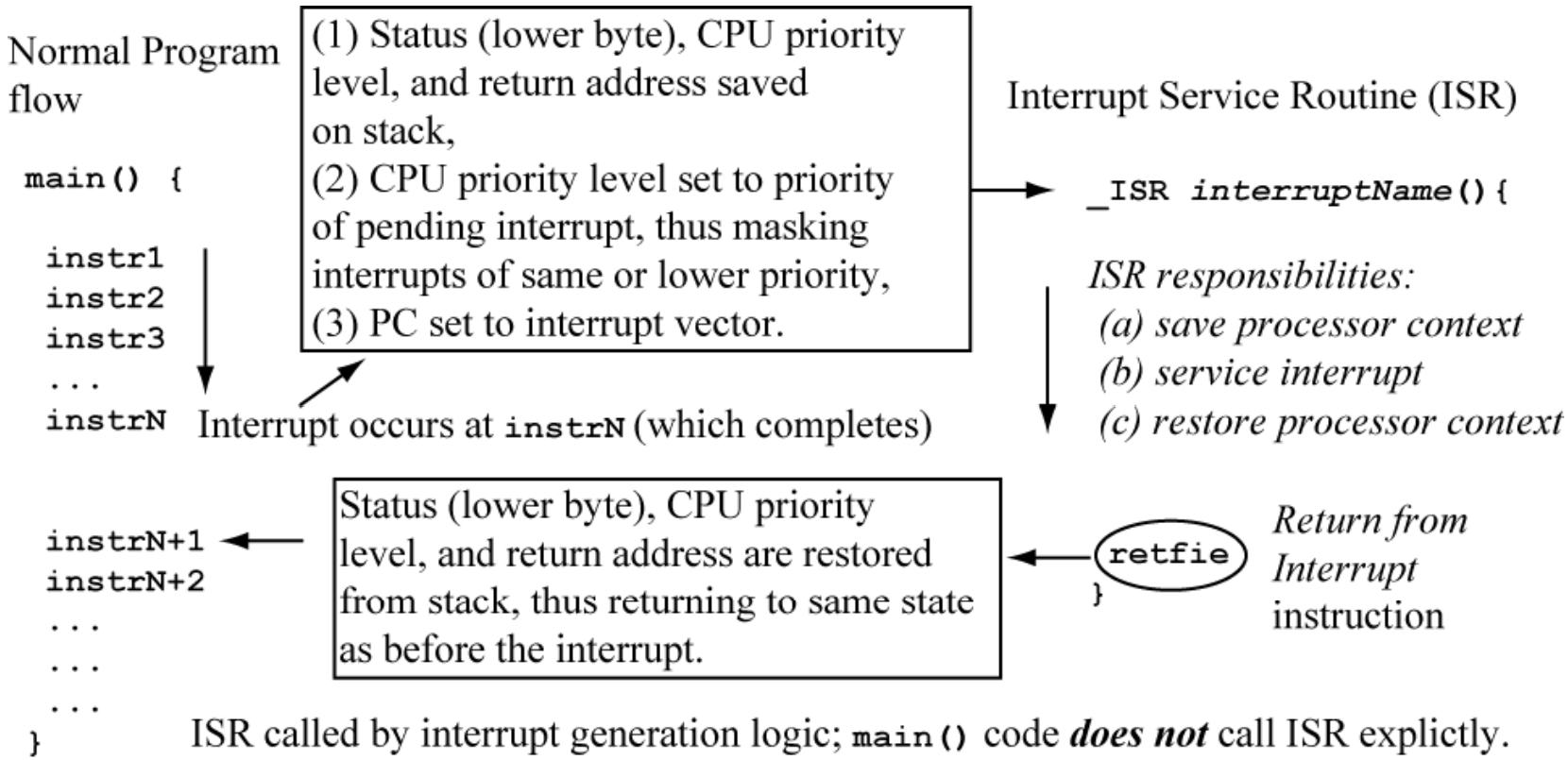
Polled IO versus Interrupt Driven IO

- Polled Input/Output (IO) – processor continually checks IO device to see if it is ready for data transfer
  - Inefficient, processor wastes time checking for ready condition
  - Either checks too often or not often enough
- Interrupt Driven IO – IO device interrupts processor when it is ready for data transfer
  - Processor can be doing other tasks while waiting for last data transfer to complete – very efficient.
  - All IO in modern computers is interrupt driven.

PIC24 µC Interrupt Operation



The normal program flow (main) is referred to as the foreground code. The **interrupt service routine** (ISR) is referred to as the background code.

From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

Vector Table

Decreasing Natural Order Priority

Reset - `goto` Instruction	0x000000
Reset - `goto` Address	0x000002
Reserved	0x000004
Oscillator Fail Trap Vector	0x000006
Address Error Trap Vector	0x000008
Stack Error Trap Vector	0x00000A
Math Error Trap Vector	0x00000C
DMAC Error Trap Vector	0x00000E
Reserved	
Reserved	
Interrupt Vector 0	0x000014
Interrupt Vector 1	0x000016
~	
Interrupt Vector 116	0x0000FC
Interrupt Vector 117	0x0000FE
Reserved	0x000100
Reserved	0x000102
Reserved	0x000104
Oscillator Fail Trap Vector	0x000106
Address Error Trap Vector	0x000108
Stack Error Trap Vector	0x00010A
Math Error Trap Vector	0x00010C
DMAC Error Trap Vector	0x00010E
Reserved	
Reserved	
Interrupt Vector 0	0x000114
Interrupt Vector 1	0x000116
~	
Interrupt Vector 116	0x0001FC
Interrupt Vector 117	0x0001FE
Start of Code	0x000200

Interrupt Vector Table (IVT)

Alternate Interrupt Vector Table (AIVT)

This contains the starting address of the ISR for each interrupt source.

