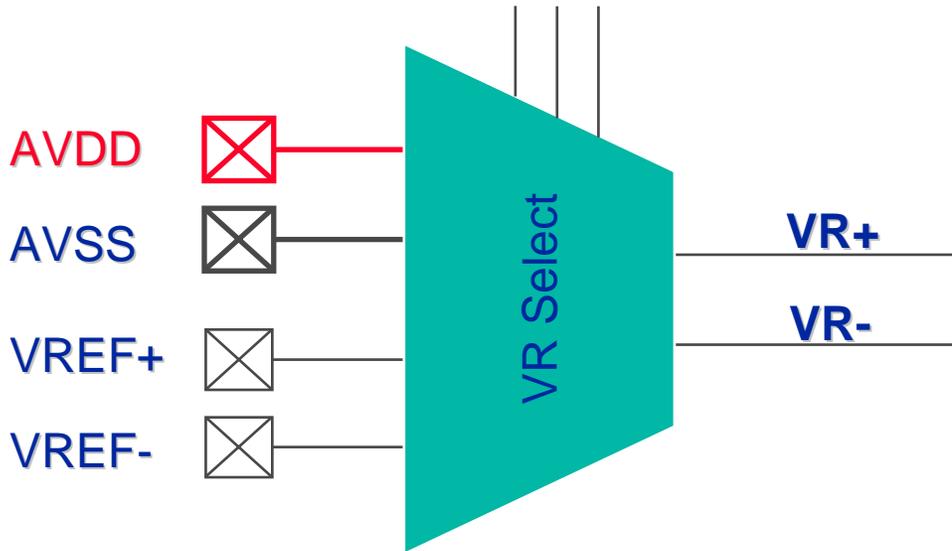
# PIC24F A/D





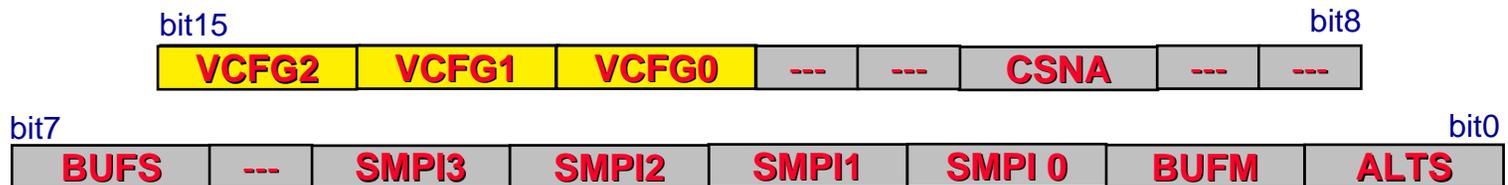
# 10-bit ADC - Block Diagram

AD1CON2<VCFG2:VCFG0>



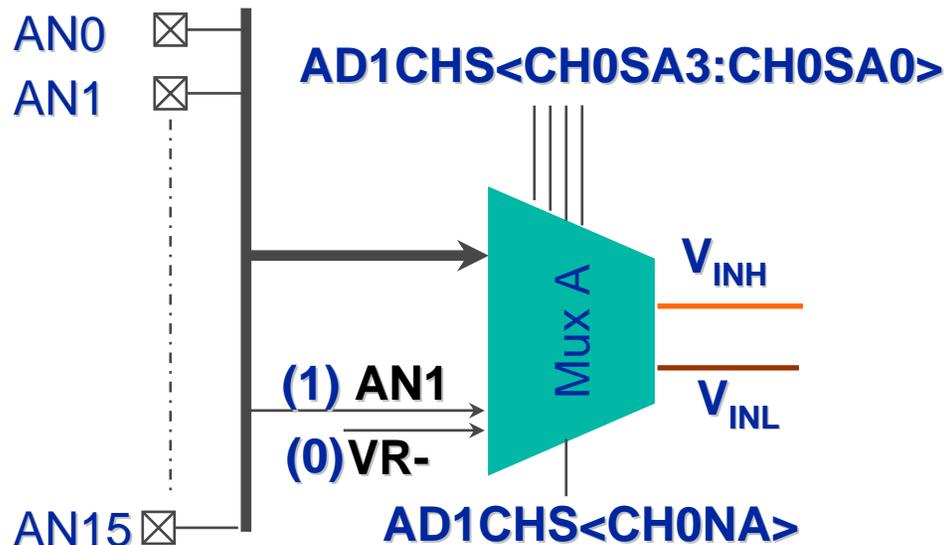
VCFG2:VCFG0	VR+	VR-
000	AVDD	AVSS
001	VREF+	AVSS
010	AVDD	VREF-
011	VREF+	VREF-
1xx	AVDD	AVSS

## AD1CON2 Register



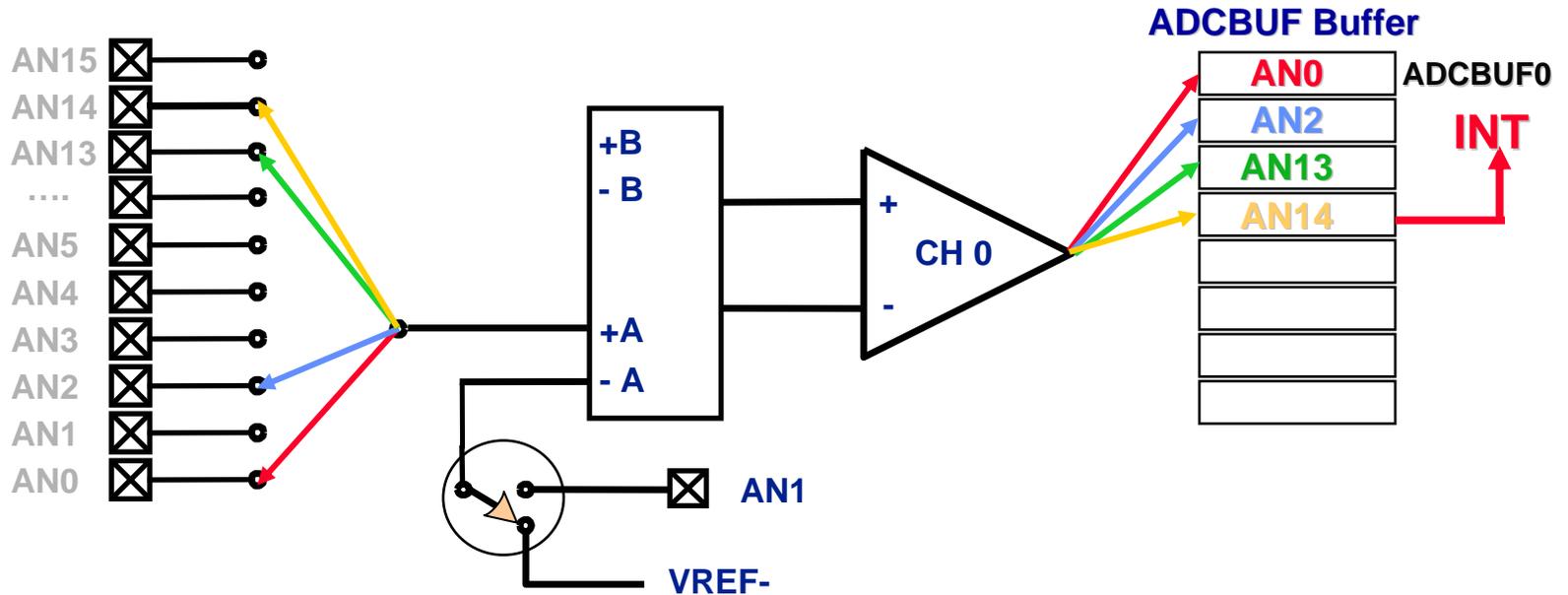


# 10-bit ADC - Block Diagram



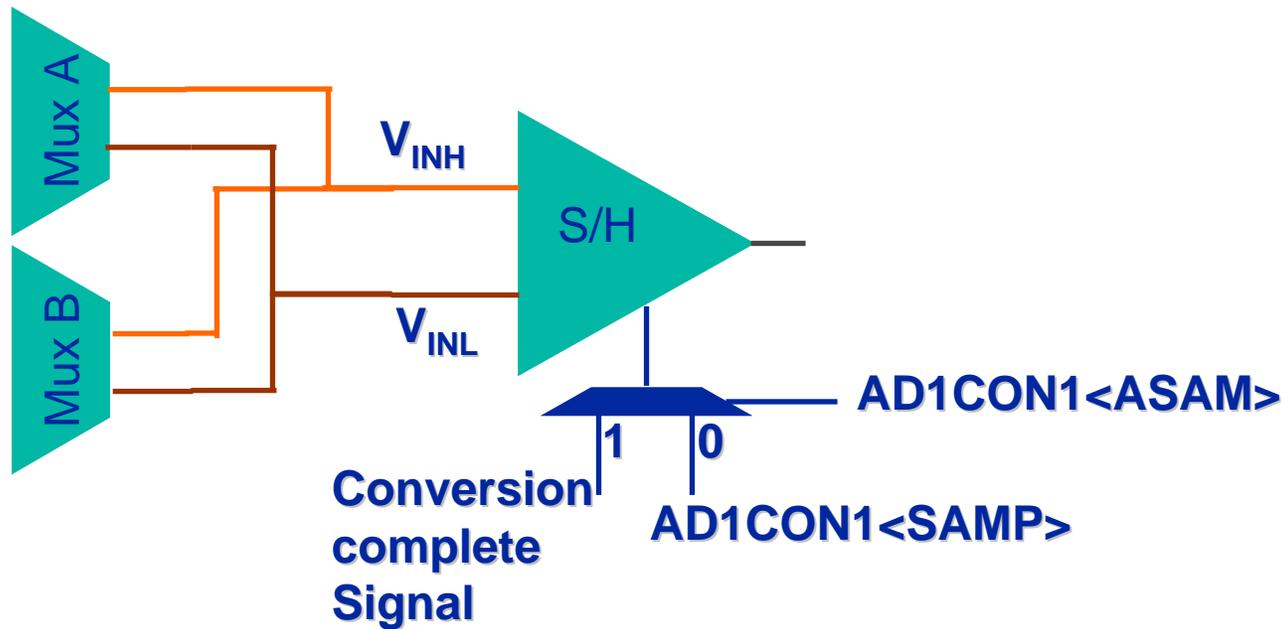


# Channel Scanning





# 10-bit ADC - Block Diagram

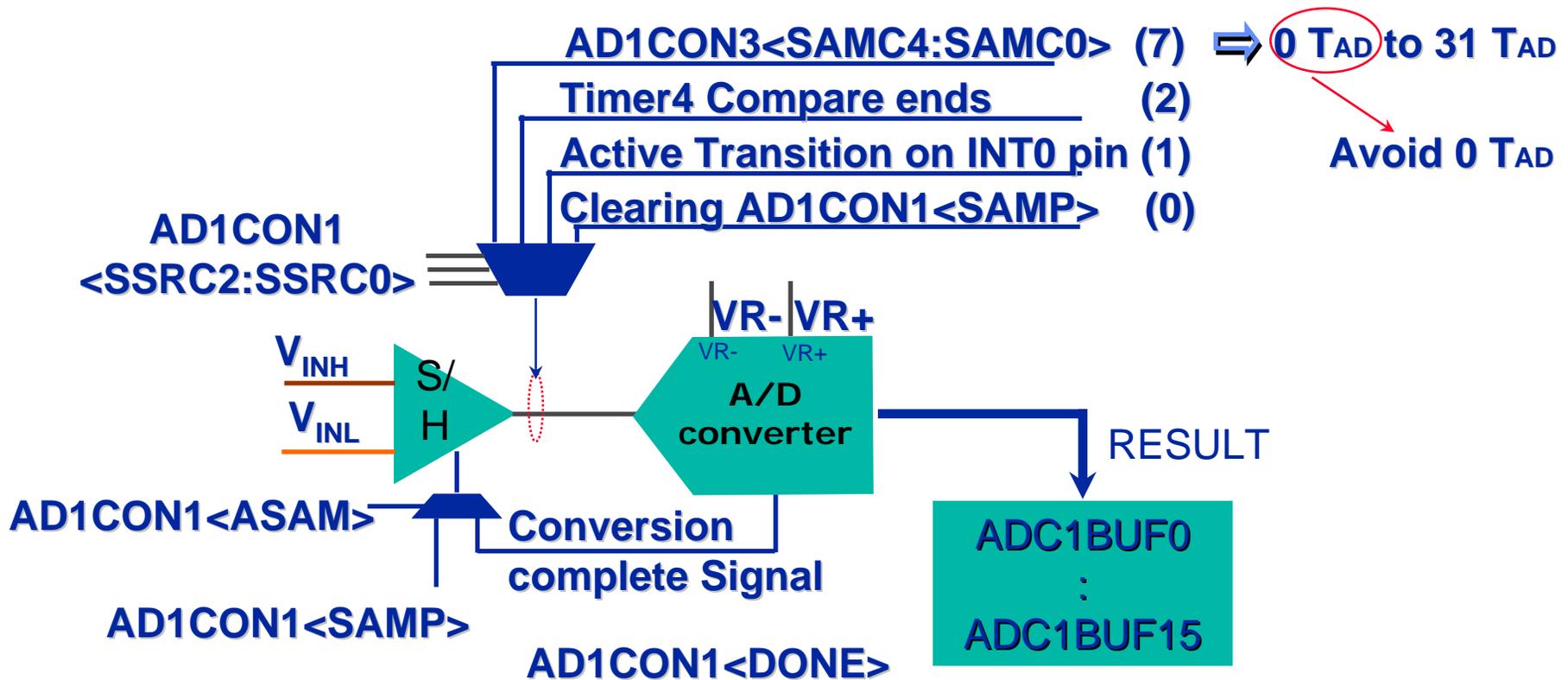


**AD1CON2 Register**



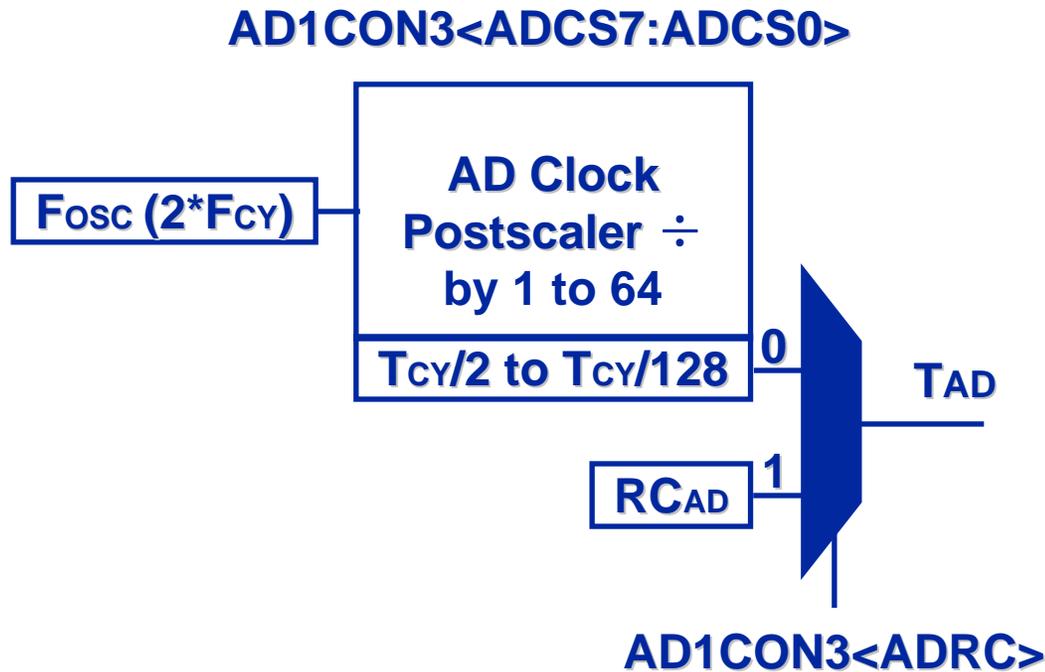


# 10-bit ADC - Block Diagram





# 10-bit ADC - Block Diagram





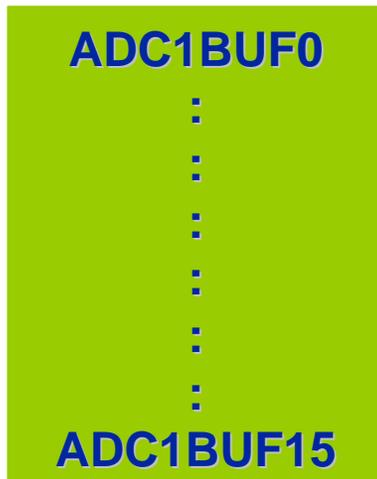
# 10-bit ADC - Block Diagram

**RESULT  
FORMAT**

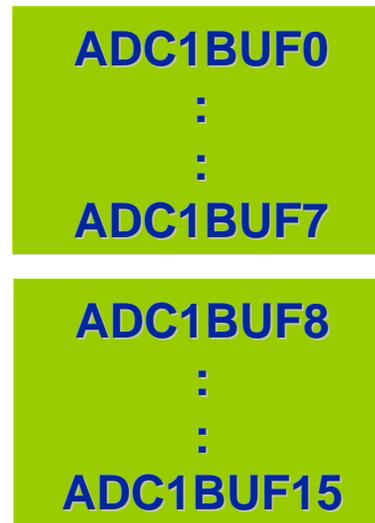
```
0000 00dd dddd dddd
      ssss sssd dddd dddd
      dddd dddd dd00 0000
      sddd dddd dd00 0000
```

