Phantom Registers, Transient Bugs, Development on the Motorola Power PC 555

Contents

A Forewarning

The MPC 555 should be used for mobility. If your project can exist sitting still, or may be "tethered" to a fixed spot, use the MPC 823. Otherwise, you are invoking all the torment of the MPC555 upon your stationary project for no apparent reason. Also note that the MPC555 could be used if you need a small, power-efficient board (regardless of movement).

Introduction

Development on the MPC 555 is not simple. For the most part, your instructors and others you ask for help do not know how to use the MPC 555. Everything you do must come from one (or both) of two sources. These sources are written documentation like this guide, other student documentation, the MPC 555 User's Manual, and anything else you can find on the Internet, and trial and error. Essentially this means you are on your own. Use your wit, and read the sections you need many times.

The purpose of this guide is to help you get up, get started, and get finished. This guide will not tell you how to do your project, nor should it be considered the only "right" way to use the MPC555. It is simply written to get you over the first few hurdles, and document all the gnats, meagers, and imps that the MPC 555 has in its arsenal.

Getting Started

-As of Spring 2009 Codewarrior will not install on Vista-

To program and use the MPC 555 you require a special program: Codewarrior. Obtain a free trial version off the Freescale website (http://www.freescale.com/). To do this, search for the MPC 555, find a software link from there, and download a trial version. You may be required to register, and in doing so Matt Smith suggests not calling yourself a student in case you need questions answered quickly. Install the program and run it.

Upon running it you will find that if you click any button (including "Register") the program will immediately crash. To fix this navigate into your install directory. From there, go to the folder "bin." Here you will find IDE.exe, right click on this and select the "Compatibility" tab. From here, uncheck the box that says "Run in compatibility mode for:". Now Codewarrior will behave, somewhat.

Launch Codewarrior and you should see a splash screen telling you how long your license is valid for. If it reports "0 days", or the splash screen does not appear, you need a new license to run Codewarrior. To get one, install on a new computer and use Codewarrior on that machine. You might also be able to use "Copy and Paste," a new computer and a file called "license.dat" in your install directory to obtain a fresh license. This may not be the most legal/honest thing to do, hence it not being detailed here. Finally, you can request a longer (30 day) evaluation license from Freescale (Cliff Hanger documentation, Winter 2007).

Now it is time to set up your first project. Run Codewarrior and select File\New. From here, select the Project tab and create a new project using EPPC New Project Wizard. On the next screen select PowerPC 555 for processor and 5xx Chip for the board. Select C for the language and the P&E BDM from the available connections. Now your project is set up and ready to go.

Upon doing this you create two projects "Internal Flash" and "Internal SRAM." These two projects represent your code for both SRAM and Flash on the PPC 555. Use flash for a final project that is debugged and ready to run on its own. Use SRAM (and your BDM cable) to run using the stepping tools.

If you plan to use Flash make sure to browse the "Guide to Flash" by Dan Zhang. It will get you set up to use Flash. If you want to use interrupts in flash or SRAM, then read that section in this guide.

Using the Queued Analog to Digital Converter (QADC)

The QADC on the MPC 555 is quite versatile. In this section, we will describe how to use the QADC in software triggered continuous scan mode. Software triggered continuous scan mode implies that the MPC 555 will perform the conversion at a certain interval and place the final result into a result register. When the value of the conversion is needed, a simple read from the result register will suffice.

In order to successfully use the QADC, we must first set some conditions. In order to do so, we create an initialization function and write the following code:

```
#include "mpc555.h"
```

```
void init_QADC();

void init_QADC()

{

        QADC_A.QADC64MCR.B.SUPV = 0; //Turn off supervisor mode

        USIU.PDMCR.B.PRDS = 1; //disable internal pullup resister
```

Supervisor mode allows us to write to certain traditionally read-only registers. Turning off supervisor mode protects us from overwriting registers we don't mean to change. Disabling the internal pullup register keeps the MPC 555 from pulling input lines high. If the pullup resister was enabled, input lines are defaulted to a high value, and change when a low value is passed. Disabling the pullup resister means that the input lines are defaulted to low values, which is what we want.

