Chapter 7 - Performance

PIC32 Clock and Optimizing the Memory Interface for Performance
“Running”

- To understand the real speed of execution, we need to look at 2 systems:
  - The clock system
  - The memory cache system
- This is best done on PIC32MX hardware.
  - So far, we have been measuring performance in time (clock cycles) and space (bytes required) using simulations
  - A source code file named `run.c` or `running.c` provides some insights into this
Clock System

- **FRC** – internal, high-speed, low-power, gives 8 MHz nominal clock rate
- **LPRC** – internal, low-speed, low-power, gives 32 KHz nominal clock rate
- **POSC** – external, high-speed, quartz-based, up to 20 MHz external crystals can be input via OSCI/OSCO pins; with 2 gains (XT, HS)
- **SOSC** – external, low-speed, low-power, with external crystals of 32,768 Hz
- **EC** – external, gives a square wave of any Hz
Performance vs. Power

- Doubling the clock speed will not double the power consumption. ($P_{\text{dyn}}$ vs. $P_{\text{stat}}$)
- PIC32 clocks can do:
  - Run-time switching between internal and external oscillator sources
  - Run-time control over clock dividers
  - Run-time control over PLL circuit
  - IDLE modes (CPU halts, but peripherals still work)
  - SLEEP mode (both CPU and peripherals halt)
  - Separate control of the peripheral clock (PBCLK) from the CPU, for slower peripherals (for power)
PIC32MX Clock Module
Primary Oscillator Clock Chain

- This is the POSC we discussed just a few slides ago. It is an external clock.
- On the PIC32 Starter Kit, an 8 MHz crystal is connected across the OSCI and OSCO pins.
- Since this is below 10 MHz, set the primary oscillator for XT (vs. HS) operating mode.
- We can use the phase lock loops (PLL) to multiply this input frequency.
- See the next figure that shows how a system clock of 72 MHz is done with the POSC chain.
- So what is PLL anyway? How do we use it?
Primary Oscillator Clock Chain

- PLLs are complex by design. But PIC32 PLL has a simplified user interface.

- "Rules" to follow when using PIC32 PLL
  - Input frequency must be $\leq 4$ MHz.
  - Allow time for the PLL to "lock in".
  - PIC32 PLL must use the OSCCON SFR to select the frequency multiplication factor (PLLMULT) and to verify proper locking (SLOCK).

- This means the 8 MHz "raw" input must be reduced to at least 4 MHz before going into the PLL.

- Look at OSCCON SFR and POSC chain.
## OSCCON SFR

### Register 10-1: OSCCON: Oscillator Control Register

<table>
<thead>
<tr>
<th>Field</th>
<th>U-0</th>
<th>U-0</th>
<th>R/W-x</th>
<th>R/W-x</th>
<th>R/W-x</th>
<th>R/W-x</th>
<th>R/W-x</th>
<th>R/W-x</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRMEN</td>
<td>SOSC</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>bit 31</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PLLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>bit 23</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PBD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>bit 15</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>COSC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>bit 7</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKLOCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Primary Oscillator Clock Chain

![Diagram of Primary Oscillator Clock Chain]

**Configuration Bits**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFCSFF0</td>
<td>7FFFF</td>
<td>ICE/ICD Comm Channel Select</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PLL Input Divider</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PLL Multiplier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System PLL Output Clock Divider</td>
</tr>
</tbody>
</table>
Can you identify the Input Divider from the PIC32MX clock block diagram?

The PLLMULT field (3 bits) allows us to use multiplication factors from 15X to 24X.

In this case, an 18X factor was used to obtain a 72 MHz output from the PLL.

The Output Divider can get us down to 1/256 of the 72 MHz produced from the PLL. That will be 280 KHz. Anything lower we should use the SOSC whose range is 32~100 KHz.
Primary Oscillator Clock Chain

- Of course we can also use the LPRC, the internal low-frequency clock (32 KHz).
- At that operating frequency, your system works at a frugal current of 200 µA.
- This is good for battery-based systems.
Peripheral Bus (PB) Clock

- PIC32 feeds a separate clock signal to all its peripherals (an old friend of ours).
- This is done by inputting the System Clock (output of the POSC Chain) into another divider that is 2-bit or 3-bit (Gen II?) wide.
- PIC32MX has a dedicated PB divider (can you identify it?), so the processor can keep on running at its maximum frequency that is much higher than all the peripherals.
- OSCCON’s PBDIV field is used for this.
Peripheral Bus (PB) Clock

- Typically for PIC32MX, we use 36 MHz for the PB, i.e. half of the processor’s speed.
- All the above features allow us to manage power consumption by controlling the clock frequency at run-time (with code).
- To control these settings initially at power-up, we use the Configuration Bits stored in the flash memory (another old friend!).
- As a user, YOU set these configuration bits in MPLAB, in Configure | Configuration Bits.
Peripheral Bus (PB) Clock

- Once you did that, the Oscillator Module reads and uses the Configuration Bits to initialize OSCCON.