**AD1CON1<FORM1:FORM0>**

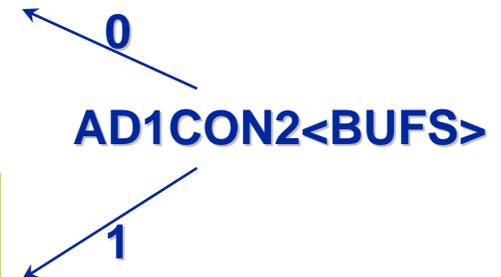
**AD1CON2<SMPI3:SMPI0>**



**AD1CON2<BUFM> = '0'**

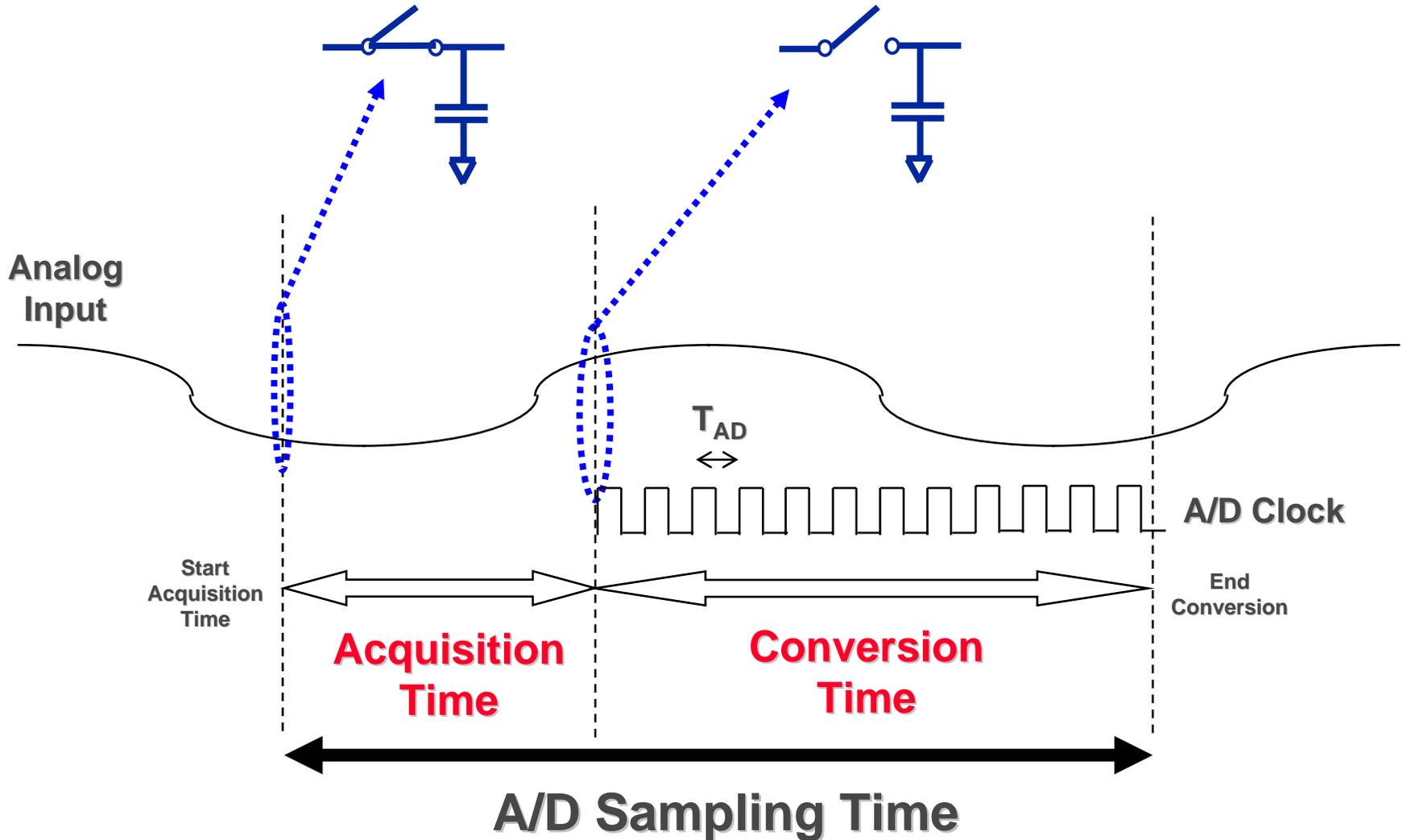


**AD1CON2<BUFM> = '1'**





# Acquisition/Conversion Timing



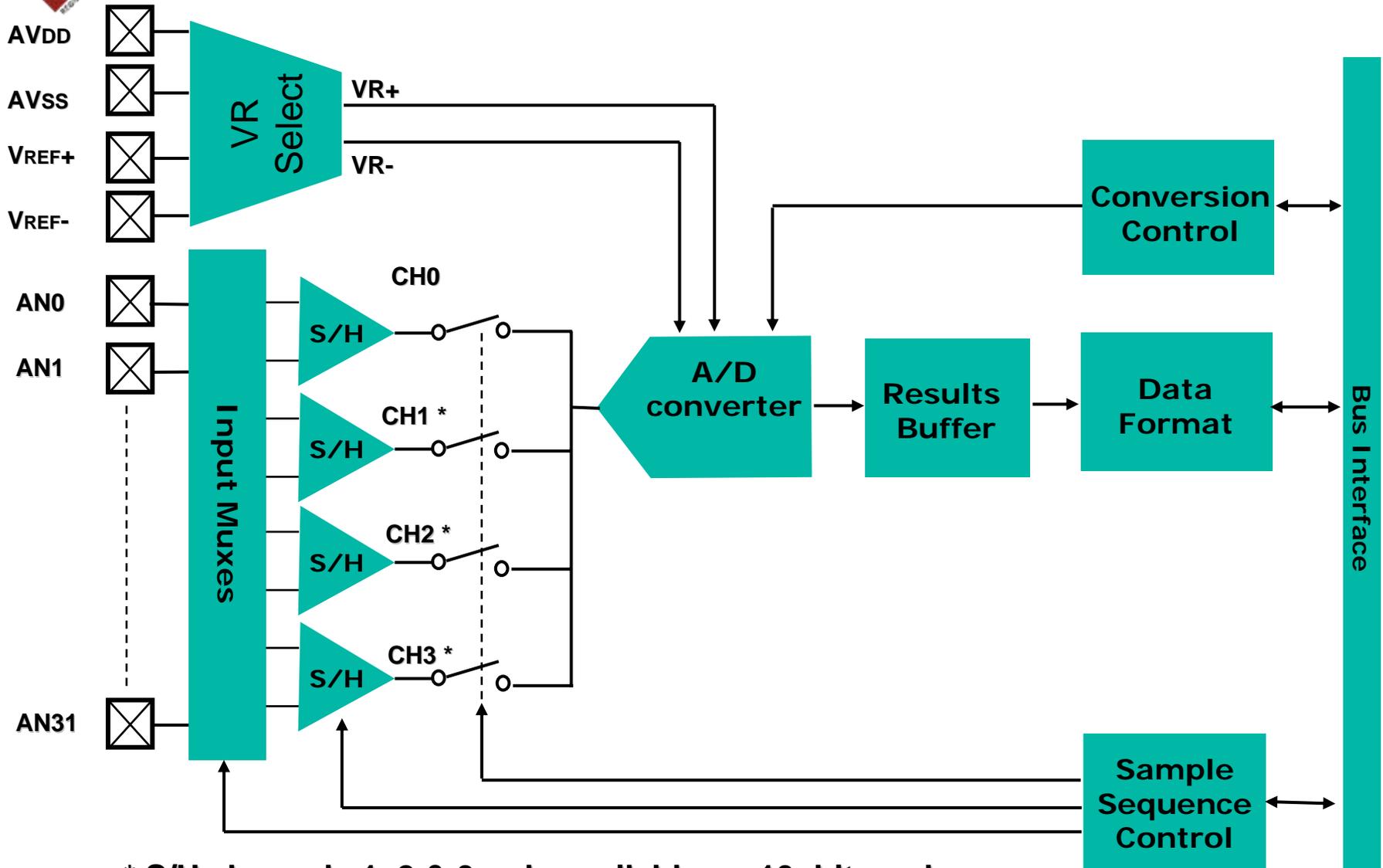


# 12-bit A/D Converter

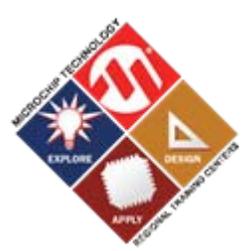
- **200 K Samples / Sec Sampling Rate**
- **Up to 16 input channels**
- **16 ADC Result buffer to store converted data**
- **External VREF+ and VREF-**
- **Analog Input Range: 0 to 5V (VREF- to VREF+)**
- **Multiple Trigger sources for Start of Conversion**
  - Software Trigger
  - Motor Control PWM
  - Timer 3
  - External Interrupt
  - Automatic conversion on internal time out



# PIC24H/30/33 A/D



\* S/H channels 1, 2 & 3 only available on 10-bit mode

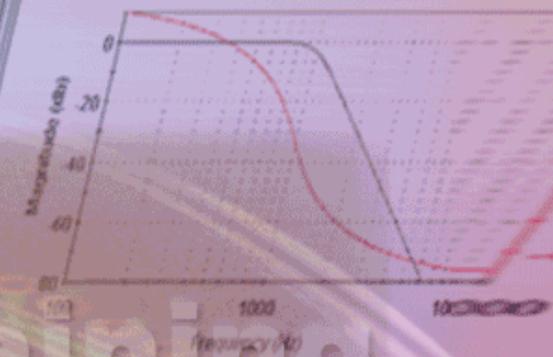


# Lab 4: Working with ADC

- **Goal**
  - To configure ADC
  - To configure I/O ports
  - To read ADC and o/p on to LEDs
- **To Do:**
  - Look into the Hand out provided
- **Expected Result:**
  - The average pot value is displayed on LEDs