Now, we must set up our QADC to perform conversions. First, we must set how we want our ADC to be multiplexed. Internal multiplexing provides plenty of channels, but if needed, the ADC can be externally multiplexed.

```
        QADC_A.QACR0.B.MUX = 0x0; //ADC is internally multiplexed
```

Next, we set the QADC's clock. The MPC 555 user manual suggests to use the system clock divided by 20 due to backwards compatibility issues.

```
        QADC_A.QACR0.B.PSH = 0xB; //Div system clock by 20

        QADC_A.QACR0.B.PSL = 0x7;
```

Now, we specify parameters for each input we are converting. We do this by writing to the Conversion Command Word (CCW). A CCW has three parameters that interest us, the pause bit (B.P), the amplifier bypass (B.BYP) bit, and the sampling rate (B.IST) bits.

```
        QADC_A.CCW[0].B.P = 0; //does not pause between conversion

        QADC_A.CCW[0].B.BYP = 0; //does not bypass amplifier

        QADC_A.CCW[0].B.IST = 0x11; //set to highest sampling time for
                                                        //accuracy
```

The code above performs conversions on whichever input line is routed to CCW[0]. If we wanted to perform conversions on another input line, we only need to change to code slightly.

```
QADC_A.CCW[1].B.P = 0; //does not pause between conversion

QADC_A.CCW[1].B.BYP = 0; //does not bypass amplifier

QADC_A.CCW[1].B.IST = 0x11; //set to highest sampling time for

                                        // accuracy
```

Now, we need to route each channel in a CCW to a specific AN pin.

```
QADC_A.CCW[0].B.CHAN = 0; //Set channel 0 to pin AN0

QADC_A.CCW[1].B.CHAN = 1; //Set channel 1 to pin AN1
```

The way the QADC works is that is will perform conversions on every CCW starting from CCW[0] until it reaches the last one and loop back to CCW[0]. In order to prevent it from trying to convert unnecessarily, we tell the QADC to loop early by indicating the end of the queue. To do that, we set the channel of the lowest number CCW not used to 63.

```
QADC_A.CCW[2].B.CHAN = 63; //End of queue code
```

Now we place the QADC in the appropriate operating mode. After setting this, the QADC will immediately begin converting. A full list of operating modes can be found in table 13-13 on page 13-38 of the User's Manuel. This is the last line of code needed to initialize the QADC, so we close the function.

```
QADC_A.QACR1.B.MQ1 = 0x11;
}
```

If we wish to generate an interrupt after the conversions are done, we can specify this is the QACR1 register by setting the CIE bit to one.

```
QADC_A.QACR1.B. CIE = 0x1;
```

Finally, when we want to read the result of a conversion, we simply read from result register.

```
volatile int result;
result = QADC_A.RJURR[0].B.RESULT;
```

Note the use of volatile int to store the value of result. This is needed because the compiler will assume that RJURR[0].B.RESULT did not change, and simply hold on to the old value of result. By declaring it a volatile int, the compiler will now assume that the value of RJURR[0].B.RESULT is not guaranteed to be the same, and will grab its value.

Using Pulse-Width Modulation (PWM)

Generating pulses of specific pulses on the MPC 555 can be done with a little bit of code. First, we must set some conditions. We create a function in order to do so.

```
#include "mpc555.h"



void init_pwm();
void init_pwm()

{
```

Now, we need to unlock the PLPRCR register before we can continue any further. In addition, we need to unlock the MF (multiplier field) and DIVF (divider field).

```
USIU.PLPRCRK.R = 0x55CCAA33; // Unlock the PLPRCR register

USIU.SCCR.B.MFPDL = 0; // Unlock MF and DIVF fields (Writable)
```

Next, we pick our clock source.

```
USIU.PLPRCR.B.CSRC = 0;              // Normal clock source
```

Next, we set the MPC 555 to normal power mode. If we set the processor to low power mode, we will not be able to use a fast clock.

```
USIU.PLPRCR.B.LPM = 0;                  // Normal power mode
```

We set the DIVF we unlocked earlier. For our uses, dividing the system clock's frequency by two suffices. We are aiming for a small resolution, so we want to divide the frequency as little as possible.

```
USIU.PLPRCR.B.DIVF = 0x0;       // Divide system clock by 2
                                        // (Minimum divide)
```

We are required to further divide the clock with the MCPSMSCR register, so we will divide as little as possible. In addition, we enable the clock so we can produce pulses.

```
MIOS1.MCPSMSCR.B.PSL = 0x2;
// Divide system clock by 2
        // (Minimum divide)
MIOS1.MCPSMSCR.B.PREN = 1;
// Clock enable
```

The code above divides the clock generally for all PWM generation. However, each PWM output line comes with its own set of parameters which we can utilize to further fine tune our output signal. The first of the parameters is the prescaler. We are using PWM output pin 19.

```
MIOS1.MPWMSM19SCR.B.CP = 0x38;
```

It is EXTREMELY important to note that this register is in 2's complement. Thus if you want to divide by a number, you must put its 2's complement in this register. Now, our clock is divided into hundredths of a millisecond, which is excellent resolution. This is obtained by dividing by 40MHz by 2 and then 200 (Two's complement of 0x38). Having a resolution this small especially helps when wanting to set a servo's position to a very specific degree. Now, we need to set the period of our pulse.

```
MIOS1.MPWMSM19PERR.R = 0x7D0;
```

The MPC 555 generates the period by multiplying the value in the PERR register by the resolution. In our case, our resolution is one hundredths of a millisecond, and our PERR register is set to 2000. So, our period is equal to .00001s * 2000 = 20ms. Imagine the period to be a count. The count is the number you put in this register. The count "counts" at the speed designated above.