Figure redrawn by author from Figure 6-1 of the PIC24 FRM datasheet (DS70224B), Microchip Technology, Inc.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Interrupt Sources

IVT Address	Vector Num	PIC24 Compiler Name	Vector Function
0x000006	1	`_OscillatorFail`	Oscillator Failure
0x000008	2	`_AddressError`	Address Error
0x00000A	3	`_StackError`	Stack Error
0x00000C	4	`_MathError`	Math Error
0x000014	8	`_INT0Interrupt`	INT0 – External Interrupt
0x000016	9	`_IC1Interrupt`	IC1 – Input Capture 1
0x000018	10	`_OC1Interrupt`	OC1 – Output Compare 1
0x00001A	11	`_T1Interrupt`	T1 – Timer1 Expired
0x00001E	13	`_IC2Interrupt`	IC2 – Input Capture 2
0x000020	14	`_OC2Interrupt`	OC2 – Output Compare 2
0x000022	15	`_T2Interrupt`	T2 – Timer2 Expired
0x000024	16	`_T3Interrupt`	T3 – Timer3 Expired
0x000026	17	`_SP1ErrInterrupt`	SPI1E – SPI1 Error
0x000028	18	`_SP1Interrupt`	SPI1 – SPI1 transfer done
0x00002A	19	`_U1RXInterrupt`	U1RX – UART1 Receiver
0x00002C	20	`_U1TXInterrupt`	U1TX – UART1 Transmitter
0x00002E	21	`_ADC1Interrupt`	ADC1 – ADC 1 convert done
0x000034	24	`_SI2C1Interrupt`	SI2C1 – I2C1 Slave Events
0x000036	25	`_MI2CInterrupt`	MI2C1 – I2C1 Master Events
0x00003A	27	`_CNInterrupt`	Change Notification Interrupt
0x00003C	28	`_INT1Interrupt`	INT1 – External Interrupt
0x000040	30	`_IC7Interrupt`	IC7 – Input Capture 7
0x000042	31	`_IC8Interrupt`	IC8 – Input Capture 8
0x00004E	37	`_INT2Interrupt`	INT2 – External Interrupt
0x000096	73	`_U1ErrInterrupt`	U1E – UART1 Error

Serial data has arrived

CNx Pin has changed state

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Interrupt Priorities

An interrupt can be assigned a priority from 0 to 7.

Normal instruction execution is priority 0.

An interrupt MUST have a higher priority than 0 to interrupt normal execution. Assigning a priority of 0 to an interrupt masks (disables) than interrupt.

An interrupt with a higher priority can interrupt a currently executing ISR with a lower priority.

If simultaneous interrupts of the SAME priority occur, then the interrupt with the LOWER VECTOR NUMBER (is first in the interrupt vector table) has the higher *natural* priority. For example, the INT0 interrupt has a higher natural priority than INT1.

Enabling an Interrupt

Each interrupt source generally has FLAG bit, PRIORITY bits, and an ENBLE bit.

The flag bit is set whenever the interrupt condition occurs, which varies by the interrupt.

The priority bits set the interrupt priority.

The enable bit must be a ‘1’ AND the interrupt priority > 0 for the ISR to be executed (interrupt is enabled). (NOTE: the interrupt does not have to be a enabled for the flag bit to be set!!!!!).

One of the things that must be done by the ISR is to clear the flag bit, or else the PIC24 CPU will get stuck in an infinite loop executing the ISR.

By default, all priority bits and enable bits are ‘0’, so interrupt ISRs are disabled from execution.

Traps vs. Interrupts

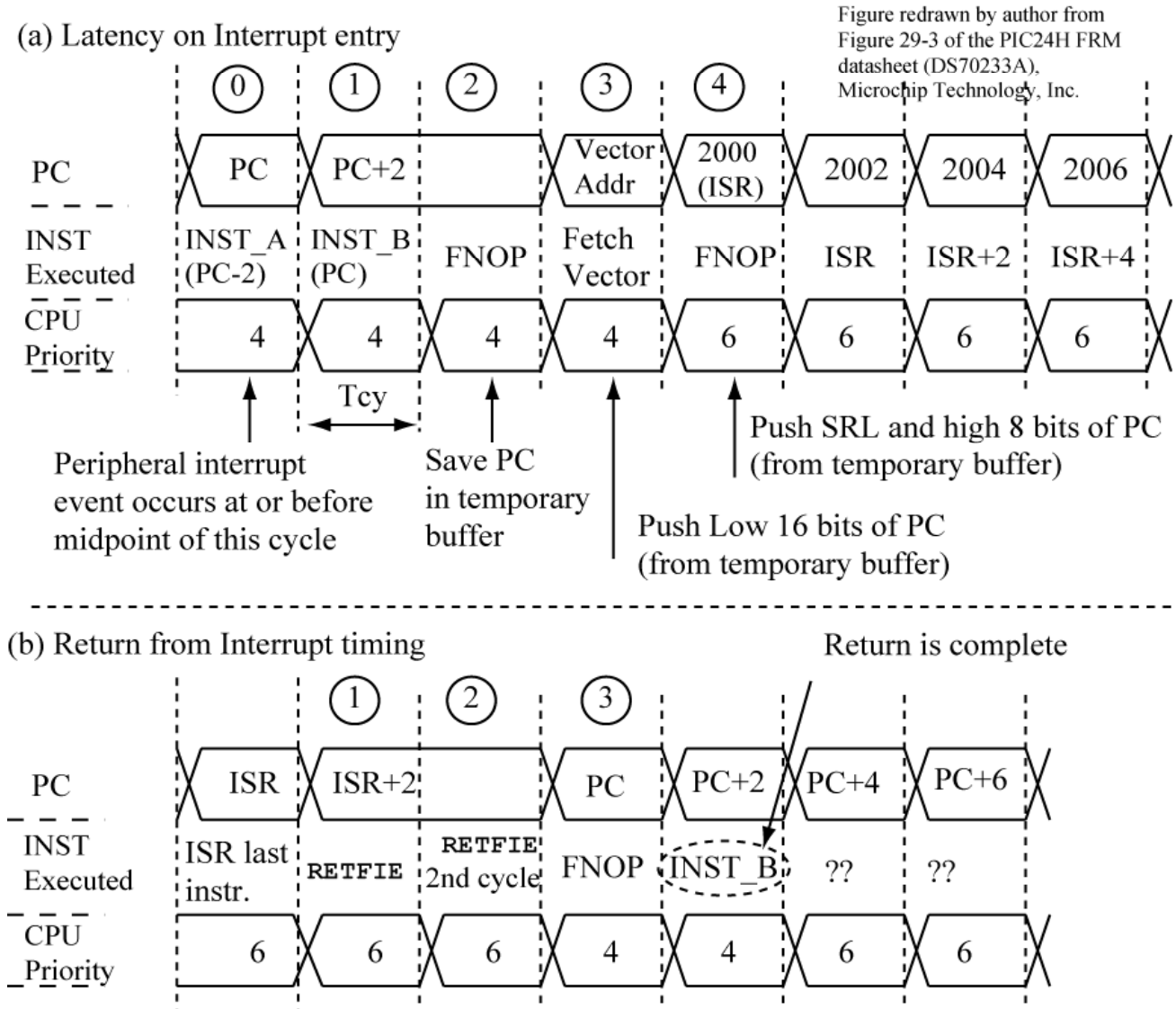
A Trap is a special type of interrupt, is non-maskable, has higher priority than normal interrupts. **Traps are always enabled!**

Hard trap: CPU stops after instruction at which trap occurs

Soft trap: CPU continues executing instructions as trap is sampled and acknowledged

Trap	Category	Priority	Flag(s)
Oscillator Failure	Hard	14	_OSCFAIL (oscillator fail, INTCON1<1>), _CF (clock fail, OSSCON<3>)
Address Error	Hard	13	_ADDRERR (address error, INTCON1<3>)
Stack Error	Soft	12	_STKERR (stack error, INTCON1<2>)
Math Error	Soft	11	_MATHERR (math error, INTCON1<4>)
DMAC Error	Soft	10	_DMACERR (DMA conflict write, INTCON1<5>)

Interrupt Latency



ISR Entry: Number of cycles from interrupt until 1st instruction of ISR is executed.