# HANDS-ON

# Training

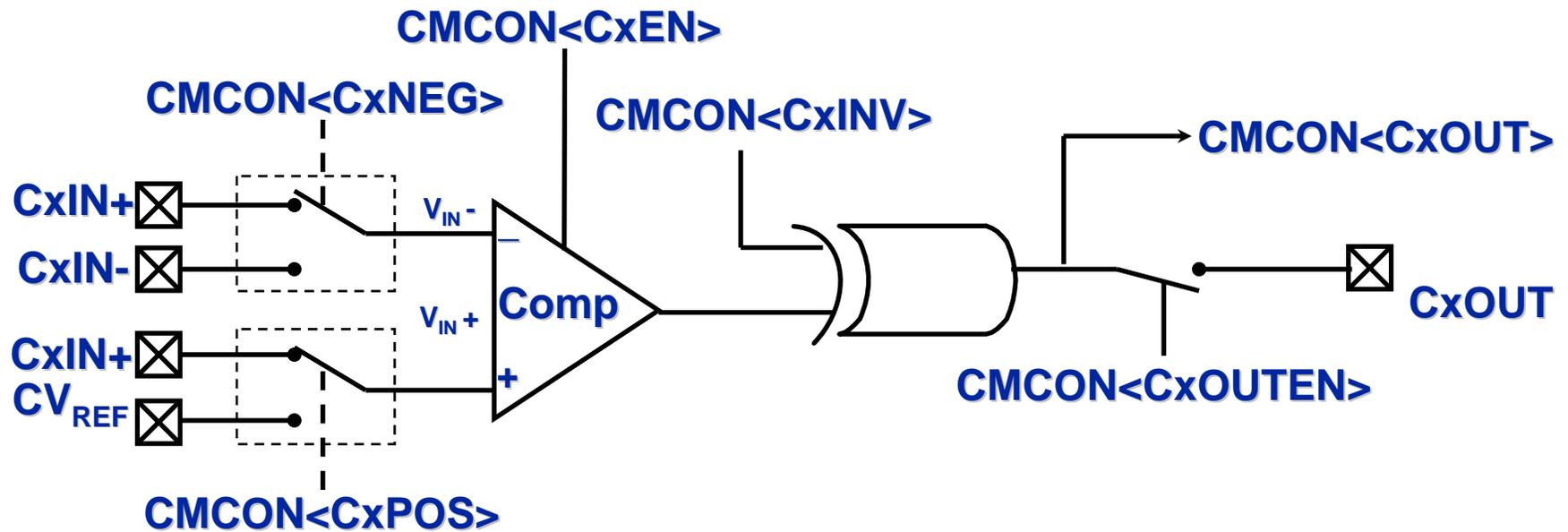


# Analog Comparator Module



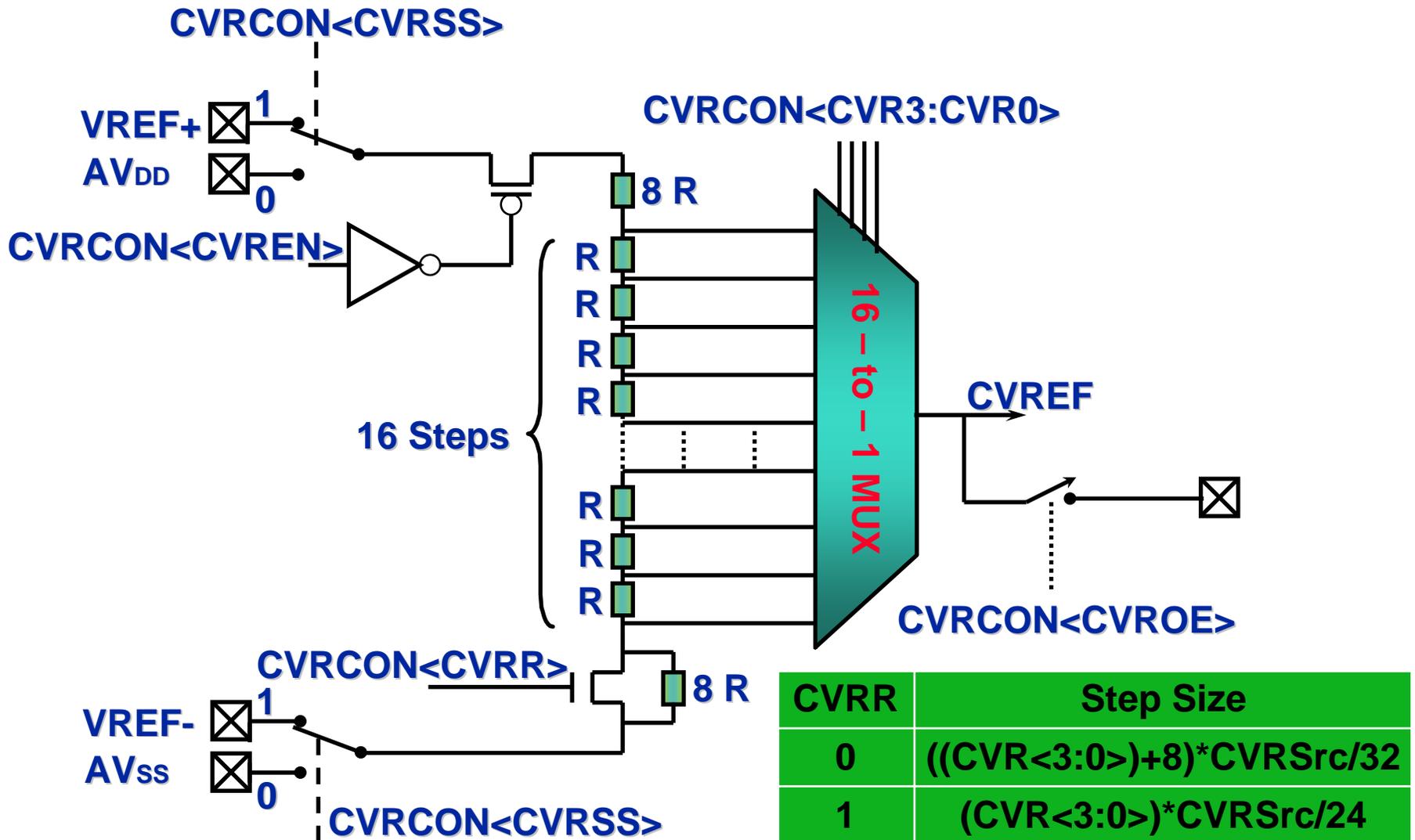


# Analog Comparator Block Diagram





# Analog Comparator Voltage Reference Generator



# HANDS-ON

# Training



# Timers



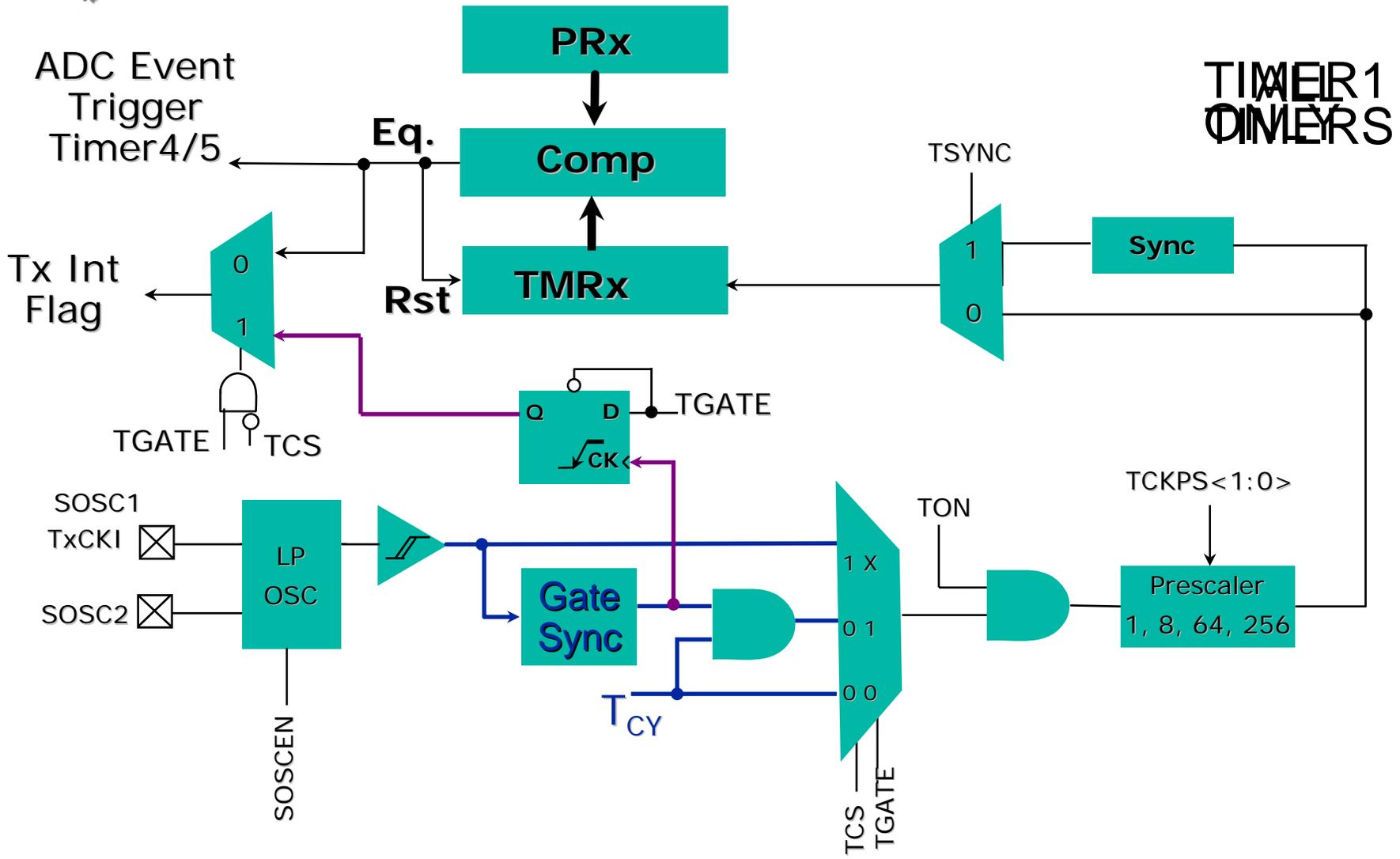


# Timer Features

- **Five 16-bit General Purpose Timers / Counters**
  - Similar functionality between all 5 timers
    - Asynchronous counter feature only in Timer1
- **Period Registers for each**
  - Interrupt generation on match
  - Reset on match
- **Gated Timer operation on each**
  - Interrupt on falling edge of gate
- **Four of these timers (Timer 2+3 & 4+5) can make two 32-bit timers/counters**



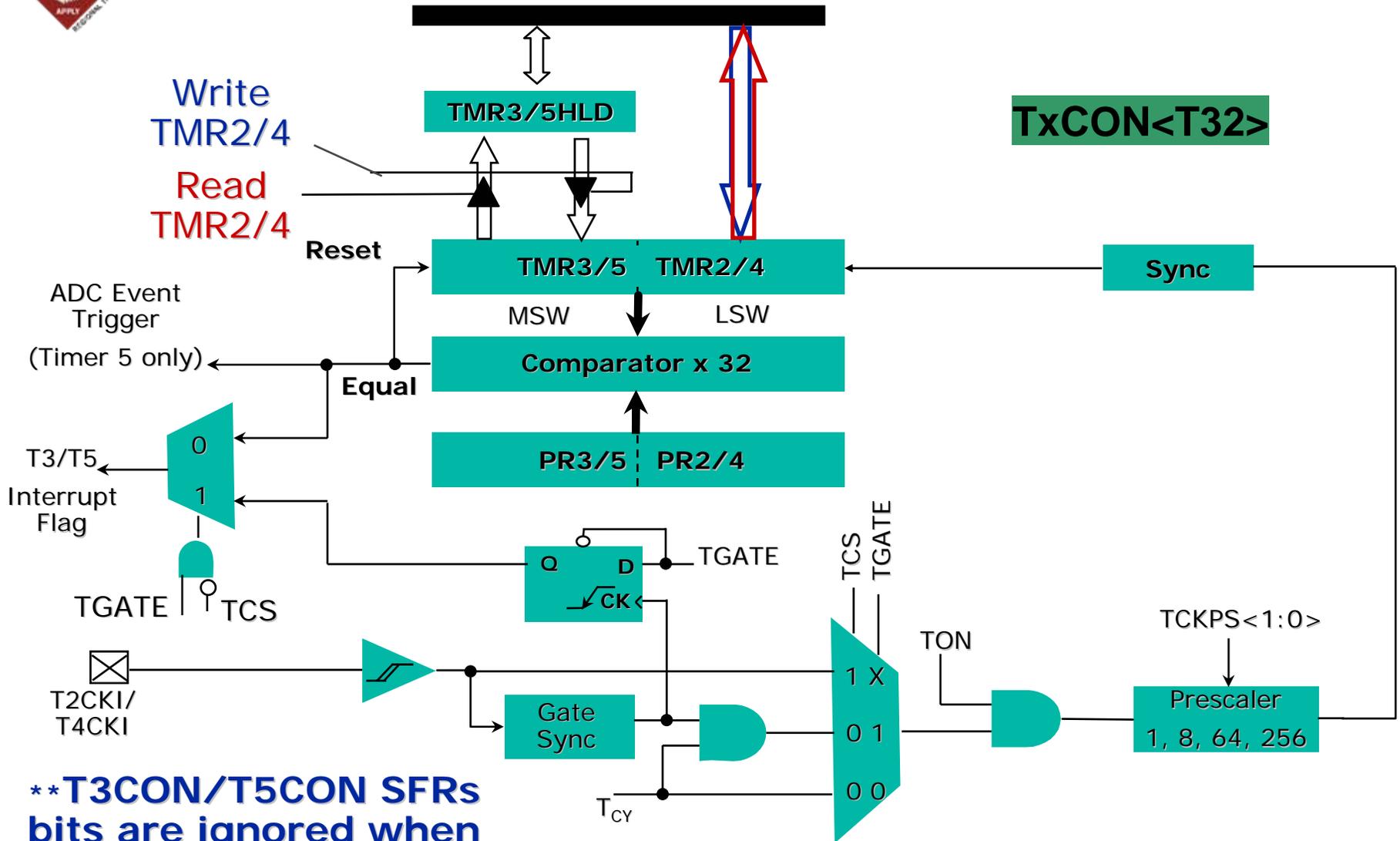
# Timer x



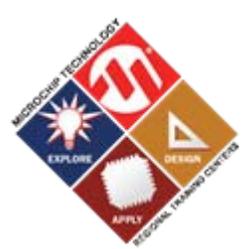
TIMER 1  
DIVERS



# 32-bit Timer (T2+T3/T4+T5)



**\*\*T3CON/T5CON SFRs bits are ignored when in 32-bit timer mode**



# Lab 5: Working with a 32 bit Timer

- **Goal**

- Understand working of Timers in 32 bit mode
- Configure the Timer 2/3 pair for 32 bit mode
- Implement a stop watch

- **To Do:**

- Look into the Hand out provided

- **Expected Result:**

- Press the Switch S3 to start timer
- Again press the Switch S3 to stop timer and LCD displays the Time elapsed between the start and stop

# HANDS-ON

# Training



# Input Capture





# Why Do We Need IC Module

- Consider an Example to measure the Pulse width of the signal
- The IC Module should be configured to
  - Capture on Every Edge
  - Interrupt on every Second Event
- Then the Pulse width of the signal can be found by using the formula:

$$\text{Pulse Width} = \frac{\text{Buffer 1} - \text{Buffer 0}}{\text{Timer running Frequency}}$$



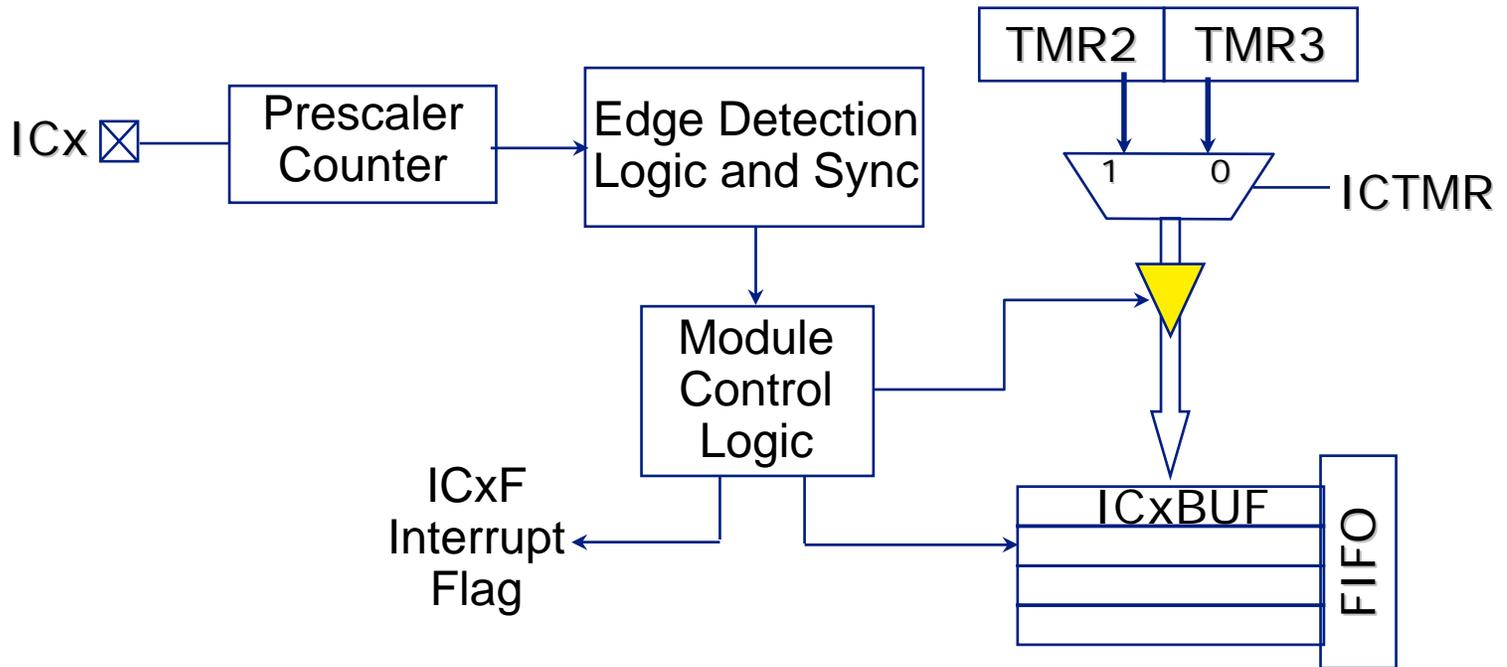
# Input Capture

## ● Input Capture

- Up to five Input Capture Channels
- Captures 16-bit timer value
- Tcy Resolution
- Timer 2 or Timer 3 as time base



# Input Capture



## ICxCON<ICI1:ICI0>

- 11: Interrupt on every fourth capture event
- 10: Interrupt on every third capture event
- 01: Interrupt on every second capture event
- 00: Interrupt on every capture event