Now, we set the pulse width by writing the PULR register.

```
MIOS1.MPWMSM19PULR.R = 0x96;
```

The pulse width is generated by multiplying the value of the PULR register with the pulse width. So, in this case, our pulse width is .00001s * 150 = 1.5ms. The pulse represents how many units of the above "count" the pulse is high.

Now we enable the pulse generator and close our initialization function.

```
MIOS1.MPWMSM19SCR.B.EN = 1;
```

}

Flash and Interrupts

*For instructions on how to replace the utility program on Flash, or how to flash the board see the Flash Guide written by Dan Zhang.

Setting up SRAM Interrupts

Interrupts on the MPC 555 are very similar in nature to the interrupts on the MPC 823. Those of you who looked over the Memory Map in the Axiom board's manual, have noticed that 0x500 falls in the dreaded land of flash. This poses a bit of a problem as the easiest way to debug interrupts is with breakpoints, reading live memory, and stepping through your code. All of these things only work when you use the BDM cable and run in the SRAM region of memory.

To be more precise, the address 0x500 lands in the "utilities" section of flash, but some of the utilities remap interrupts to 0x010000. This means that our external interrupts are available to us at 0x10500. However, we set up our project so that we need not be aware of this small inconsistency, but it is fun to note anyway.

The first step is to set up several new files. These files need to create the "redirection" infrastructure of the interrupts. It is easiest to start with the file that Codewarrior generates to handle interrupts: eppc_exception.asm. Navigate to ".org 0x500" (Line 155) this is where Codewarrior places external interrupts. Here, change the code to redirect up into the flash RAM section:

```
#########################################################################
#######
#
#    0x0500 External Interrupt
#
#########################################################################
#######
      .org  0x0500

      isr_prologue

      li          r3,0x0500
      lis         r4,0x3fa000@h    //redirect to SRAM
      ori         r4,r4,0x3fa000@l
      mtlr  r4
      blrl

      isr_epilogue
```

Here, the interrupt redirects to SRAM at 0x3fa000. In this way, you can write your interrupt at 0x3fa010, which is in SRAM, and thus, can be debugged like the rest of your SRAM project.

Notice also the .org 0x500. This line essentially says that this code should be placed at 0x500 off of the "origin." The origin is defined as 0x10000 by the linker so that it does not interfere with the utility programs loaded into flash in the first two blocks by axiom.

Next, create an assembly file (.asm) to hold your interrupt routine. Write code at exactly 0x3fa000. To do so you need only have the following tag above your interrupt handler:

```
.section .abs.0x3fa000 //Absolute positioning
//Your assembly code here (Perhaps a branch table…)
```

Now you have created a redirect to 0x3fa000 and written your handler at this location.

Next, look at the link order tab for both the SRAM and Flash projects. Here, the eppc_exception.asm should be included for the Flash project and CANNOT be included in the SRAM project. That is because SRAM and the BDM cable can't write to address 0x10000 and therefore will throw an error if you link this file into the SRAM project. Your new .asm file (the one that contains the .section .abs.0x3fa000 tag) CANNOT be in the Flash project. This is because the flash utilities can't write to SRAM.

The next step is to make sure that other code does not try and exist at the same location. The first step to reserving space is to set up your linker command file. The top section of the linker command file should look as follows:

```
MEMORY {
     ram : org = 0x003fa100 //Adjusted here
     rom : org = 0x00010000 //Desired ROM address(boot address for 555)
       }
```

There are two changes here. The first is to set ram to 0x3fa100. This is merely precautionary, as the linker command file is not used for the SRAM project. If you were to run in Flash, this would push the RAM location back by 0x100 (to avoid your ISR) but your ISR won't exist at 0x003fa100 unless you are running in SRAM.

Next, some project settings need to be adjusted (they perform the same function for the SRAM project). Navigate to the project settings for your SRAM project. Click on the Linker section and then select EPPC Linker tab. On the upper right, you should see "Segment Addresses." Make sure to check the code segment and select address 0x003fa100. This tells the linker of your SRAM project that it should place code at and after 0x003fa100. In this way, you have reserved the addresses 0x003fa000 -0x003fa100 for your interrupt handler.

Let's review what has happened so far. The project has:

- Files to redirect interrupts from Flash to SRAM
- Absolutely positioned interrupt handler
- Shifted SRAM code past interrupt handler

The final step is to tell Codewarrior not to catch exceptions, but to ignore them and let your code catch them. To do this, go under SRAM settings and scroll down to the Debugging section. Select EPPC exceptions. Here you will find a list of exceptions that Codewarrior can catch for you. If checked, the exception will be caught by Codewarrior and Codewarrior will "crash" your program for you if it detects one. Uncheck "0x02000000 External Interrupts" to allow your board to handle them.