ISR Exit:

From RETFIE to program resumed.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

ISR Overhead

- **Ientry**: Number of instruction cycles for ISR entry (four on the PIC24 μC).
- **Ibody**: Number of instruction cycles for the ISR body (not including `retfie`).
- **Iexit**: Number of instruction cycles for ISR exit (three on the PIC24 μC).
- **Fisr**: Frequency (number of times per second) at which the ISR is triggered.
- **Tisr**: The ISR triggering period, which is 1/Fisr. For example, if an ISR is executed at 1 KHz, Tisr is 1 ms.

ISR Overhead (cont)

Percentage of CPU time taken up by one ISR:

ISR% = [(Ientry + Ibody + Iexit) x Fisr]/Fcy x 100

ISR CPU Percentage for FCY = 40 MHz, IBODY = 50 instr. cycles

Tisr = 10 ms	**Tisr = 1 ms**	**Tisr = 100 μs**	**Tisr = 10 μs**
0.01%	0.14%	1.43%	14.3%

GOLDEN RULE: An ISR should do its work as quickly as possible. When an ISR is executing, it is keeping other ISRs of equal priority and lower from executing, as well as the main code!

Interrupt Vectors in Memory

(b) MPLAB Program Memory

Line	Address	Opcode	Label	Dis
1	0000	040C02		goto _reset
2	0002	000000		nop
3	0004	000D8C		_DefaultInterrupt
4	0006	000D8C		_DefaultInterrupt
5	0008	000D8C		_DefaultInterrupt
6	000A	000D8C		_DefaultInterrupt
7	000C	000D8C		_DefaultInterrupt

Unhandled interrupts use **_DefaultInterrupt**

Math Error Trap Vector

The compiler uses the **_DefaultInterrupt** function as the default ISR. If an interrupt is triggered, and the ISR is the **_DefaultInterrupt**, then the user did not expect the interrupt to occur. This means the interrupt is 'unhandled'. We have written our own **_DefaultInterrupt** that prints diagnostic information since this is an unexpected occurrence.

(a) Code for default interrupt handler

```
_PERSISTENT const char* sz_lastError;
_PERSISTENT char* sz_lastTimeoutError;
_PERSISTENT INTTREGBITS INTTREGBITS_last;
#define u16_INTTREGlast \
          BITS2WORD(INTTREGBITS_last)

void _ISR _DefaultInterrupt(void) {
  u16_INTTREGlast = INTTREG;
  reportError("Unhandled interrupt, ");
}

void reportError(const char*
                 sz_errorMessage) {
  sz_lastError = sz_errorMessage;
  asm ("reset");
}

void printResetCause(void) {
...print reset cause, see Chapter 8...
  if (u16_INTTREGlast != 0) {
    outString("Error trapped: ");
    outString(sz_lastError);
    if (sz_lastInterrupt != 0) {
      outString("Priority: ");
      outUint8(INTTREGBITS_last.ILR);
      outString(" , Vector number: ");
      outUint8(INTTREGBITS_last.VECNUM);
    }
    outString("\n\n");
    sz_lastError = NULL;
    u16_INTTREGlast = 0;
  }
}
```

`_PERSISTENT` error variables used for tracking errors across resets.

This allows treating the `INTTREGBITS_last` structure as a single `uint16` value.

`_DefaultInterrupt` is the name of the default ISR used by the PIC24 compiler. Our version saves the interrupt cause (`INTTREG`) then does a software reset.

Saves the error message, then does a software reset

After reset, `printResetCause()` prints the error message.

Output error message saved from last reset

If last reset was caused by an unhandled interrrupt, print the priority (`ILR`) and vector number (`VECNUM`)

Clear `_PERSISTENT` error variables.

Our _DefaultInterrupt ISR

Used for all interrupts when you do not provide an ISR. Our version saves the interrupt source, does a sofware reset, then interrupt source is printed.

Output from the _DefaultInterrupt ISR

(a) Simplified test code (*trap_test.c*) to generate a Math Error Trap

```
int main (void) {
  volatile uint8 u8_zero;
  configBasic(HELLO_MSG);
  while (1) {
    outString("Hit a key to start divide by zero test...");
    inChar();
    outString("OK. Now dividing by zero.\n");
    u8_zero = 0;
    u8_zero = 1/u8_zero;
    doHeartbeat();
  } // end while (1)
}
```

Generates divide-by-zero (Math Error) trap

(b) Console Output

```
Reset cause: Power-on.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003001 (A2)
Fast RC Osc with PLL

trap_test.c, built on Jun  6 2008 at 10:17:57
Hit a key to start divide by zero test...OK. Now dividing by zero.
Reset cause: Software Reset.
Error trapped: Unhandled interrupt, Priority: 0x0B, Vector number: 0x04
```

pressed a key

`_DefaultInterrupt()` ISR saves error message and interrupt information from INTTREG, then causes the software reset.

`_printResetCause()` then prints out the saved error message, interrupt information.

A User-provided ISR

(a) In C

```c
void _ISR _MathError (void)
{
 // only action is
 // to clear the error
  _MATHERR = 0;
  RCOUNT = 0;
}
```

In Assembly (C30)

```
_MathError:
  push PSVPAG          ;save
  push W8              ;save
  mov.b #0,W8
  mov W8, PSVPAG       ;PSVPAG = page 0
  pop W8               ;restore
  ;clr _MATHERR flag
  bclr.b INTCON1,#4    ;_MATHERR = 0
  clr RCOUNT           ;RCOUNT = 0
  pop PSVPAG           ;restore
  retfie      ;return from interrupt
```

In *include\pic24_util.h*

```c
#define _ISR __attribute__((interrupt)) __attribute__ ((auto_psv))
```

(b) In C

```c
void _ISRFAST _MathError (void)
{
 // only action is
 // to clear the error
  _MATHERR = 0;
  RCOUNT = 0;
}
```

In Assembly (C30) The `no_auto_psv` causes PSVPAG to not be saved, reducing ISR size.

```
_MathError:
  ;clr _MATHERR flag
  bclr.b INTCON1,#4  ;_MATHERR = 0
  clr RCOUNT         ;RCOUNT = 0
  retfie      ;return from interrupt
```

In *include\pic24_util.h*

```c
#define _ISRFAST __attribute__((interrupt)) __attribute__ ((no_auto_psv))
```