## ICxCON<ICM2:ICM0>

- 111: Input Capture functions just as an interrupt pin while in SLEEP or IDLE mode
- 101: Capture on every sixteenth rising edge
- 100: Capture on every fourth rising edge
- 011: Capture on every rising edge
- 010: Capture on every falling edge
- 001: Capture on every edge (both rising and falling)
- 000: Capture module is turned off

# HANDS-ON

# Training

# Output Compare & PWM



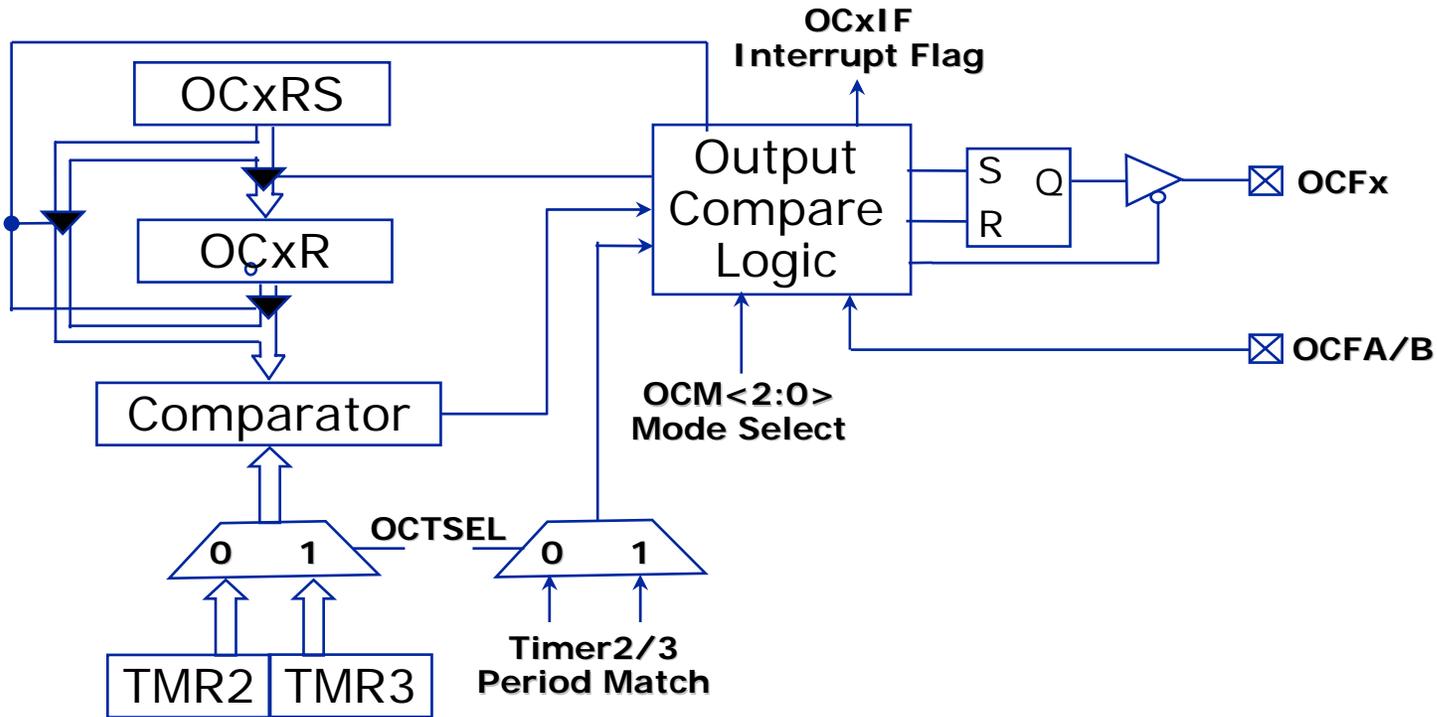


# Output Compare & PWM

- **Up to 8 Output Compare / PWM Channels**
- **Timer 2 or Timer 3 as time base**
- **16-bit Compare**
- **Minimum  $1T_{cy}$  pulse width allowed**
- **Several Compare Modes:**
  - Set, Reset or Toggle Pin
  - Single Pulse, Continuous pulse
  - PWM



# Output Compare



## OCxCON<OCM2:OCM0>

- 111: PWM mode with Fault pin option
- 110: PWM mode without Fault pin option
- 101: Initialize OCx-low, generate continuous signal
- 100: Initialize OCx-low, generate single pulse
- 011: Toggle OCx on every compare match
- 010: Initialize OCx-High, pull OCx low on compare match
- 001: Initialize OCx-low, pull OCx high on compare match
- 000: Compare module is turned off



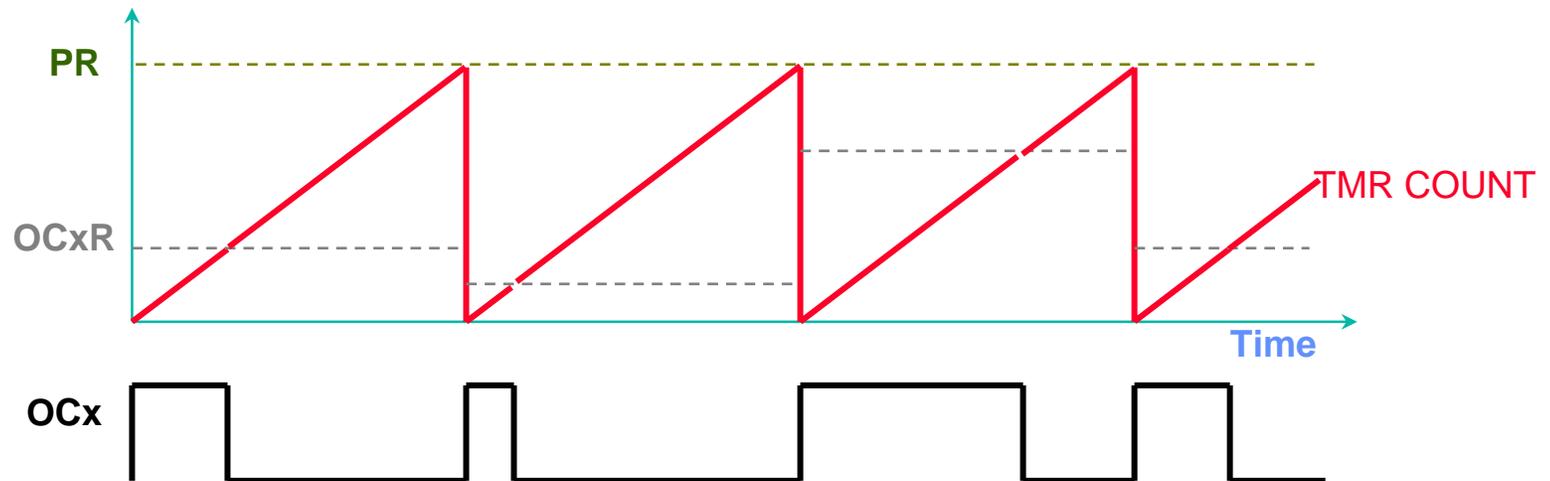
# Output Compare Module

- **PWM mode**

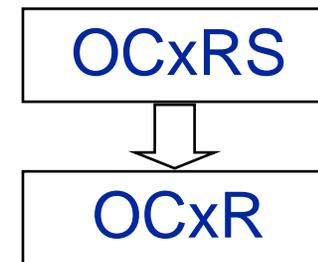
- 16-bit glitch-less (double buffered) PWM output
- Full range of 0 to 100% duty cycle
- Wide frequency range
- Selectable PWM shutdown on fault detection
  - This is an **Asynchronous** shutdown



# OC PWM Mode

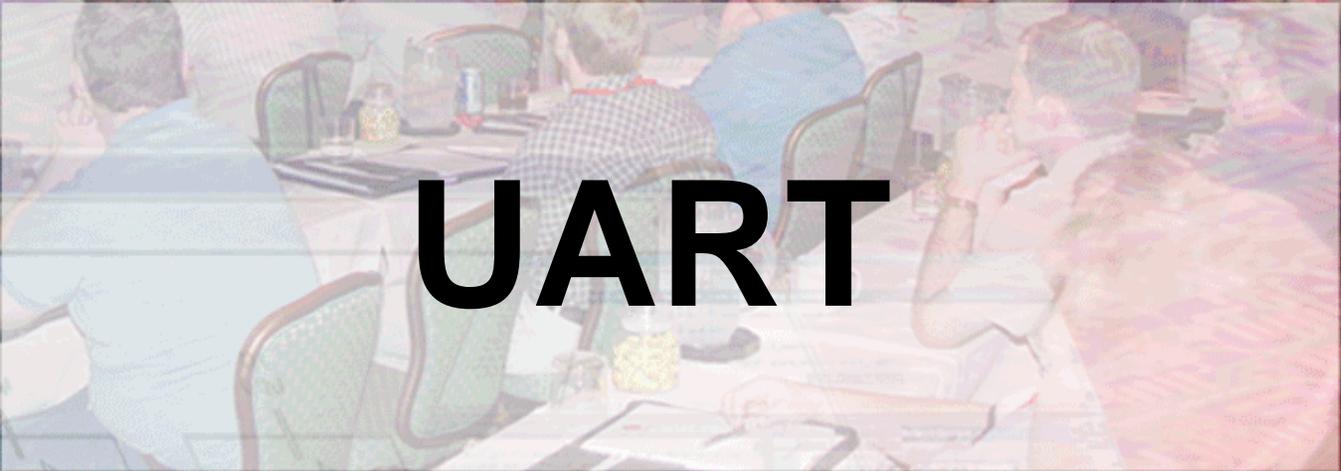


- **On timer period match**
  - OCx set
  - OCxRS copied to OCxR
- **On OCxR match**
  - OCx clear



# HANDS-ON

# Training



# UART

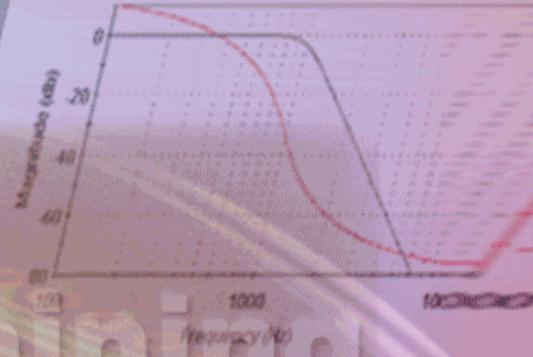
MICROCHIP TECHNOLOGY

EXPLORE

DESIGN

APPLY

REGIONAL TRAINING CENTERS



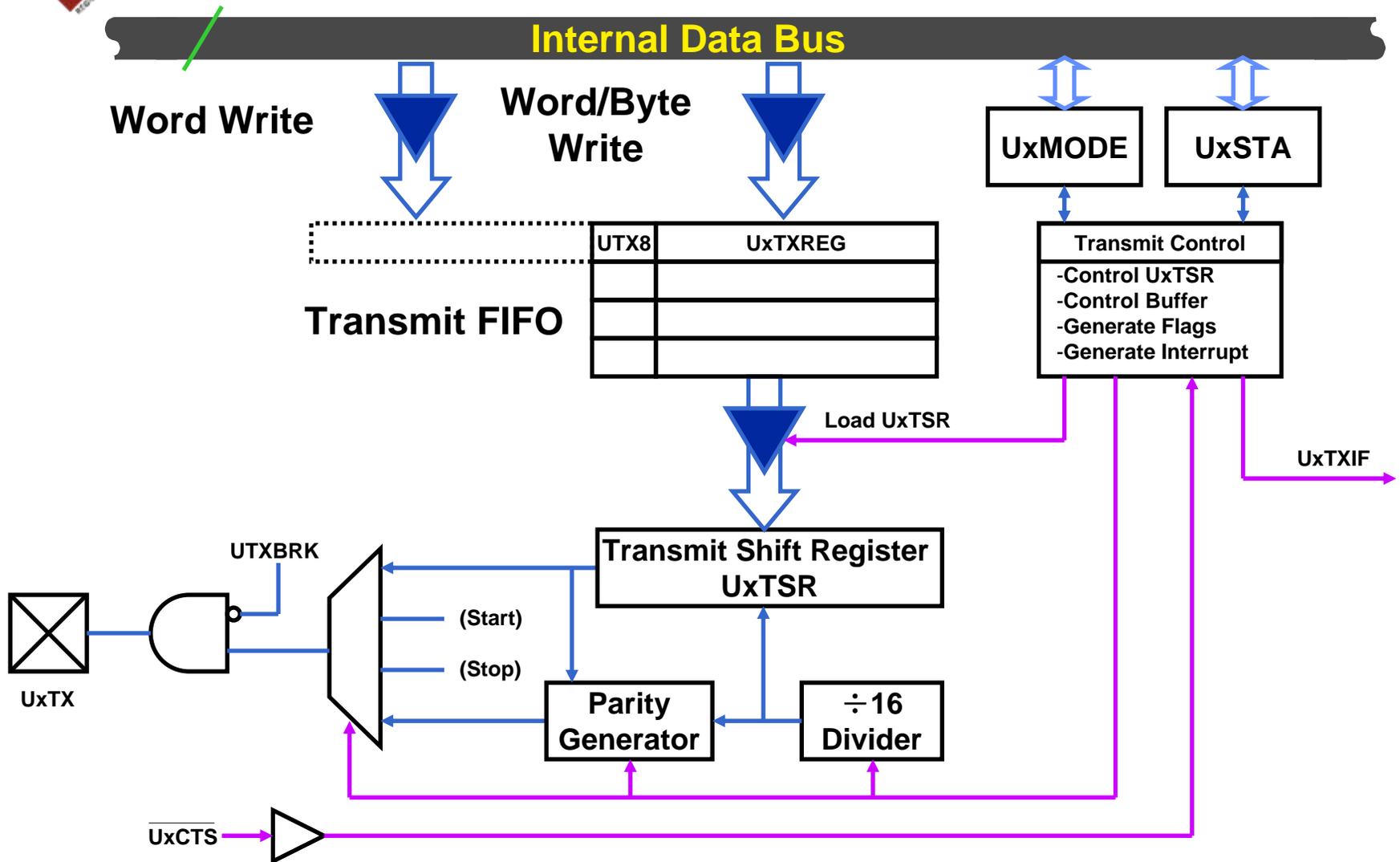


# UART Features

- **4-deep FIFO Transmit Data Buffer**
- **4-deep FIFO Receive Data Buffer**
- **Parity, Framing and Buffer Overrun Error Detection**
- **Support for 9-bit mode with Address Detect (9th bit = 1)**
- **Transmit and Receive Interrupts**
- **Loopback mode for Diagnostic Support**
- **LIN 1.2 Protocol Support**
- **IrDA<sup>®</sup> Support**