- Pages 148~149 of the textbook show the recommended settings.

- The settings are saved in the .mcw file (the workspace file). These will be “programmed” into the device configuration bits each time you write and build new code for the device.

- There is PLLODIV, similar to PLLIDIV.
Configuration Bits in Code

- Since we have access to OSCCON and all its fields in code, why not the ability to access and set the configuration bits in code?
- PIC32MX uses `#pragma` to set these bits.
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

**NOTE:** All PIC32MX devices are currently released to operate at frequencies up to **80 MHz**. *FPLLMUL = MUL_20* provides the required multiplier. Try it?
Fast Fourier Transform Algorithm

// input vector
unsigned char inB[N_FFT];

// input complex vector
float xr[N_FFT];
float xi[N_FFT];

// Fast Fourier Transformation
void FFT(void)
{
    int m, k, i, j;
    float a, b, c, d, wwr, wwi, pr, pi;

    // FFT loop
    m = N_FFT/2;
    j = 0;
    while(m > 0)
    {
        /* log(N) cycle */
        k = 0;
        while(k < N_FFT)
        {
            for(i = 0; i < m; i++)
            {
                a = xr[i+k];       b = xi[i+k];
                c = xr[i+k+m];     d = xi[i+k+m];
                wwr = wr[i<<j];    wwi = wi[i<<j];
                pr = a-c;          pi = b-d;
                xr[i+k]   = a + c;
                xi[i+k]   = b + d;
                xr[i+k+m] = pr * wwr - pi * wwi;
                xi[i+k+m] = pr * wwi + pi * wwr;
            }
            // for i
            k += m<<1 ;
        } // while k
        m >>= 1;
        j++;
    } // while m
} // FFT
Windowing (to Smooth Out Input)

```c
// apply Hann window to input vector
void windowFFT(unsigned char *s)
{
    int i;
    float *xrp, *xip, *wwp;

    // apply window to input signal
    xrp= xr; xip= xi; wwp= ww;
    for(i=0; i<N_FFT; i++)
    {
        *xrp++ = (*s++ - 128) * (*wwp++);
        *xip++ = 0;
    }
} // windowFFT
```
Scaling Back Modulus of Output

```c
void powerScale(unsigned char *r) {
    int i, j;
    float t, max;
    float xrp, xip;

    // compute signal power (in place) and find maximum
    max = 0;
    for(i=0; i<N_FFT/2; i++) {
        j = rev[i];
        xrp = xr[j];
        xip = xi[j];
        t = xrp*xrp + xip*xip;
        xr[j]=t;
        if (t > max)
            max = t;
    }

    // bit reversal, scaling of output vector as unsigned char
    max = 255.0/max;
    for(i=0; i<N_FFT/2; i++) {
        t = xr[rev[i]] * max;
        *r++ = t;
    }
}
```

Di Jasio - Programming 32-bit Microcontrollers in C
Initialization of FFT Vectors

```c
void initFFT(void) {
    int i, m, t, k;
    float *wwp;

    for(i=0; i<N_FFT/2; i++) {
        // rotations
        wr[i] = cos(PI2N * i);
        wi[i] = sin(PI2N * i);
        // bit reversal
        t = i;
        m = 0;
        k = N_FFT - 1;
        while (k > 0) {
            m = (m << 1) + (t & 1);
            t = t >> 1;
            k = k >> 1;
        }
        rev[i] = m;
    } // for I

    // initialize Hanning window vector
    for(wwp = ww, i = 0; i < N_FFT; i++)
        *wwp++ = 0.5 - 0.5*cos(PI2N * i);
}
```

Di Jasio - Programming 32-bit Microcontrollers in C
“FFT.h” Header (Declarations)

/*
 **  FFT.h
 **
 **  power of two optimized algorithm
 */

#include <math.h>

#define N_FFT   256              // #samples must be power of 2
#define PI2N    2 * M_PI / N_FFT

extern unsigned char inB[];
extern volatile int inCount;

// preparation of the rotation vectors
void initFFT(void);

// input window
void windowFFT(unsigned char *source);

// fast Fourier transform
void FFT(void);

// compute power and scale output
void powerScale(unsigned char *dest);
The “Running” Project

```c
/*
 ** Run.c (This should be the correct file name)
 **
 */
#endif

#include <p32xxxx.h>
#include <plib.h>#include "fft.h"

main()
{
    int i, t;
    double f;

    // 1. initializations with Omitted Options

    // System config performance
    SYSTEMConfigPerformance(72000000L); // what happens?

    // configure PB frequency and the number of wait states
    //SYSTEMConfigWaitStatesAndPB(72000000L);

    // enable the cache for max performance
    //CheKseg0CacheOn();

    // enable instruction prefetch
    //cheConfigure(0, 0, 3, 3);
    //mCheConfigure(CHECON | 0x30);
```
Capturing Time