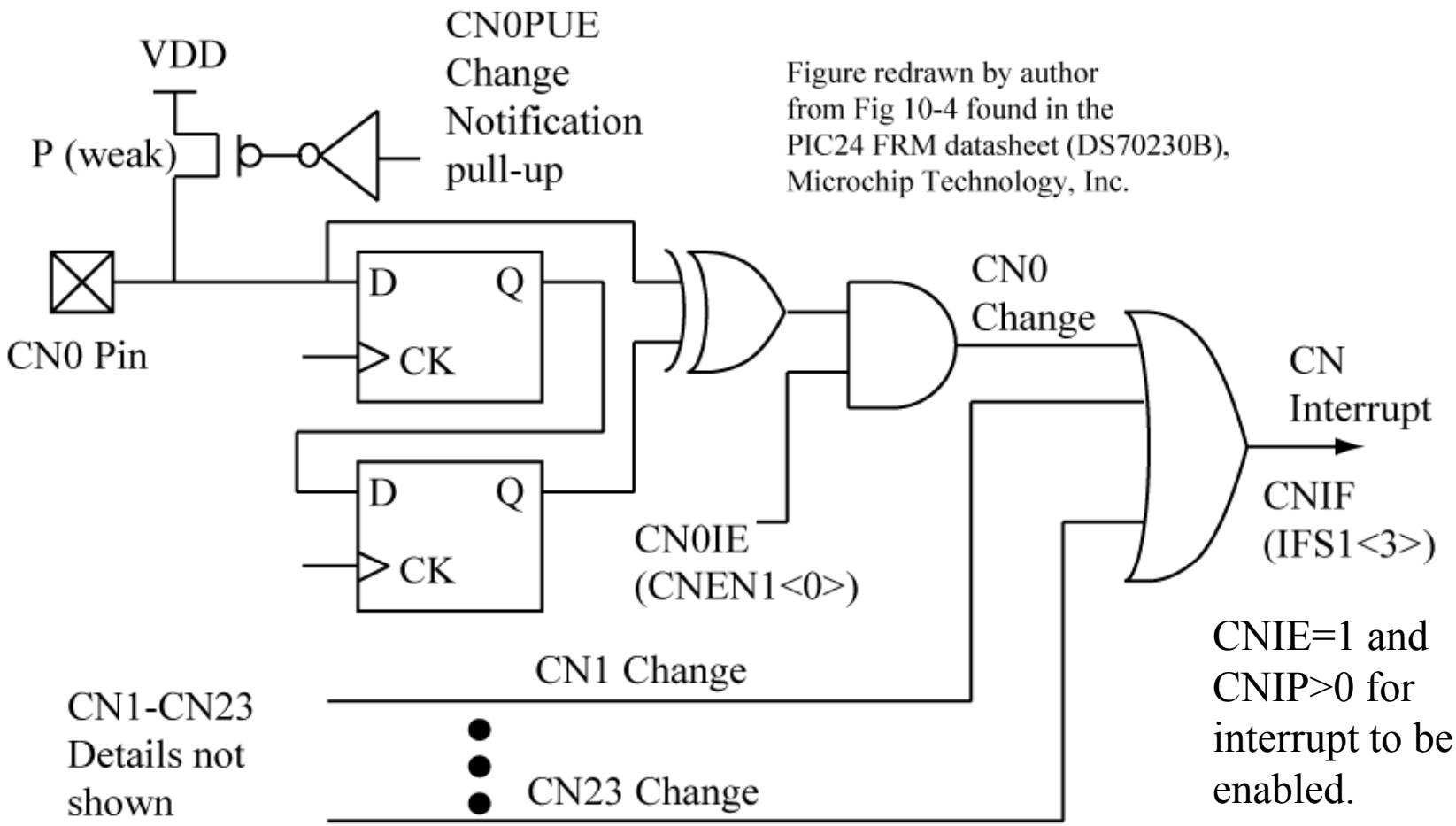
(c) MPLAB Program Memory

Line	Address	Opcode	Label	Dis
1	0000	040C02		goto _reset
2	0002	000000		nop
3	0004	000D8C		_DefaultInterrupt
4	0006	000D8C		_DefaultInterrupt
5	0008	000D8C		_DefaultInterrupt
6	000A	000D8C		_DefaultInterrupt
7	000C	00109A		_MathError

Math Error Trap vector now contains address of `_MathError` ISR.

These ISRs just clear the _MATHERR interrupt flag and return. If the interrupt flag is not cleared, get stuck in an infinite interrupt loop.

Change Notification Interrupts



When enabled, triggers an interrupt when a change occurs on a pin.

Use Change Notification to wake from Sleep

```
//Interrupt Service Routine for Change Notification
void _ISRFAST _CNInterrupt (void) {
  _CNIF = 0;    //clear the change notification interrupt bit
}
```

Clear the interrupt flag before exiting!

```
/// Switch1 configuration
inline void CONFIG_SW1()  {
  CONFIG_RB13_AS_DIG_INPUT();     //use RB13 for switch input
  ENABLE_RB13_PULLUP();           //enable the pull-up
  ENABLE_RB13_CN_INTERRUPT();     //CN13IE = 1
  DELAY_US(1);                    // Wait for pull-up
}
```

Macro to set CN*x*IE bit associated with RB13 port.

```
int main (void) {
  configBasic(HELLO_MSG);
  /** Configure the switch ***********/
  CONFIG_SW1();  //enables individual CN interrupt also
  /** Configure Change Notification general interrupt  */
  _CNIF = 0;        //Clear the interrupt flag
  _CNIP = 2;        //Choose a priority
  _CNIE = 1;        //enable the Change Notification general interrupt
  while(1) {
   outString("Entering Sleep mode, press button to wake.\n");
    // Finish sending characters before sleeping
    WAIT_UNTIL_TRANSMIT_COMPLETE_UART1();
    SLEEP();       //macro for asm("pwrsav #0")
  }
}
```

Pushing the switch here generates CN interrupt, causing wakeup and execution of the `_CNinterrupt` ISR, which then returns here and loop continues.

An interrupt flag (`_CNIF`) should be cleared before the interrupt is enabled (`_CNIE=1`). The priority (`_CNIP = 2`) chosen here was arbitrary, but it must be greater than 0 for the ISR to be executed.

Remappable Pins

Some inputs/outputs for internal modules must be mapped to RP*x* pins (remappable pins) if they are to be used.