# UART Module – Transmit

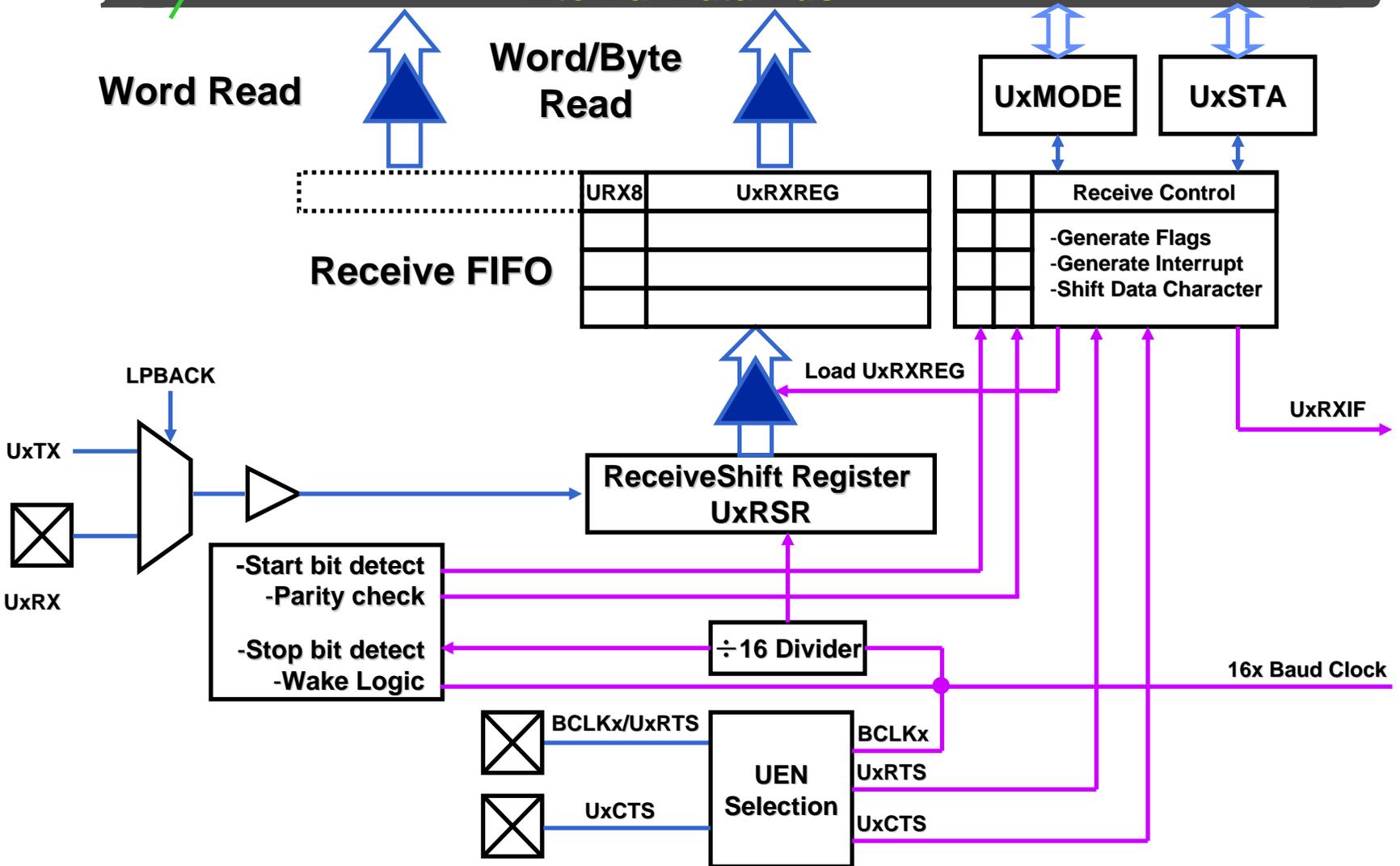


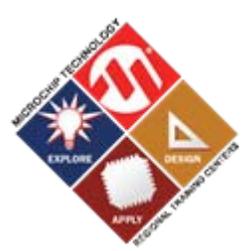


# UART Module – Receive

16

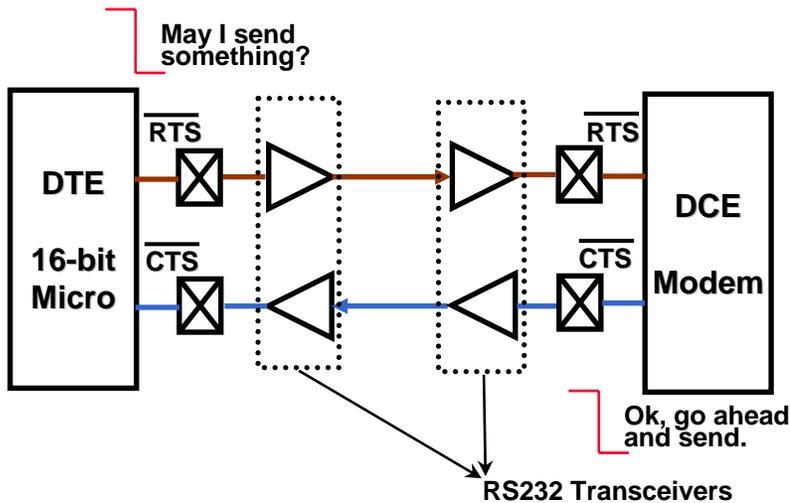
**Internal Data Bus**



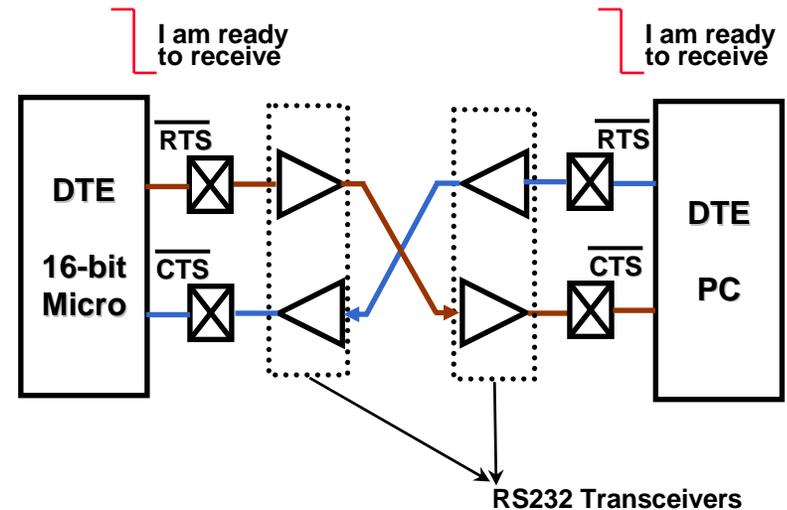


# UART with CTS and RTS

- **Controlled Transmission and Reception**
  - Simplex Mode
  - Flow Control Mode



## Simplex Mode

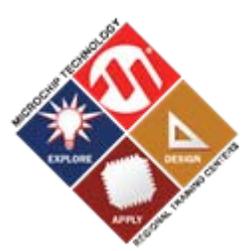


## Flow Control Mode



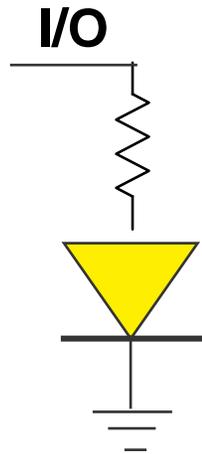
# UART - IrDA<sup>®</sup> Support

- **Built-in IrDA Encoder and Decoder**
  - The IrDA Encoder and Decoder can be enabled by setting **UxMODE<IREN>** bit
- **16x Baud Clock is generated to support external IrDA Encoder and Decoder**
  - 16x Baud Clock is generated by setting **UxMODE<UEN1 : UEN0>** to '11'
  - The generated Baud Clock is available on the pin **BCLKx**

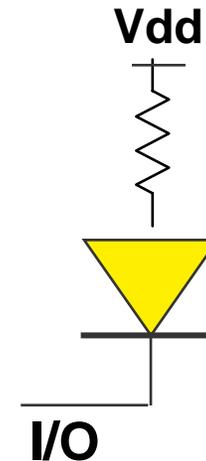


# Optical Communication

- **Two ways to connect LED**



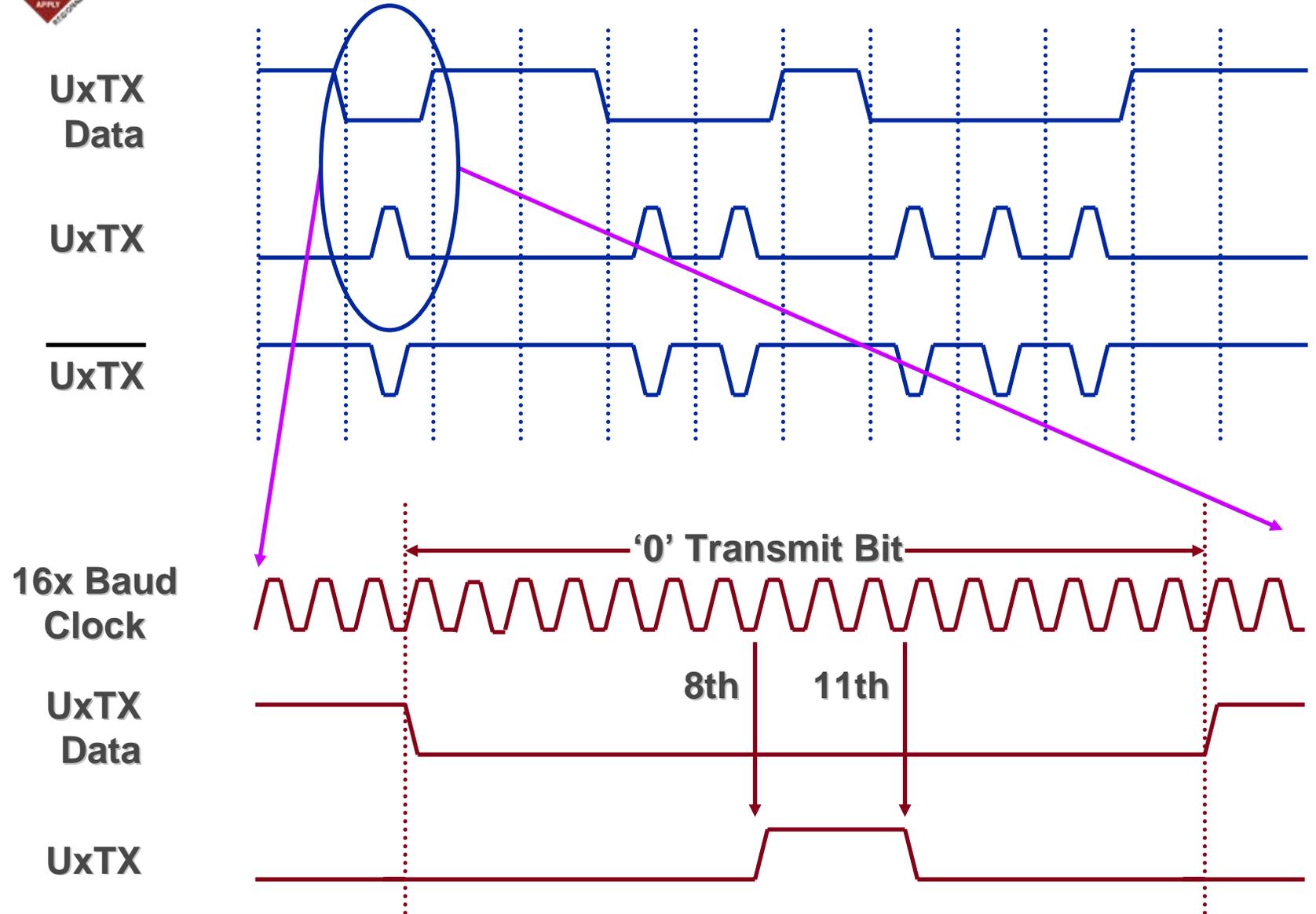
**Logic High = LED On**  
**Logic Low = LED off**



**Logic Low = LED On**  
**Logic High = LED off**



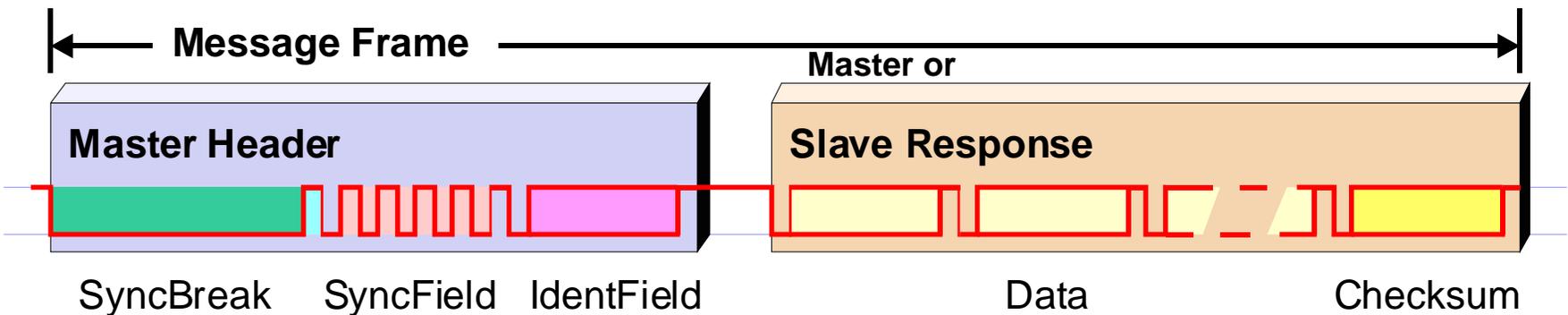
# IrDA<sup>®</sup> Mode Bit Encoding





# UART - LIN Support

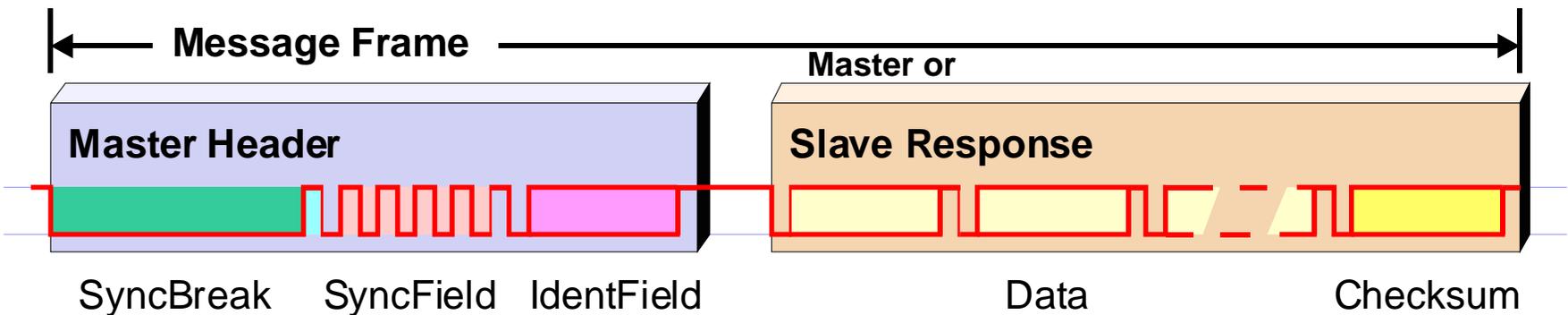
- **Transmission of Break Characters**
  - A Break character can be transmitted by setting the `UxSTA<UTXBRK>` bit for at least 13 bit times
- **Autobaud Detection**

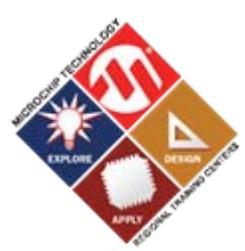




# LIN - Protocol

- **UxSTA<UTXBRK> Enables Transmitting Break condition on the bus (Drive low TX pin for 13-Bit Time)**
- **UxMODE<ABAUD> Enables to measure the baud-rate using Input Capture unit during SYNCH Field (0x55).**





# LAB 6: Working with UART

- **Goal:**

- Understand Configuration of UART module
- Understand Transmit and Receive interrupts
- Write a software to Transmit and Receive data using UART