Now it is time to upload the project to the board. This has to be done in two steps. First "make" the Flash project. Make SURE the eppc_exception.asm is included. Start up the on-board utilities and flash the internal_FLASH.mot file onto the board.

Next, compile and run the SRAM project with the BDM cord attached. Again, make sure that eppc_exception.asm is NOT included and your own .asm file IS included. If you have followed all the steps correctly, your project will run and the board will catch interrupts and send them up to SRAM for your breakpoints to handle.

Debugging and Interrupts

There are some important things to note if you plan to run interrupts in SRAM (or even if you want to run in Flash).

The first is that the prologue that Codewarrior provides already saves the SRR* registers. Therefore, even if you place breakpoints in your interrupt handlers, you can continue to

run your program afterward and it will not fail. Just don't "step" out of your SRAM code back into the Flash code. This will cause the process to tank, as breakpoints and stepping doesn't play well in Flash land.

The second thing to note is some helpful code for controlling interrupts. You only need to use assembly to handle the MSR. Here is the code to enable interrupts:

```
asm
{
 mfmsr r12     //Get MSR
 ori r12,r12,0x00008002@l  //Enable interrupts
 mtmsr r12                 //Replace MSR

 }
```

This ASM block can be embedded straight into your C code. Next, use the following few structs to control interrupts:

```
USIU.SIPEND.B.IRQ6 = 1;  //Clearing IRQ in SIPEND
USIU.SIEL.R   |= 0x2AA80000;  //Set 1,2,3,4,5,6 to edge triggered
USIU.SIMASK.R |= 0x2AA80000;  //Unmask the above IRQ's
```

Please note the '|=' "ors" into the register, and thus does not unset anything. Use an '=' to set the register verbosely. Also, note the struct handles ALL 32 bits of the register, so the last 4 digits of your numbers should always be 0000, as these bits are reserved.

You are now a master of interrupts.

Flash Back to Flash

So now that your project is all debugged and ready to go, you must be ready to go completely Flash. Therefore, you need to revert your code to not redirect interrupts up to Flash. Otherwise, Flash will catch an interrupt and redirect it to uninitialized memory and your project will die a painful death.

To do this, you need to modify the code in eppc_interrupts.asm. It should look something like the following.

```
#########################################################################
#######
#
#    0x0500 External Interrupt
#
#########################################################################
```

```
#######
      .org  0x0500

      isr_prologue

      li          r3,0x0500
      lis         r4,FunctionName@h    //Call FunctionName
      ori         r4,r4,FunctionName@l //To handle interrupt
      mtlr  r4
      blrl

      isr_epilogue
```

Here, instead of branching to your absolute address in SRAM, it simply links to the label that you have placed in the code. It is just like calling a function and could even be a proper C function.

If you plan to use the same assembly file as in SRAM to handle interrupts in Flash, you must make sure that the .section .abs.0x003fa000 tag is deleted. Observe the following:

```
.text
.global FunctionName
FunctionName:
```

Notice, this is in the .text section (code section). Make sure to have that and the global definition or it won't compile and link properly.

Next, repair your linker command file to:

```
MEMORY {
     ram : org = 0x003f9800
     rom : org = 0x00010000 // desired ROM address (boot address for
555)
}
```

This is how the command file should have been to make the Flash project work.

Finally, make sure that all of your files are included in "Internal Flash" project. To do this, make sure that all the files are present in the "link order" tab. This INCLUDES your .asm file or C file that contains the handler (FunctionName). Make your project and flash the .mot file. Now interrupts are contained in Flash and your project could be mobile.

General Woes of the PPC

Number one, pressing the "reset" button does not necessarily clear registers. Some registers (like PWM registers) do not reset on reset. Thus, you will maintain values

across a reset if you are not careful. ALWAYS remember to initialize registers you use OR pull the plug.

Number two, Codewarrior likes to die a thousand deaths on your time. If it is throwing random errors, power-cycle the board, and make sure to reset the board without the BDM cable plugged in.

Number three, always plug in the BDM when the power is off. If you don't, then Codewarrior may start reporting "Unknown Protocol error"s. To fix this, apply step "Number 2" several thousand times and the board may grand you mercy.

Number four, never forget your jumpers. The PROG jumper, if on, allows programming (BDM) and runs the Flash Utilities (Serial Cable), and if off, it runs your program in Flash at 0x010000. Also, the VPP jumper lets you program Flash.

Behold Number 5:

Never lose hope, the PPC 555 WILL bend to your will, if you work at it.