Input Name	**Function Name**	**Example Assignment mapping inputs to RP*n***
External Interrupt 1	INT1	_INT1R = *n*;
External Interrupt 2	INT2	_INT2R = *n*;
Timer2 Ext. Clock	T2CK	_T2CKR = *n*;
Timer3 Ext. Clock	T3CK	_T3CKR = *n*;
Input Capture 1	IC1	_IC1R = *n*;
Input Capture 2	IC2	_IC2R = *n*;
UART1 Receive	U1RX	_U1RXR = *n*;
UART1 Clr To Send	U1CTS	_U1CTSR = *n*;
SPI1 Data Input	SDI1	_SDI1R = *n*;
SPI1 Clock Input	SCK1	_SCK1R = *n*;
SPI1 Slave Sel. Input	SS1	_SS1R = *n*;

Remappable Pins (cont.)

Output Name	**Function Name**	**RP*n*R<4:0> Value**	**Example Assignment**
Default Port Pin	NULL	0	_RP*n*R = 0;
UART1 Transmit	U1TX	3	_RP*n*R = 3;
UART1 Rdy. To Send	U1RTS	4	_RP*n*R = 4;
SPI1 Data Output	SDO1	7	_RP*n*R = 7;
SPI1 Clock Output	SCK1OUT	8	_RP*n*R = 8;
SPI1 Slave Sel. Out.	SS1OUT	9	_RP*n*R = 9;
Output Compare 1	OC1	18	_RP*n*R = 18;
Output Compare 2	OC2	19	_RP*n*R = 19;

Mapping outputs to RPx pins.

Remapping Macros

Contained in pic24_ports.h:

```
CONFIG_U1RX_TO_RP(pin)
CONFIG_U1TX_TO_RP(pin)
```

etc..

Example Usage:

```
CONFIG_U1RX_TO_RP(10);  //UART1 RX to RP10
CONFIG_U1TX_TO_RP(11);  //UART1 TX to RP11
```

INT2, INT1, INT0 Interrupts

These are input interrupt sources (INT*x*) that can be configured to be rising edge triggered or falling-edge triggered by using an associated INT*x*EP bit ('1' is falling edge, '0' is rising edge').

On the PIC24HJ32GP202, INT1 and INT2 must be brought out to remappable pins (RPx); INT0 is assigned a fixed pin location.

```
//Interrupt Service Routine for INT1
void _ISRFAST _INT1Interrupt (void) {
  _INT1IF = 0;    //clear the interrupt bit
}
/// Switch1 configuration, use RB13
inline void CONFIG_SW1()  {
  CONFIG_RB13_AS_DIG_INPUT();   //use RB13 for switch input
  ENABLE_RB13_PULLUP();         //enable the pullup
  DELAY_US(1);                  // Wait for pull-up
}
int main (void) {
  configBasic(HELLO_MSG);
  /** Configure the switch ***********/
  CONFIG_SW1();
  CONFIG_INT1_TO_RP(13);   //map INT1 to RP13
  /** Configure INT1 interrupt  */
  _INT1IF = 0;   //Clear the interrupt flag
  _INT1IP = 2;   //Choose a priority
  _INT1EP = 1;   //negative edge triggerred
  _INT1IE = 1;   //enable INT1 interrupt
  while(1) {
   outString("Entering Sleep mode, press button to wake.\n");
   //finish sending characters before sleeping
   WAIT_UNTIL_TRANSMIT_COMPLETE_UART1();
   SLEEP();       //macro for asm("pwrsav #0")
  }
}
```

Use INT1 to wake from Sleep mode

Hardware Timers

Recall that a Timer is just a counter. Time can be converted from elapsed Timer Ticks (*Ticks*) by multiplying by the clock period (*Ttmr*) of the timer:

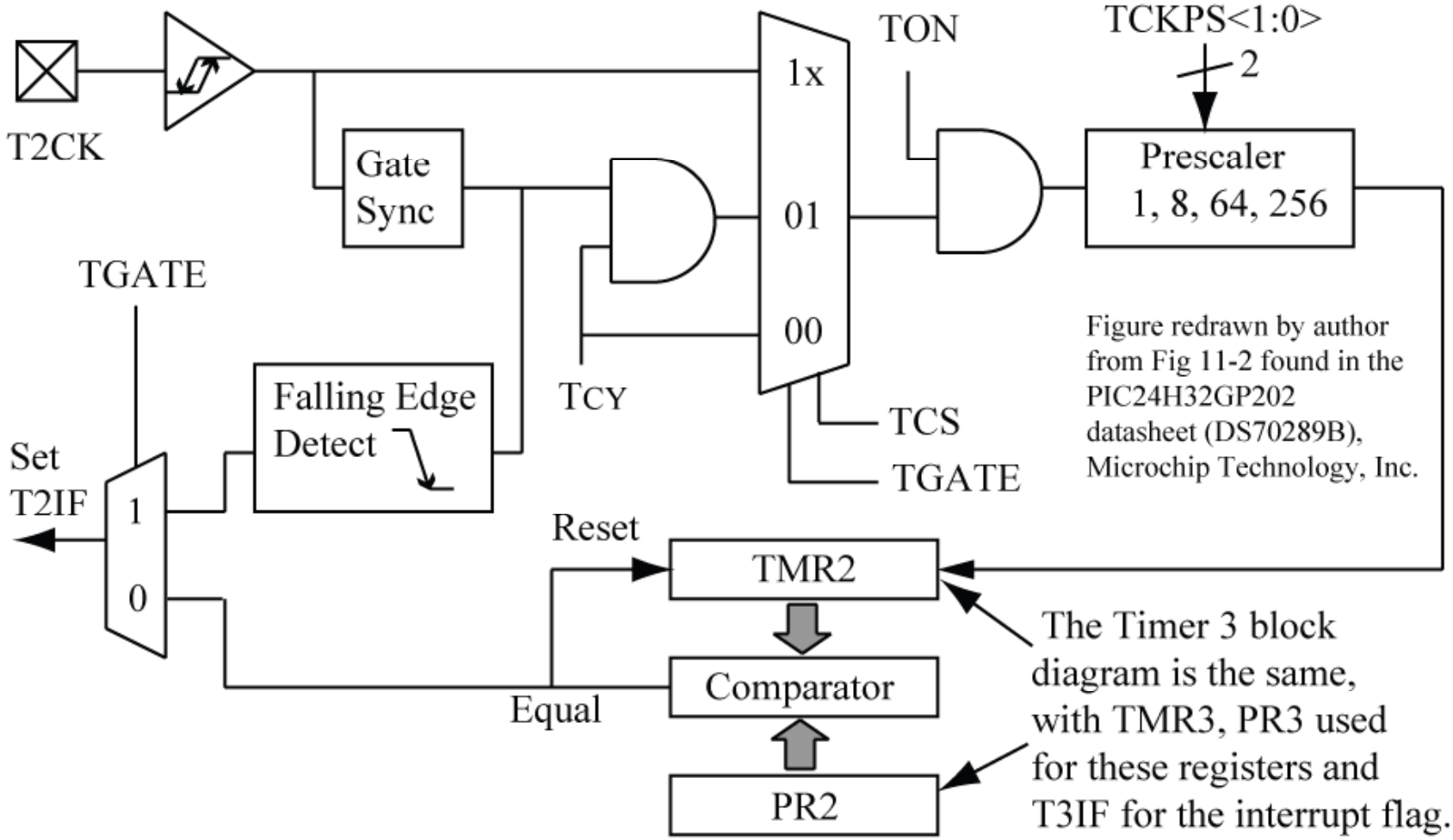
Time = Ticks x Ttmr

If a timer is a 16-bit timer, and it is clocked at the FCY = 40 MHz, then will count from 0x0000 to 0xFFFF (65536 ticks) in:

Time = 65536 x (1/40 MHz)

= 65536 x 25 ns = 1638400 ns = 1638.4 us = 1.6384 ms

Timer 2 Block Diagram



T2IF Period

The T2IF flag is set at the following period (T_{t2if}):

T_{t2if} = (PR2+1) x PRE x Tcy = (PR2+1) x PRE/Fcy

Observe that because Timer2 is a 16-bit timer, if PR2 is its maximum value of 0xFFFF (65535), and the prescaler is ‘1’, this is just:

T_{t2if} = 65536 x 1/Fcy

We typically want to solve for Tt2if, given a PRE value:

PR2 = (T_{t2if} x Fcy /PRE) − 1

Example T2IF Periods

PR2/PRE Values for T_{t2if} = 15 ms, Fcy = 40 MHz

	PRE=1	**PRE=8**	**PRE=64**	**PRE=256**
PR2	600000	75000	9375	2344
	(invalid)	(invalid)		

The PR2 for PRE=1, PRE=8 are invalid because they are greater than 65535 (PR2 is a 16-bit register).

Configuring Timer2 to interrupt every T_{t2if} period is called a **PERIODIC INTERRUPT**.

Timer2 Control Register

R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
TON	UI	TSIDL	UI	UI	UI	UI	UI
15	14	13	12	11	10	9	8
U-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	U-0
UI	TGATE	TCKPS<1:0>		T32	UI	TCS	UI
7	6	5	4	3	2	1	0

Bit 15: TON: Timer2 On Bit

When T32 = 1:
1 = Starts 32-bit Timer2/3
0 = Stops 32-bit Timer2/3

When T32 = 0:
1 = Starts 16-bit Timer2
0 = Stops 16-bit Timer2

Bit 13: TSIDL: Stop in Idle Mode Bit
1 = Discontinue module operation device enters Idle mode
0 = Continue module operation in Idle mode

Bit 6: TGATE: Timer2 Gated Time Accumulation Enable

When TCS = 1:
This bit is ignored.

When TCS = 0:
1 = Gated time accumulation enabled
0 = Gated time accumulation disabled

Bit 5-4: TCKPS<1:0>: Timer2 Input Clock Prescale Select Bits
11 = 1:256 , 10 = 1:64, 01 = 1:8, 00 = 1:1

Bit 3: T32: 32-bit Timer Mode Select bit[1]
1 = Timer2 and Timer3 form a single 32-bit timer
0 = Timer2 and Timer3 act as two 16-bit timers

Bit 1: TCS: Timer2 Clock Source Select bit
1 = External clock from pin T2CK (on the rising edge)
0 = Internal clock (FCY)

Legend:
R = Readable bit
-n = Value at POR
U = Unimplemented bit, read as '0'
W = Writeable bit
'1' = bit is set
'0' = bit is cleared
'x' = bit is unknown

Figure redrawn by author from Reg 11-1 found in the PIC24H32GP202 datasheet (DS70289B), Microchip Technology, Inc.

Note 1: In 32-bit mode, T3CON bits do not affect 32-bit operation

include\pic24_timer.h excerpts:

```
/*T2CON: TIMER2 CONTROL REGISTER*/
#define T2_ON                 0x8000
#define T2_OFF                0x0000

#define T2_IDLE_STOP          0x2000
#define T2_IDLE_CON           0x0000

#define T2_GATE_ON            0x0040
#define T2_GATE_OFF           0x0000

#define T2_PS_1_1             0x0000
#define T2_PS_1_8             0x0010
#define T2_PS_1_64            0x0020
#define T2_PS_1_256           0x0030

#define T2_32BIT_MODE_ON      0x0008
#define T2_32BIT_MODE_OFF     0x0000

#define T2_SOURCE_EXT         0x0002
#define T2_SOURCE_INT         0x0000
```


From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

Programming the configuration register

Just write a 16-bit value to the Timer2 configuration register to configure Timer2:

```
T2CON =  0x0020;   //Timer off, Pre=64, Internal clock
```

More readable:

```
T2CON =  T2_OFF | T2_IDLE_CON | T2_GATE_OFF |
         T2_32BIT_MODE_OFF | T2_SOURCE_INT |
         T2_PS_1_64;
```

This is actually:

```
T2CON =  0x0000 | 0x0000 | 0x00000 |
         0x0000 | 0x0000 |
         0x0020;
```

Can also set individual bit fields:

```
T2CONbits.TON = 1;      //Set TON bit = 1, turn timer on
```

Square Wave Generation

```
#define WAVEOUT _LATB2            //state
inline void CONFIG_WAVEOUT()  {
  CONFIG_RB2_AS_DIG_OUTPUT();     //use RB2 for output
}

//Interrupt Service Routine for Timer2
void _ISRFAST _T2Interrupt (void){
  WAVEOUT = !WAVEOUT;    //toggle output
  _T2IF = 0;             //clear the interrupt bit
}

#define ISR_PERIOD  15    // in ms
void  configTimer2(void) {
  //T2CON set like this for documentation purposes.
  //could be replaced by T2CON = 0x0020
  T2CON = T2_OFF | T2_IDLE_CON | T2_GATE_OFF
          | T2_32BIT_MODE_OFF
          | T2_SOURCE_INT
          | T2_PS_1_64 ;  //results in T2CON= 0x0020
  //subtract 1 from ticks value assigned to PR2 because period is PR2 + 1
  PR2 = msToU16Ticks (ISR_PERIOD, getTimerPrescale(T2CON)) - 1;
  TMR2  = 0;             //clear timer2 value
  _T2IF = 0;             //clear interrupt flag
  _T2IP = 1;             //choose a priority
  _T2IE = 1;             //enable the interrupt
  T2CONbits.TON = 1;     //turn on the timer
}

int main (void) {
  configBasic(HELLO_MSG);
  CONFIG_WAVEOUT();    //PIO Config
  configTimer2();      //TMR2 config
  //ISR does the work!
  while (1) {
    doHeartbeat(); //ensure that we are alive
  } // end while (1)
}
```

RB2 used for square wave output

On each interrupt, toggle the output pin to generate the squave wave, clear the interrupt flag.