- **To Do:**

- Look into the Hand out provided

- **Expected Result:**

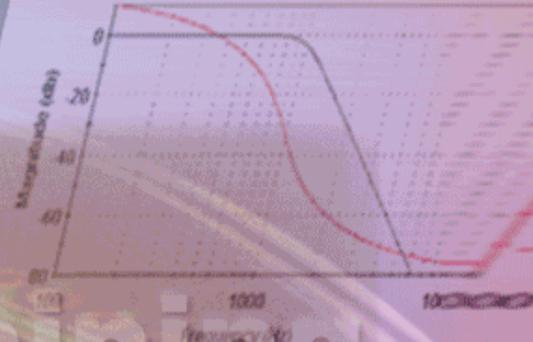
- LED D3 flashes at different rates to indicate lower CPU time spent handling UART data using FIFO buffer

# HANDS-ON

# Training



# I<sup>2</sup>C™



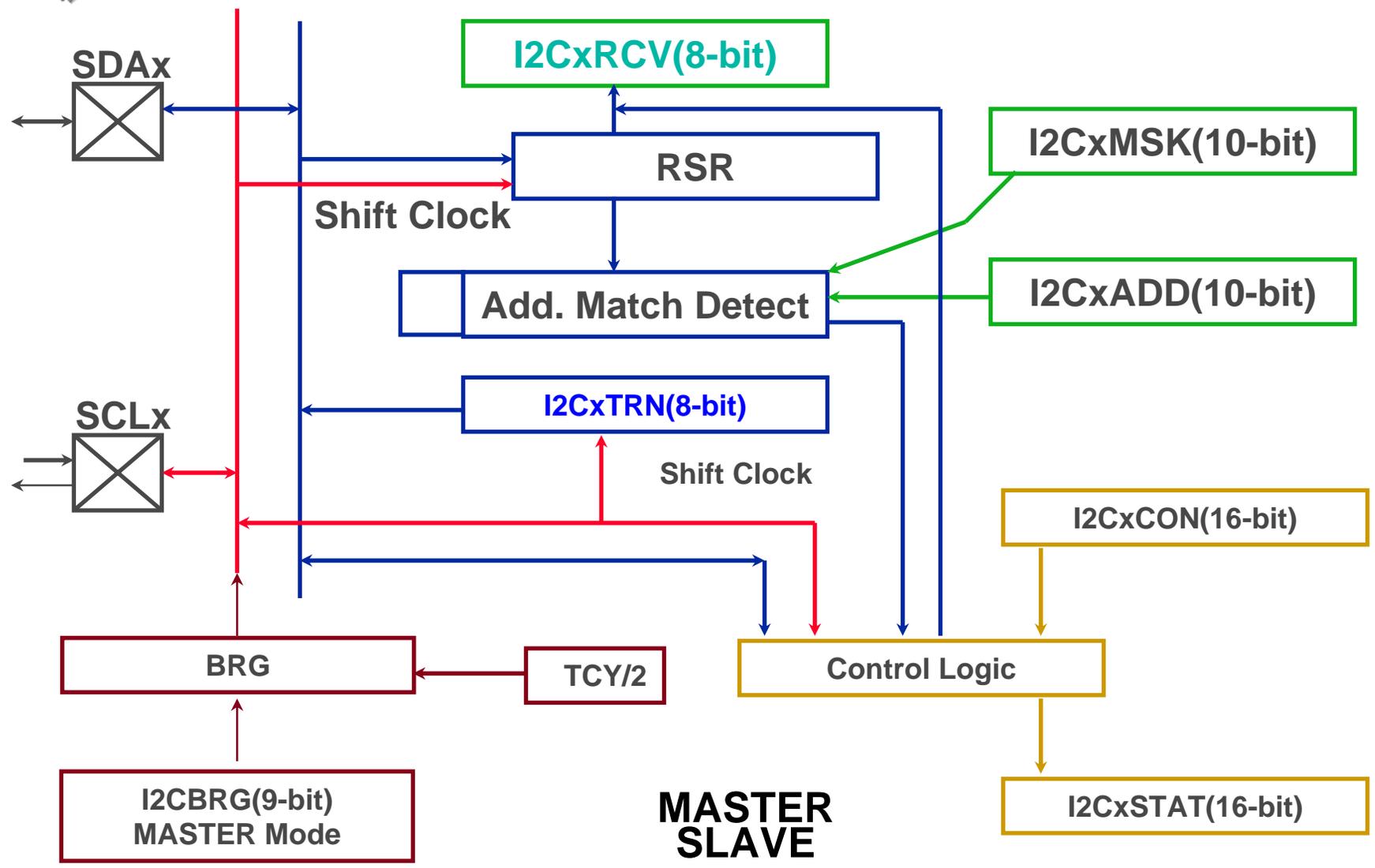


# I<sup>2</sup>C™ Overview

- **Synchronous 2-wire communication**
- **Master-Slave protocol**
- **Serial interface (Half-duplex):**
  - SDA: Data line to & from the master
  - SCL: Clock line, master controlled



# I<sup>2</sup>C™



**MASTER  
SLAVE**



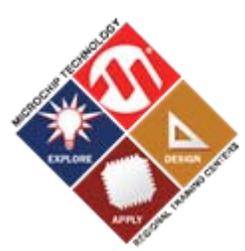
# Address Masking

## What is it?

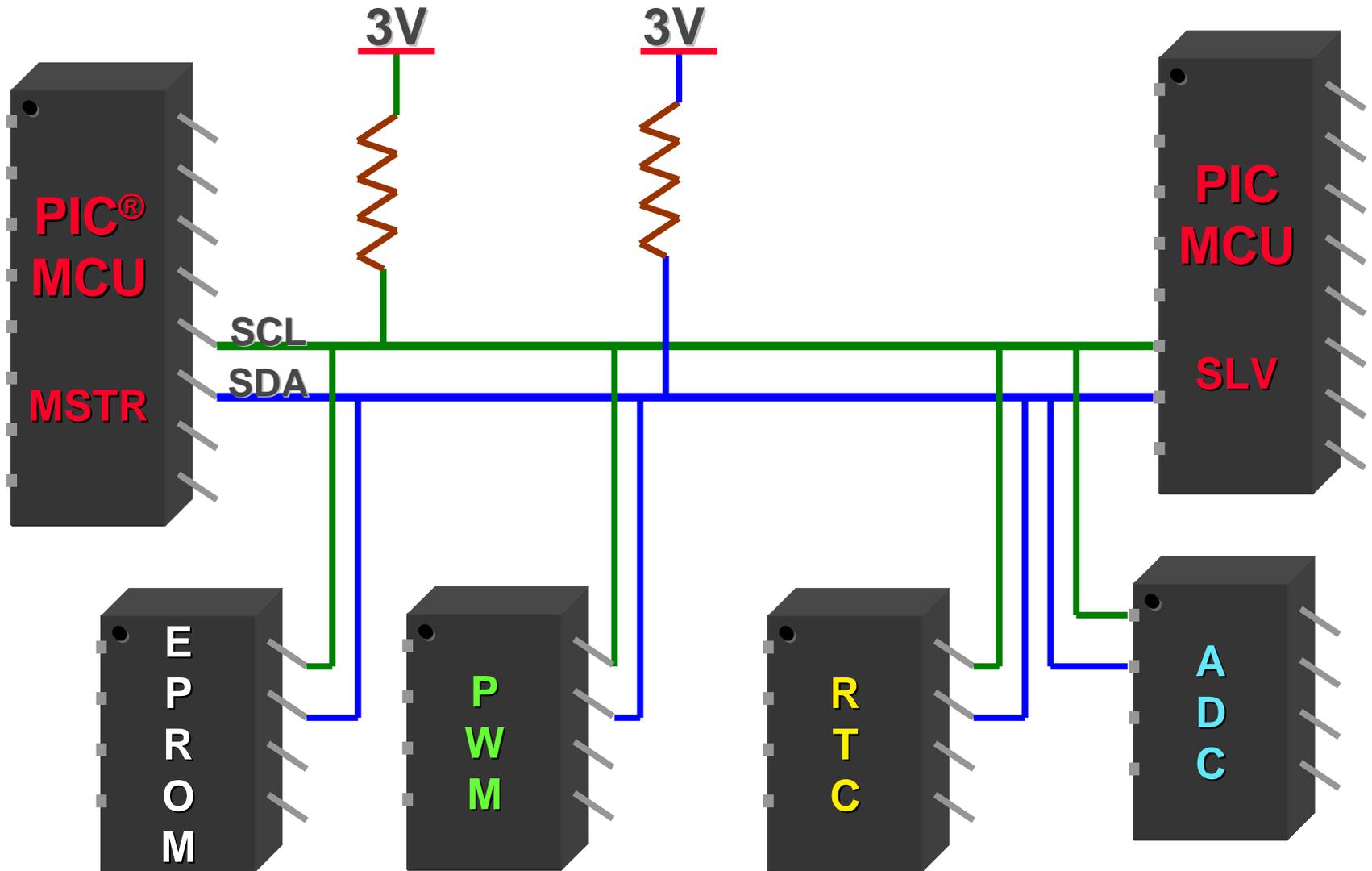
- It allows a PIC<sup>®</sup> MCU to ACK more than one address.

## What is it good for?

- IPMI: Intelligent Platform Management Interface
- Embedded system device networking: using a PIC MCU as multiple I<sup>2</sup>C<sup>™</sup> devices

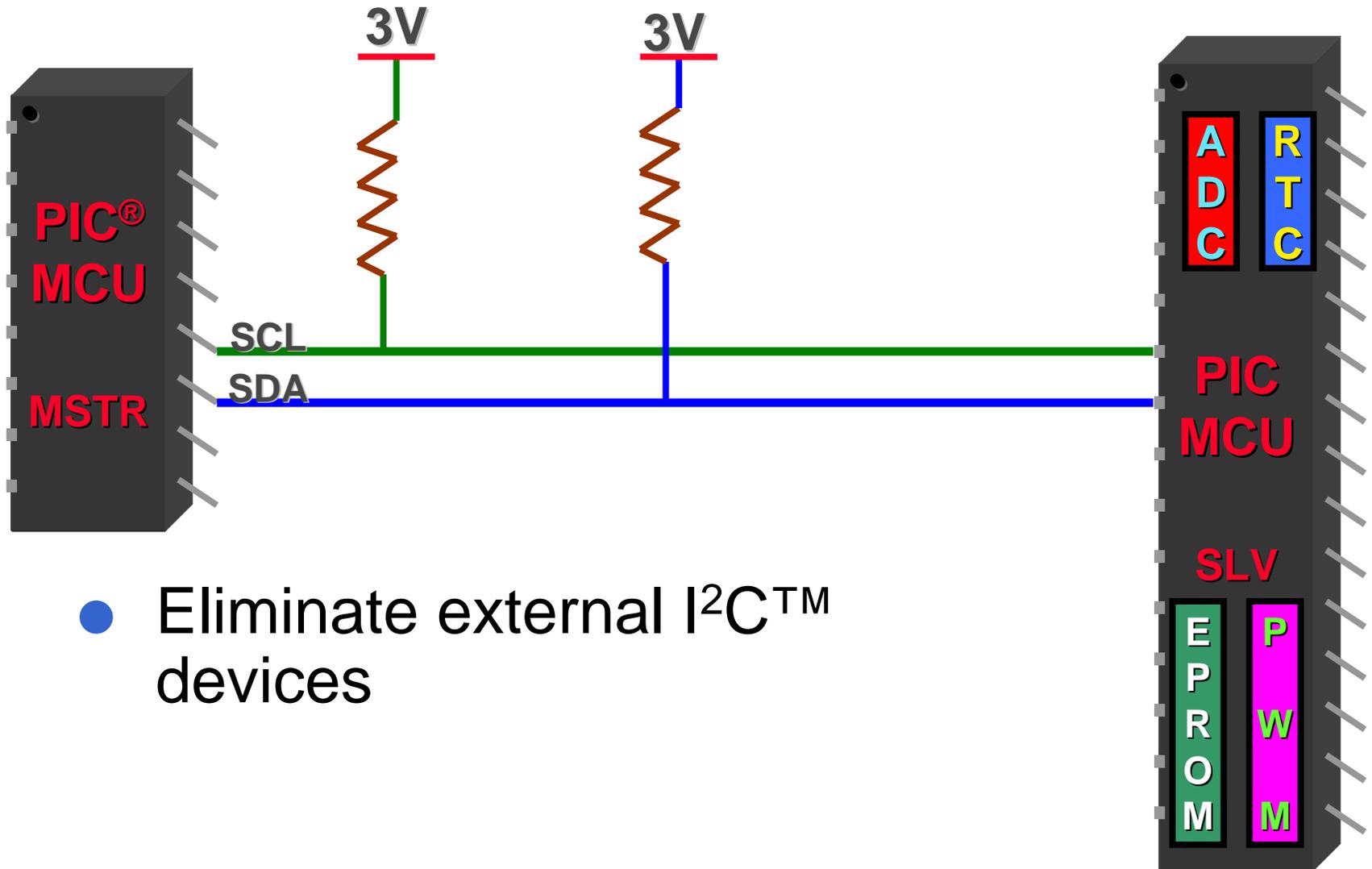


# Embedded System Device Networking





# Embedded System Device Networking

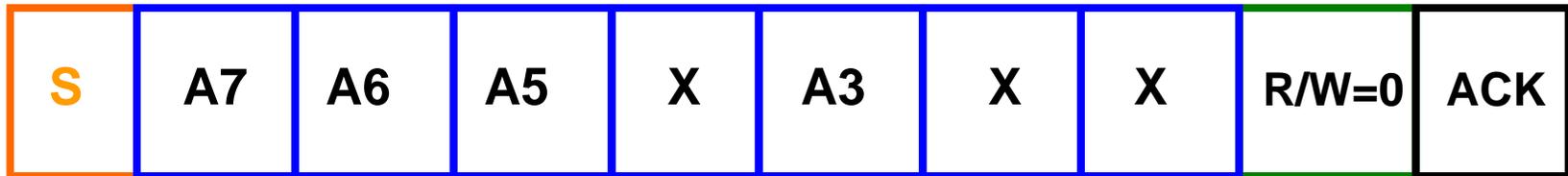


- Eliminate external I<sup>2</sup>C™ devices



# Address Masking Example

- By setting the ADMSK bits in SSPCON2, we mask the address bits.
- In this case  $ADMSK[7:1]=0001011$

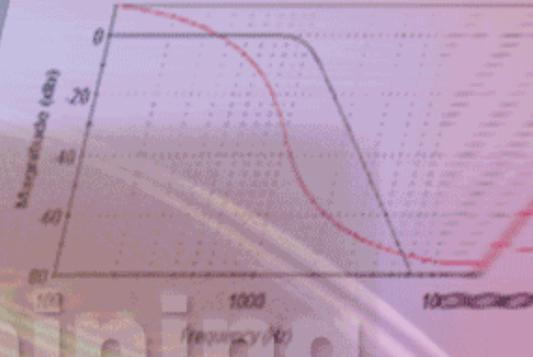


## All addresses that will be ACKed:

- A7.A6.A5.0.A3.0.0
- A7.A6.A5.0.A3.1.0
- A7.A6.A5.1.A3.0.0
- A7.A6.A5.1.A3.1.0
- A7.A6.A5.0.A3.0.1
- A7.A6.A5.0.A3.1.1
- A7.A6.A5.1.A3.0.1
- A7.A6.A5.1.A3.1.1

# HANDS-ON

# Training



# SPI





# SPI - Overview

- **Serial transmission and reception of 8/16-bit data**
- **Full-duplex, synchronous communication**
- **4-wire interface**
- **Supports 4 different clock formats and serial clock speeds up to 10 Mbps**
- **Buffered Transmission and Reception**







