

Motorola PowerPC 555 and You

Introduction

The MPC 555 is a fun piece of hardware to play with, but it has a rather large learning curve. As a cautionary piece of advice I highly recommend that you stick with the MPC823 for your project unless you absolutely need the mobility that the MPC555 provides. With that being said, the purpose of the rest of this document is to make using the MPC555 as painless as possible. My goal is to provide documentation that would have significantly improved our quality of life if we had it prior to building our project.

General Help

The single most valuable piece of information that I can give to you for using CodeWarrior and the 555 is this: when in doubt pull the power cord out. The 555 and CodeWarrior occasionally conspire to thwart your efforts by displaying useless error messages when you try to run your project. I've found that the only way to get it back in line is to starve of power. You can try to be nice and reprimand it gently by pressing the reset button, but this often doesn't work. Instead you have to show it some tough love by removing the power cable and putting it back in.

While this seems to be a rather obvious thing to do in a situation where things aren't working the way they should, I assure you when you are working on trying to program the flash memory to get interrupts to run, it is easy to forget this and spend the next couple hours wondering why the heck the flash memory hates you so much.

Getting to Know the 555

The MPC555 is just like the 823 in that it has memory mapped registers that control things such as timers, interrupts, and I/O pins. The major difference here is two fold: you can't use an FPGA, and even though you have access to memory mapped registers, you don't have to interact with them directly if you don't want to.

To address the former, FPGA's use a lot of power to run. As such, they are about as much a use for a mobile platform (if you aren't using the 555 for mobility please see the Introduction) as a 40 inch plasma screen is for a laptop. While the 555 certainly could be made to work with the FPGA, it would defeat the purpose of using the 555 to begin with. So what good is the 555 if you can't program your own digital logic? It turns out that the processor already provides a lot of the functionality that you would want to program into an FPGA. The main functionalities that we use in our project that we had previously implemented on the FPGA attached to the 823 were: built in pulse width modulation, analog to digital conversion, and general purpose I/O pins. The 555 also allows you to analyze PWM signals (TPU) and interface with external buses among other things that I have no idea how to use. So you're on your own for that stuff.

While nothing is stopping you from using the memory mapped registers to control the functionality of the processor, there exists a slightly easier and more C / C++ friendly way of doing so. Our good friends at Motorola created a bunch of data structures (read C struct's) that sit on top of the appropriate memory locations that the registers have been mapped to. These structures have been conveniently crafted and padded to allow you to modify the correct register by changing the values of the member variables in these data structures. After writing all of this wonderful code for MPC555 users everywhere, the people at Motorola proceeded to hide it in the bowels of their website, in a few cryptic links that really should have been named something closely related to: "Holy crap this stuff is useful!!!one!" In order to prevent you from experiencing the wonderful experience that is The Documentation Scavenger Hunt, we have provided links to useful information including these files. I know, we're mean nasty horrible people.

MPC555 Tutorials

I'd like to take this time to throw out a disclaimer to the masses before we get into some actual code. All of the following documentation isn't necessarily the best or even the right way to do this stuff. It is merely what worked for us.

Using the General Purpose I/O

The general purpose I/O pins allow you to read and write a 1 or a 0 to pin groups SGPIOA, SGPIOC, and SGPIOD. This is done by specifying the chip mode for the processor, specifying which pins are to be used for input and which are to be used for output, and finally either reading or writing to the memory mapped register that represents the pin states.

The first thing that you have to do is set the SC field of the SIU Module Configuration Register (SIUMCR) to the appropriate value. This field determines the functionality of the pin groups SGPIOA and SGPIOD. If you wanted to, you could use these pin groups on the 555 as address (SGPIOA) and data (SGPIOD) buses to communicate with an external chip. This will prevent you from using these pin groups as general purpose I/O pins, however. Since we were not interfacing with an external device, we set the SC field to 0x3 which tells the processor to not use these pin groups as buses, but general purpose I/O instead. This is done in C with the following line of code:

```
#include "mpc555.h"
...
USIU.SIUMCR.B.SC = 0x3;           //single chip mode
```

Everything in the above line of code sets the SC field of the SIUMCR to "single chip mode" which means that we are using the 555 as a stand alone chip that will not be communicating with any other chip. The only thing that was slightly confusing to me when I first used this was the "B" that has been snuck between "SIUMCR" and "SC." In other code samples, you will see a similar thing where instead of the "B" there is an "R."