// disable RAM wait states
//mBMXDisableD WrightStat();

// init FFT vectors and constants
initFFT();

// the test sinusoid
for (i=0; i<N_FFT; i++)
{
    f = sin(2 * PI2N * i);
    inB[i] = 128 + (unsigned char) (120.0 * f);
} // for

// init 32-bit timer4/5
OpenTimer45(T4_ON | T4_SOURCE_INT, 0);
WritePeriod45(-1L);  // don't write Timer45

// clear the 32-bit timer count
WriteTimer45(0);

// 2. perform the FFT computation
windowFFT(inB);
FFT();
powerScale(inB);

// read the 32-bit timer value
t = ReadTimer45();  // t counts # 1/(36MHz) cycles
f = t/36E6;  // f gives running time in sec

// 3. infinite loop
while(1);  // set a breakpoint here
}
} // main

Di Jasio - Programming 32-bit Microcontrollers in C
The CHECON Register

<table>
<thead>
<tr>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
<td>U-0</td>
<td>U-0</td>
<td>U-0</td>
<td>U-0</td>
<td>U-0</td>
<td>R/W-0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CHECOH</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
<td>r-0</td>
<td>r-0</td>
<td>U-0</td>
<td>U-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DCSZ[1:0]</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>U-0</td>
<td>R/W-1</td>
<td>R/W-1</td>
<td>R/W-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**CHECONbits.PFMWS = 7;**  // set max number of wait states

**.PFMWS ranges from 0 to 7**  // which value gives best time?
Optimizing WaitStates

SYSTEMConfigWaitStatesAndPB((72000000L);  // Try (80000000L)?
// CHECONbits.PFMWS = 7; ← worst performance + safest memory operation?
// CHECONbits.PFMWS = 6; ← better performance + less safe memory ops?
...
// CHECONbits.PFMWS = 0; ← best performance + erratic Flash memory ops?
// # Wait States ≡ # of CPU cycles spent waiting for Flash memory
// How do you achieve the following “f-value”, i.e. running time?
// See also documentation on SYSTEMConfigPerformance();

![Image of Watch window with symbols t and f with values 1535635 and 0.0426565277777778 respectively]
Turning the Cache ON

SYSTEMConfigWaitStatesAndPB(72000000L); // safer than specific .PFMWS
// try (80000000L) Gen II PIC32MX?
// now add the following call to enable the PIC32 cache
// the cache ≡ small but fast RAM between core bus and memory bus
// 256-byte cache is FIFO, for both instructions and data
// improves performance for non-zero # of wait states
CheKseg0CacheOn(); // comes from pcache.h library
// enabling cache cuts the “f-value” in half.
// only Kseg0 text+data can be cached, not Kseg1, because cached
  contents in Kseg0 can be examined (i.e. with CP turned ON)!
Enabling Pre-Fetch

```c
SYSTEMConfigWaitStatesAndPB(72000000L);
Checkseg0CacheOn();

// Instruction pre-fetching uses cache as well, by reading a block of
// 4 instructions (i.e. total 16 bytes of instructions) into cache
// sequential execution ➞ next 3 memory fetches have zero wait states!
// branch execution ➞ discard pre-fetched & reload new instructions
// in general, zero wait states is highly desirable but more difficult
mCheckConfigure(CHECON | 0x30); // equivalent to CHECONbits.PREFEN = 3;
// instruction pre-fetch ➞ additional 20% reduction in running time
```
Remove RAM Wait States

// configure PB frequency and optimize number of wait states
SYSTEMConfigWaitStatesAndPB(72000000L); // try (80000000L)?

// enable cache for data accesses
CheKseg0CacheOn();

// enable instruction pre-fetch
mCheConfigure(CHECON | 0x30);

// disable RAM wait states \(\Rightarrow\) (last) 1% reduction in running time
mBMXDisableDRMWaitState(); // impact depends on your code!

SYSTEMConfigPerformance(72000000L); // one call does it all?
Some Observations

- We have used virtually every available HW feature to reduce overall running time.
  - Wait States (4X), Cache (further 2X), Pre-fetch (further 0.2X), and disabling Wait States (further 0.1X).
  - These amount to a total 10X improvement.
- Can more be done?
- Connection between Wait States and Cache
- Make the best use of spatial and temporal locality of cache for more improvement.
Set a breakpoint at the OpenTimer45() call after the inB[] initialization ➔ “2-Hz” input sinusoid

X-axis: sample count (modeling “time domain”)
Y-axis: sample magnitude

Change “Sample Count” from 256 to 128, since FFT output size is half of its input size
FFT (2-Hz Spectral Component)