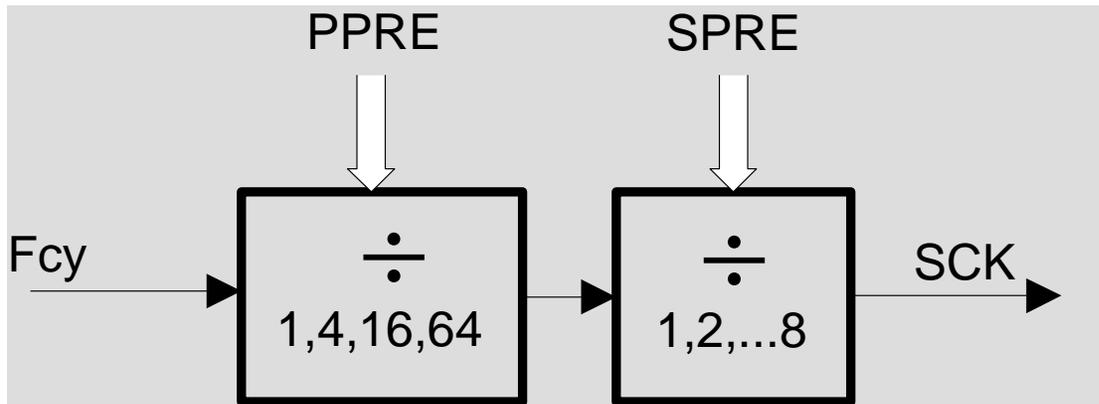
# SPI Clock

- **CKP: Clock Polarity Select**

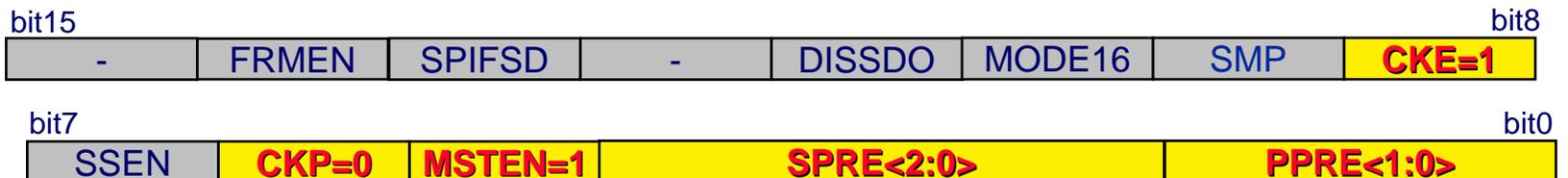
- **0**: Idle State of CLK is LOW
- **1**: Idle State of CLK is HIGH

- **CKE: Clock Edge Select**

- **0**: Transmit data on Idle to Active Edge
- **1**: Transmit data on Active to Idle Edge

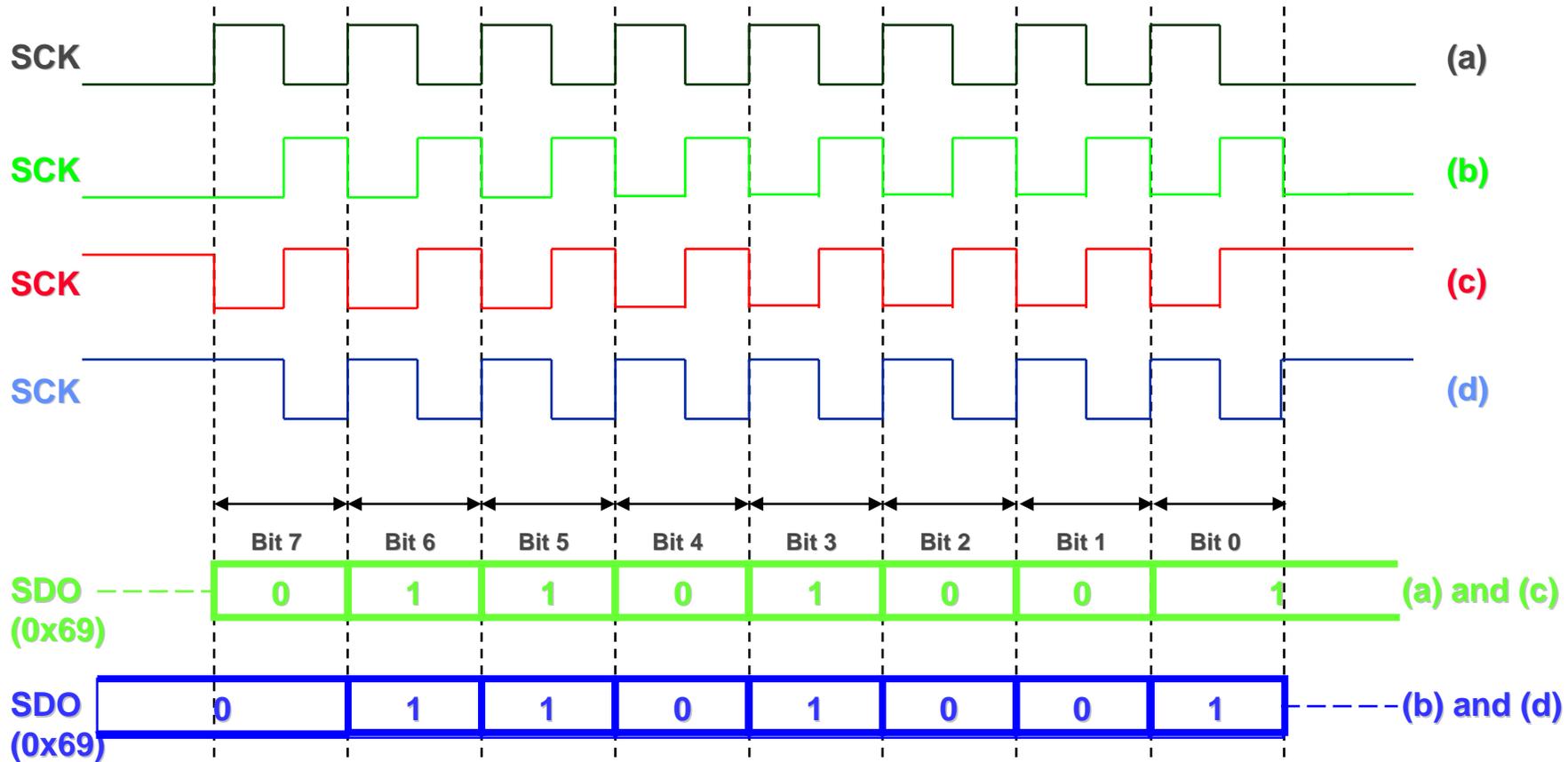


## SPIxCON Register





# SPI - Serial Clock Formats



(a) CKP = 0, CKE = 0

(b) CKP = 0, CKE = 1

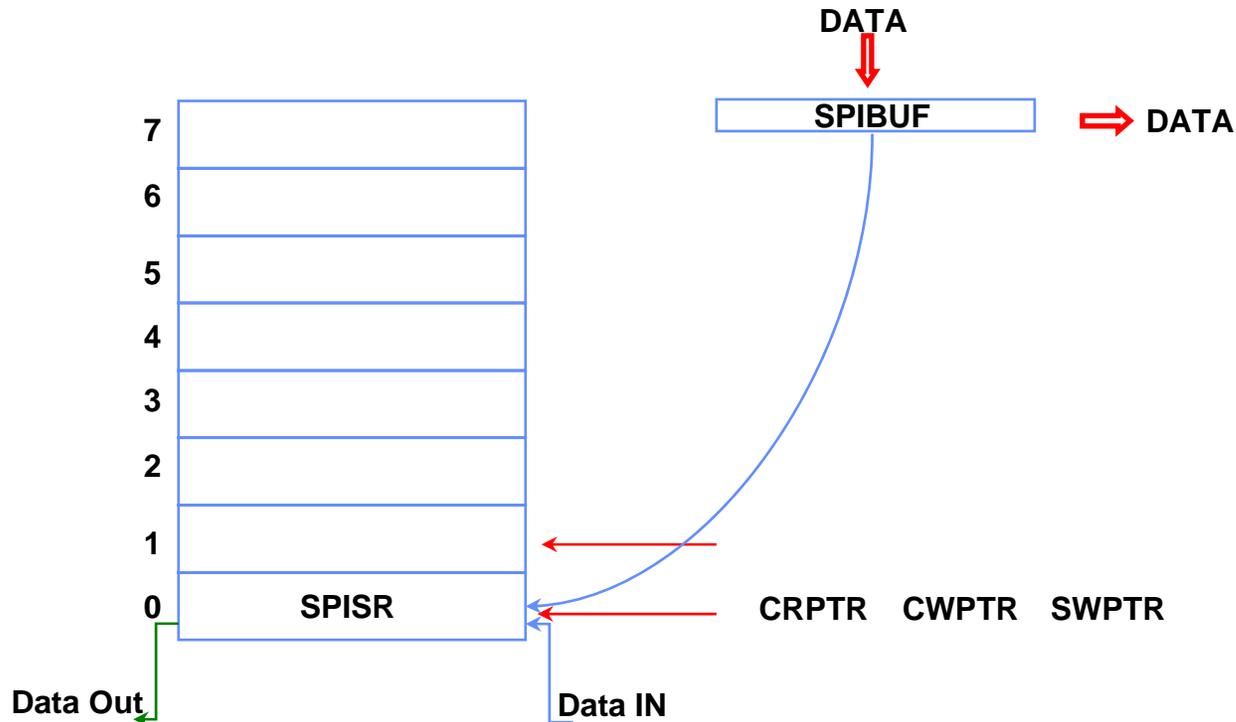
(c) CKP = 1, CKE = 0

(d) CKP = 1, CKE = 1



# Buffer Operation in Enhanced Mode

- The 8 level buffer can act as 8\*8-bit array or 8\*16-bit array, decided by the bit `SPIxCON<MODE16>`



~~Now CWPTR = SWPTR~~

# HANDS-ON

# Training

# Special Features

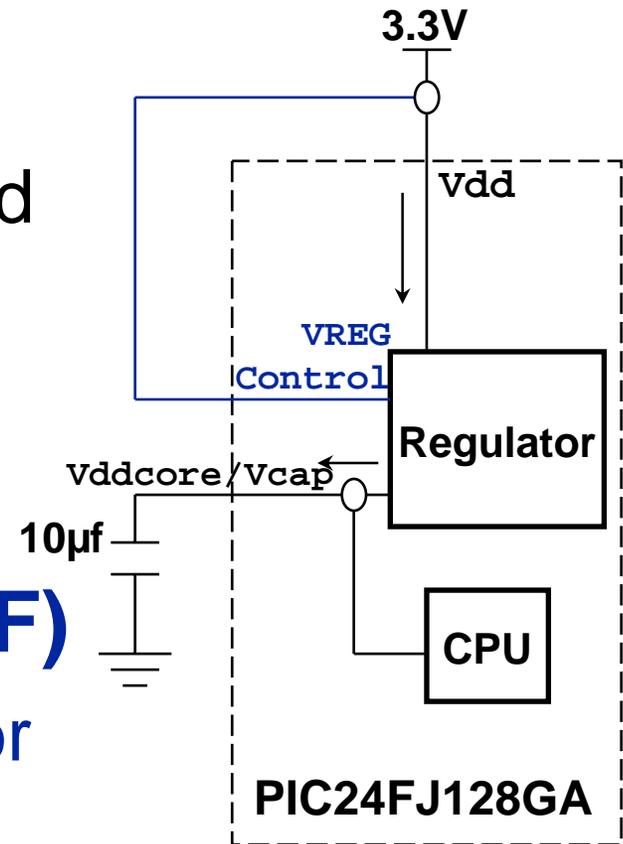




# Powering CPU

## PIC24FJ, PIC24HJ, dsPIC33FJ

- **Internal on-chip regulator**
  - 2.5V nominal for the CPU
  - 3.3V nominal for the I/O and peripheral voltage
- **Low ESR cap required**
  - 10 $\mu$ f recommended
- **VREG Control pin (PIC24F)**
  - Enable/disable the regulator
- **BOR**





# Power Management

- **Power management options:**
  - Lower the supply voltage
  - Lower the clock speed
    - **Postscaler**
    - **Clock switching**
  - Disable unused peripherals
    - **Minimal power saving**
  - Execute `pwrsav` instruction
    - **SLEEP**
    - **IDLE**
  - DOZE mode

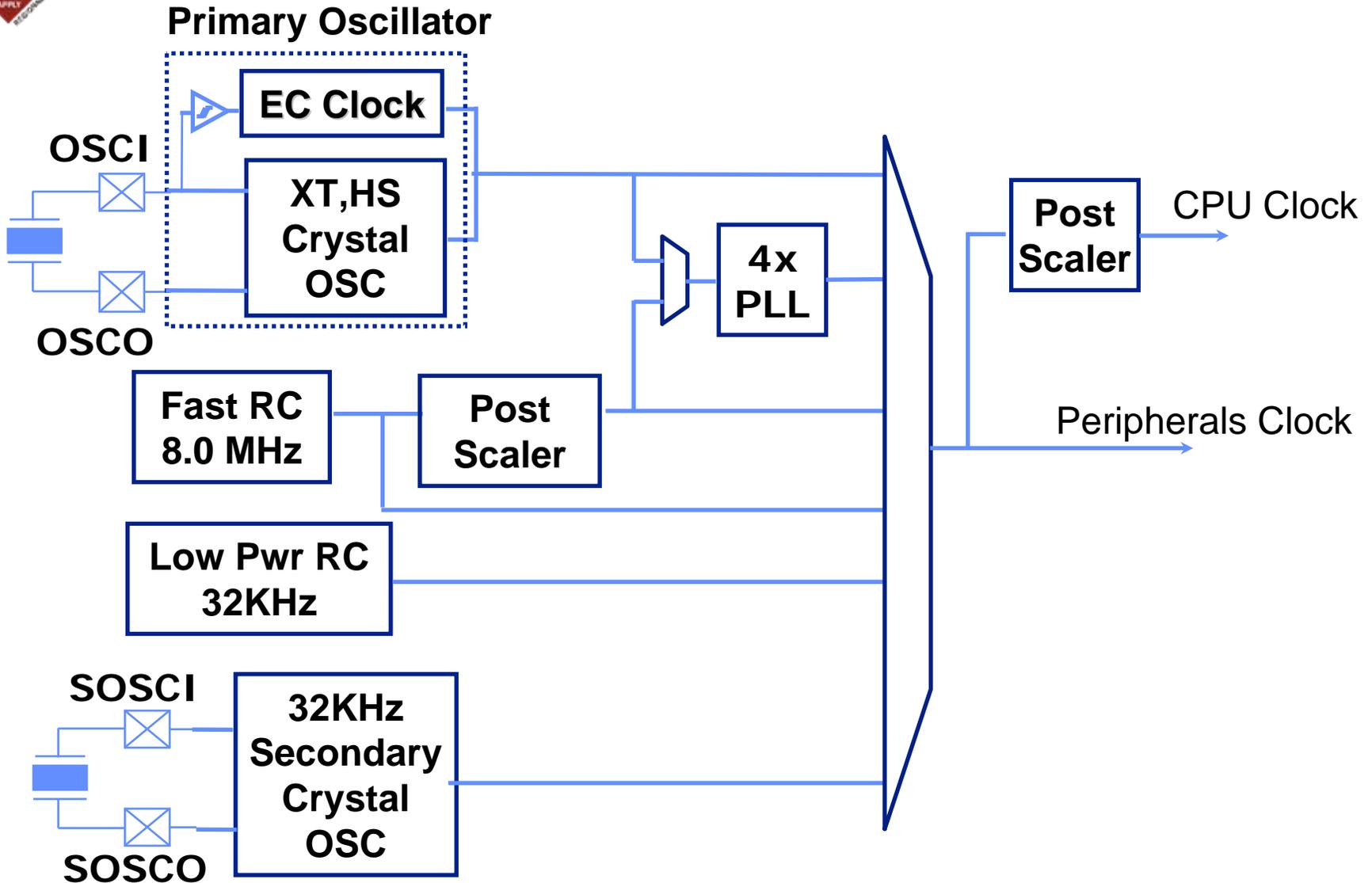


# Power Management

- **Exiting Power Saving mode**
  - System Reset
  - Watch Dog Timer
  - Any enabled interrupt
    - **Interrupt Priority Level  $\leq$  CPU IPL**
      - Execute the next instruction
    - **Interrupt Priority Level  $>$  CPU IPL**
      - Execute the interrupt handler
- **How to know it is a wake up**
  - Check **RCON<SLEEP>** and **RCON<IDLE>** bits