Timer2 configuration sets T2CON, PR2; enables the Timer2 interrupt; turns on the timer.

The `msToU16Ticks()` value is decremented by 1 before PR2 assignment because timer period is PR2+1

After configuration, the ISR does the work of generating the square wave.

Timer2 configured to generate an interrupt every 15 ms. An output pin is toggled in the ISR, so square wave has period of 30 ms.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Switch Sampling

```
#define CONFIG_LED1() CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1   _LATB14      //led1 state

inline void CONFIG_SW1()  {
  CONFIG_RB13_AS_DIG_INPUT();  //use RB13 for switch input
  ENABLE_RB13_PULLUP();        //enable the pullup
  DELAY_US(1);                 // Wait for pullup
}

#define SW1_RAW          _RB13        //raw switch value
#define SW1              u8_valueSW1  //switch state
#define SW1_PRESSED()    SW1==0       //switch test
#define SW1_RELEASED()   SW1==1       //switch test

//debounced switch value that is set in the timer ISR
volatile uint8 u8_valueSW1 = 1;   //initially high

//Interrupt Service Routine for Timer3
void _ISRFAST _T3Interrupt (void) {
  u8_valueSW1 = SW1_RAW;     //sample the switch
  _T3IF = 0;                 //clear interrupt bit
}
//... other functions not shown ...
```

A Timer3 periodic Timer interrupt is used to sample the switch.

Switch state is now stored in a variable!

Timer 3 Configuration

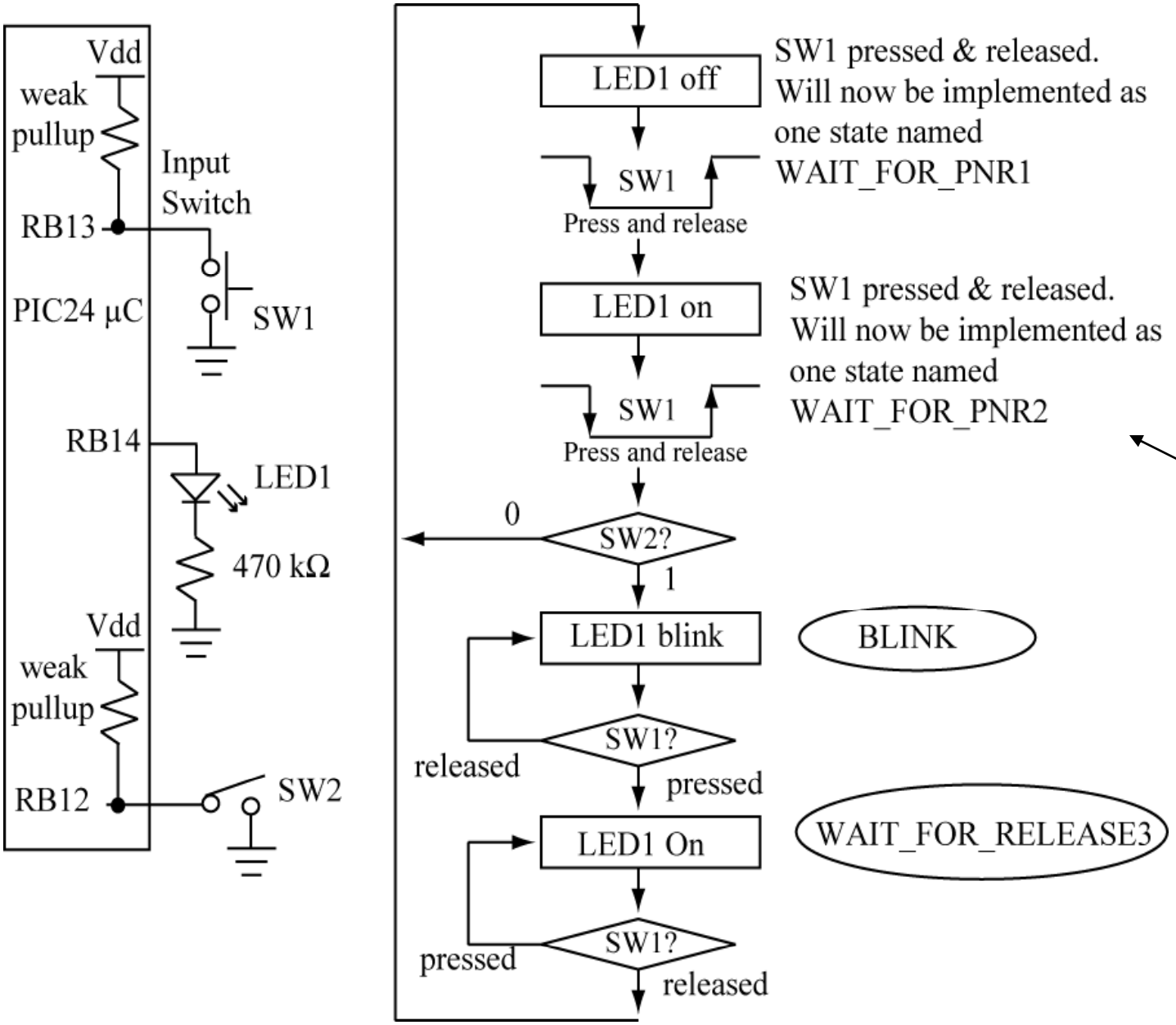
```
#define ISR_PERIOD     15                    // in ms
void  configTimer3(void) {
  //ensure that Timer2,3 configured as separate timers.
  T2CONbits.T32 = 0;      // 32-bit mode off
  //T3CON set like this for documentation purposes.
  //could be replaced by T3CON = 0x0020
  T3CON = T3_OFF |T3_IDLE_CON | T3_GATE_OFF
          | T3_SOURCE_INT
          | T3_PS_1_64 ;  //results in T3CON= 0x0020
  PR3 = msToU16Ticks (ISR_PERIOD, getTimerPrescale(T3CONbits)) - 1;
  TMR3  = 0;                          //clear timer3 value
  _T3IF = 0;                          //clear interrupt flag
  _T3IP = 1;                          //choose a priority
  _T3IE = 1;                          //enable the interrupt
  T3CONbits.TON = 1;                  //turn on the timer
}
```

Switch Sampling (cont.)

```
int main (void) {
  STATE e_mystate;
  //... config not shown ...
  e_mystate = STATE_WAIT_FOR_PRESS;
  while (1) {
    printNewState(e_mystate);
    switch (e_mystate) {
      case STATE_WAIT_FOR_PRESS:
        if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE;
        break;
      case STATE_WAIT_FOR_RELEASE:
        if (SW1_RELEASED()) {
          LED1 = !LED1;   //toggle LED
          e_mystate = STATE_WAIT_FOR_PRESS;
        }
        break;
     default:
        e_mystate = STATE_WAIT_FOR_PRESS;
    }//end switch(e_mystate)
    doHeartbeat();     //ensure that we are alive
  } // end while (1)
}
```

`DELAY_MS(DEBOUNCE_DLY)` removed from end of loop as the ISR periodically samples the input.