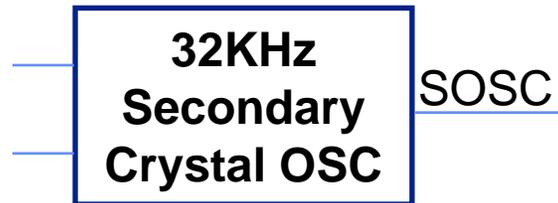
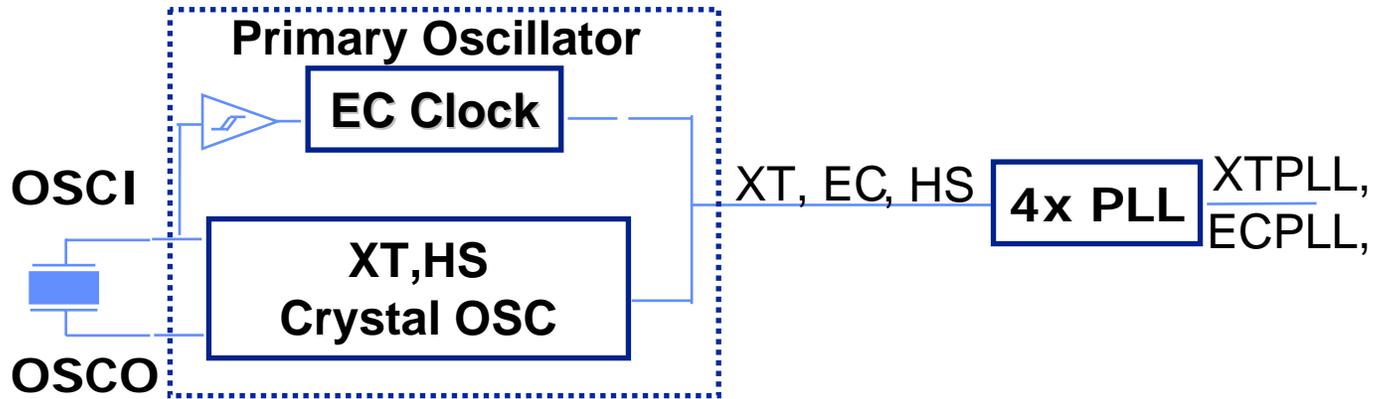


# Clock Sources





# Clock Sources

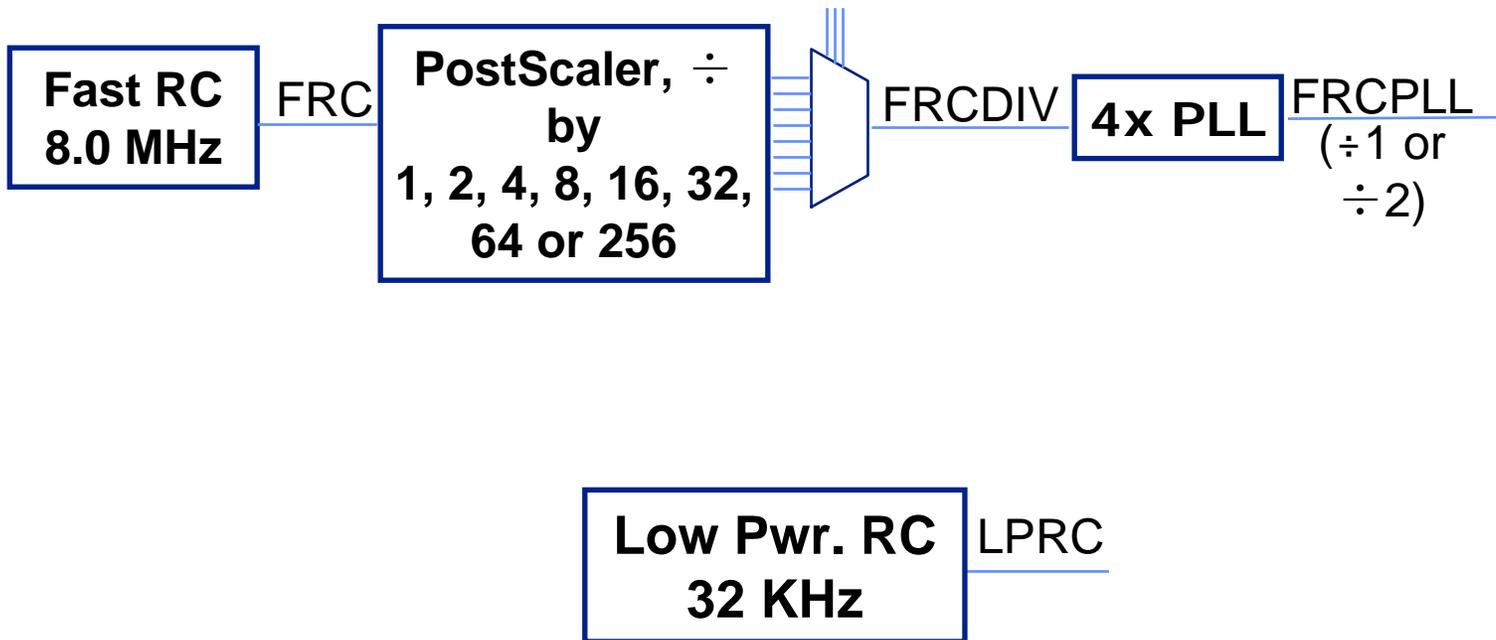


- It is 4x PLL only in PIC24F. In other 16 bit controllers it can be configured into 4x, 8x or 16x PLL



# Clock Sources

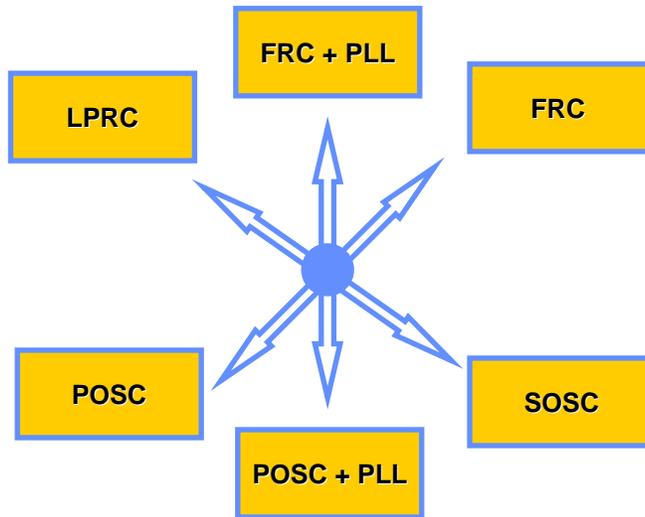
CLKDIV<RCDIV2:RCDIV0>





# Power Saving Technology

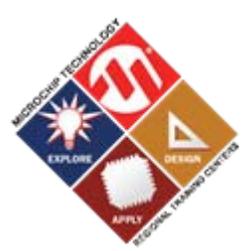
## On-the-Fly Clock Switching



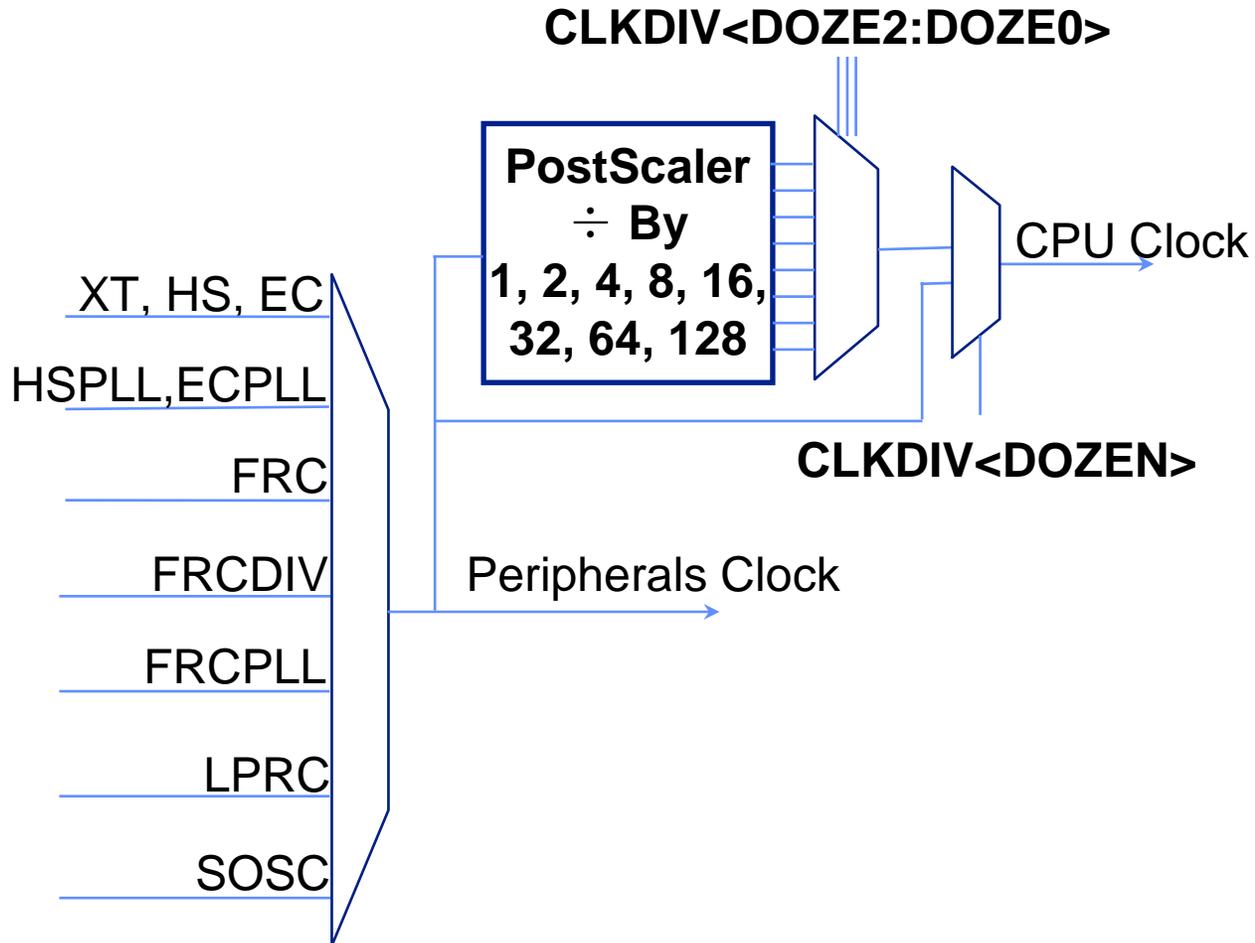
- `OSCCON<OSWEN>` bit to enable On Fly Clock Switching and `OSCCON<NOSC2:NOSC0>` bits to select the new clock source

## ● Instruction-Based Power Saving Modes

- Sleep
- Idle



# Doze Mode





# Fail-Safe Clock Monitor

- **What happens on Oscillator Failure?**
  - Device switches automatically to the FRC oscillator if FSCM is enabled
    - **FSCM controlled by the bit `FOSC<FCKSM>`**
  - Oscillator Failure Trap is generated
  - Code execution vectors to the Oscillator Fail trap handler, if one exists. Here the application should:
    - **Clear the Clock Fail bit, `OSCCON<CF>`**
    - **Clear the Osc FAIL trap flag, `INTCON1<OSCFAIL>`**
    - **Switch to required Clock Source form FRC**
    - **Execute a RETFIE**



# Resets

## ● PIC24 RESETS

- Power-on Reset (POR)
- Pin Reset (MCLR)
- RESET Instruction (SWR)
- Watchdog Timer Reset (WDT)
- Brown-out Reset (BOR)
- Trap Conflict Reset (TRAPR)
- Illegal Opcode Reset (IOPUWR)
- Uninitialized W Register Reset (UWR)
- Configuration word mismatch Reset (CM)

## ● For all the resets PC vectors to address 0



# Watchdog Timer

- **Recovers from software malfunction**
- **Resets MCU if not attended on-time**
  - Software must clear it periodically (**CLRWDT**)
- **Programmable timeout period**
  - 1 ms to 131 s typical
- **Fuse time programmable prescaler and postscaler**
- **Both Fuse time and Run time Enable available**
- **Option to select Windowed WDT**
  - **CLRWDT** should be used only in last quadrant
  - If **CLRWDT** is used earlier, resets the device



# Programming Support

- **In Circuit Serial Programming™ (ICSP™):**
  - Device can be programmed in circuit
  - Useful to Combine Programming and Final test
- **Enhanced In Circuit Serial Programming™ (Enhanced ICSP™) capability:**
  - Uses programming executive for faster programming
- **Run Time Self Programming (RTSP):**
  - Device can program its own Flash memory
  - Useful for Remote code updates
- **JTAG Interface**
  - Programming support through Serial Vector Format files
  - Provides for boundary scan



# Summary

- **We discussed about Architecture overview**
  - 16-bit Architecture
  - Programmers' Model
  - Addressing Modes
  - Interrupt Handling and latency
- **Discussed about Basic Peripherals**
  - I/O Ports
  - ADC
  - Comparators, Voltage Reference
  - Timers
  - Input Capture
  - Output Compare & PWM
  - USART
  - I<sup>2</sup>C<sup>TM</sup>
  - SPI
- **Special features of 16-bit microcontrollers**
  - Oscillator and power saving modes



# Summary

- **Carried out the Labs on**
  - **Using MPLAB and C30**
  - **PSV initialization and usage**
  - **Interrupt handling and its priority setting**
  - **Initialization and ADC conversion**
  - **32-bit Timer configuration and its usage**
  - **UART configuration and use of FIFO**



# References

- PIC24F, PIC24H, dsPIC33 F Datasheets
- PIC24F family reference manual
- dsPIC30F family reference Manual
- Explorer 16 User's Guide
- C30 Compiler User's Guide
- Other 16-bit Courses
  - **204 ADV: Advanced 16-bit peripherals**
  - **104 DSP: DSP Engine and Libraries**



**Thank You**

**Please complete the review form**



## Trademarks

**The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KeeLoq, microID, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, PowerSmart, rfPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.**

**AmpLab, FilterLab, Migratable Memory, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.**

**Analog-for-the-Digital Age, Application Maestro, dsPICDEM, dsPICDEM.net, dsPICworks, ECAN, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, Linear Active Thermistor, Mindi, MiWi, MPASM, MPLIB, MPLINK, MPSIM, PICKit, PICDEM, PICDEM.net, PICLAB, PICtail, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, rfPICDEM, Select Mode, Smart Serial, SmartTel, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.**

**SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.**

**All other trademarks mentioned herein are property of their respective companies.**