Semaphores



Will use a 'button press& release' semaphore to implement this as one state

From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

Press&Release Semaphore

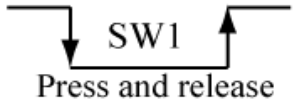
```
typedef enum  {
  STATE_WAIT_FOR_PRESS = 0,
  STATE_WAIT_FOR_RELEASE,
} ISRSTATE;

volatile uint8 u8_valueSW1 = 1;
volatile uint8 u8_pnrSW1 = 0;
ISRSTATE e_isrState = STATE_WAIT_FOR_PRESS;

//Interrupt Service Routine for Timer3
void _ISRFAST _T3Interrupt (void) {
  u8_valueSW1 = SW1_RAW;     //sample the switch
  switch(e_isrState) {
   case STATE_WAIT_FOR_PRESS:
      if (SW1_PRESSED() && (u8_pnrSW1 == 0))
          e_isrState = STATE_WAIT_FOR_RELEASE;
      break;
   case STATE_WAIT_FOR_RELEASE:
      if (SW1_RELEASED()){
         e_isrState = STATE_WAIT_FOR_PRESS;
         u8_pnrSW1 = 1; //set semaphore
         break;
      }
   default:  e_isrState = STATE_WAIT_FOR_RELEASE;
  }
  _T3IF = 0;  //clear the timer interrupt bit
}
```

ISR states

`u8_pnrSW1` semaphore set to 1 on press & release



Do not process another press & release until first has been handled.

ISR is now a state machine!

Set the `u8_pnrSW1` semaphore for foreground code.

A semaphore is a flag set by an ISR when an IO event occurs. The **`main()`** code is generally responsible for clearing the flag.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

```
typedef enum  {
  STATE_RESET = 0,  STATE_WAIT_FOR_PNR1,  STATE_WAIT_FOR_PNR2,
  STATE_BLINK,      STATE_WAIT_FOR_RELEASE3
} STATE;

int main (void) {
  STATE e_mystate;
  ... config not shown ...
  e_mystate = STATE_WAIT_FOR_PNR1;
  while (1) {
   printNewState(e_mystate);
   switch (e_mystate) {
    case STATE_WAIT_FOR_PNR1:
     LED1 = 0; //turn off the LED
     if (u8_pnrSW1) {
      u8_pnrSW1 = 0; //clear
      e_mystate = STATE_WAIT_FOR_PNR2;
     }
     break;
    case STATE_WAIT_FOR_PNR2:
     LED1 = 1; //turn on the LED
     if (u8_pnrSW1) {
      u8_pnrSW1 = 0; //clear semaphore
      if (SW2) e_mystate = STATE_BLINK;
        else e_mystate = STATE_WAIT_FOR_PNR1;
     }
     break;
    case STATE_BLINK:
     LED1 = !LED1; DELAY_MS(100); //blink if not pressed
     if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE3;
     break;
    case STATE_WAIT_FOR_RELEASE3:
     LED1 = 1;   //Freeze LED1 at 1
     if (u8_pnrSW1) {
      u8_pnrSW1 = 0;
      e_mystate = STATE_WAIT_FOR_PNR1;
     }
     break;
    default:
     e_mystate = STATE_WAIT_FOR_PNR1;
   }//end switch(e_mystate)
   doHeartbeat();     //ensure that we are alive
  } // end while (1)
}
```

main states

Replaces states WAIT_FOR_PRESS1, WAIT_FOR_RELEASE1 of original code

Test press & release semaphore

Clear the semaphore indicating that this press & release has been consumed.

Replaces states WAIT_FOR_PRESS2, WAIT_FOR_RELEASE2 of original code

Test press & release semaphore instead of SW1_RELEASED() because the semaphore is set on release and must be cleared.

main() code

Differences:

Only one state used for each press and release.

Use the **`u8_pnrSW1`** semaphore to determine when press/release occurred.

Another Solution

```
volatile uint8 u8_valueSW1 = 1;
volatile uint8 doBlink = 0;
STATE e_mystate;
//Interrupt Service Routine for Timer3
void _ISRFAST _T3Interrupt (void) {
 u8_valueSW1 = SW1_RAW;     //sample the switch
 switch (e_mystate) {
  case STATE_WAIT_FOR_PRESS1: ...
  case STATE_WAIT_FOR_RELEASE1: ...
  case STATE_WAIT_FOR_PRESS2: ...
  case STATE_WAIT_FOR_RELEASE2: ...
  case STATE_BLINK:
    doBlink = 1;
    if (SW1_PRESSED()) {
     doBlink = 0;
     e_mystate = STATE_WAIT_FOR_RELEASE3;
    }
    break;
  case STATE_WAIT_FOR_RELEASE3: ...
  default:
    e_mystate = STATE_WAIT_FOR_PRESS1;
  }
  _T3IF = 0; //clear the timer interrupt bit
}

int main (void) {
  ... config not shown ...
  e_mystate = STATE_WAIT_FOR_PRESS1;
  /* While loop just checks the doBlink semaphore */
  while (1) {
    printNewState(e_mystate);  //debug message when state changes
    if (doBlink) {
      LED1 = !LED1;
      delayMs(100);
    }
    doHeartbeat();     //ensure that we are alive
  } // end while (1)
}
```

blink semaphore

Unchanged from Figure 8.30

Tells the `main()` code to blink the LED. Do NOT put a software delay here to blink the LED!!!!!

Tell the `main()` code to stop blinking the LED.

Unchanged from Figure 8.30

Blink the LED when the `doBlink` semaphore is set.

Put entire FSM into the ISR instead of using a press&release semaphore.

Now use a **`doBlink`** semaphore to tell the **`main()`** code when to blink the LED.

Do not Blink in ISR! This delays exit from ISR.

Dividing Work between the ISR and main()

There are usually multiple ways to divide work between the ISR and main().

The ‘right’ choice is the one that services the I/O event in a timely manner, and there can be more than right choice.

Golden Rules:

The ISR should do its work as fast as possible.

Do not put long software delays into an ISR.

An ISR should never wait for I/O, the I/O event should trigger the ISR or the ISR should just sample the input!

An ISR is never called as a subroutine.

What do you have to know?

- How interrupts behave on the PIC24 μC
- Interrupt Priorities, Enabling of Interrupts
- Traps vs. Interrupts
- Change notification Interrupts
- Timer2 operation
- Periodic Interrupt generation
- Switch sampling using periodic timer interrupts