From what I can tell, the “B” stands for bit field and indicates that you are interacting with a subset of bits inside of a register and the “R” stands for register and indicates that you are interacting with an entire register. To see what other values of this field can provide you, see table 6-9 on page 6-21 of the MPC555 User’s Manual (aka White Book).

After you have chosen the appropriate value of the SC field for your project, next up is choosing which pins you want to be used for input and which you want to be use for output. This is done by setting the appropriate values of the SGPIO Control Register (SGPIOCR). When making this decision, you have a couple of choices depend on which pin group you’re talking about. For the majority of the I/O pins, you can only specify if the entire group is going to be an input or an output rather than each individual pin. For pin groups such as these, you have to set the Group Data Direction (GDDR0 – GDDR5) fields inside of the SGPIOCR. For the pin group SGPIOD[24:31] that we will be using inside of the tutorial, each pin’s I/O direction can be set on an individual basis using the SPGIO Data Direction for SGPIOD (SDDRD) field of the SGPIOCR. In this tutorial we are going to set pins SGPIOD[30:31] up as input pins and pins SGPIOD[24:29] as output pins. The code for this is as follows:

```
USIU.SGPIOCR.B.SDDRD = 0xFC;
```

To make a pin an input, you set its corresponding position in the register to a 0. Writing a 1 has the opposite effect and configures the pin for output. For more information about setting the input / output status of a pin see page 6-37 of the User Manual.

Now for the easy part. Once all of chip mode and pin uses have been set up all you have to do to manipulate the general purpose I/O pins is to read and write to a particular register depending on which pin group you are using. For our purposes we are going to write to bits 24-31 of the SGPIO Data Register 1 (SGPIOD1):

```
USIU.SGPIODT1.R ^= 0xFF;
```

The above line of code sets the current state of the SGPIOD[24:31] to the XOR of the bits’ previous state and 0xFF, effectively flipping state of the bits. This is a good time to note that since we set pins SGPIOD[30:31] to be input pins, the above line of code will not have any effect on them. All inputs will ignore any writes to the SGPIODT1 register and maintain whatever state they are being driven to. Reading from the pins is equally easy:

```
int data = USIU.SGPIODT1.R;
```

After this line of code executes, data will contain the states of *all* of the pins including the current state of the output pins.

Using the Queued Analog to Digital Converter

The Queued Analog to Digital Converter (QADC) has many modes of operation but in this tutorial we will only be covering one of them: Software Triggered Continuous Scan Mode. So what exactly does this mean? Putting the QADC into software triggered continuous scan mode allows you to interact with the QADC in a very straightforward way. When you want a value from the ADC, you just read it from a register some place. The QADC does all of the other work for you behind the scenes. Gone are the worries about handling interrupts or polling until the data is ready. Instead what happens is the QADC is constantly performing ADC on the specified pins. Each time it finishes performing the conversion, it writes the result to a results register and then moves on to the next conversion. If you have a situation where you have many signals that need to be converted and each signal has a different priority, using this mode probably isn't a very good choice, but for most everything else software triggered continuous scan is a dream come true.

Before we can get the party started with software triggered continuous scan we need to write some code to get things setup:

```
QADC_A.QADC64MCR.B.SUPV = 0; //Turn off supervisor mode
USIU.PDMCR.B.PRDS = 1;      //disable internal pullup resister
```

The first thing that we do in the above code is to allow ourselves full access to all of the QADC registers regardless of whether we are running in supervisor mode or not. The second thing that we do is disable the internal pull up resister. Now I'm just a lowly CS major and I don't have a clue as to what a pull up resister is. I was just told by people who know more about that stuff than I do that this was a good idea. If you have some compelling reason to not use this particular line of code, I won't stop you, but just remember this: since I have absolutely no idea what the consequences of this line of code are, not using it could very well cause the earth to explode. Consider yourself forewarned.

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

QADC_A.QACR0.B.PSH = 0xB;      //Div system clock by 20
QADC_A.QACR0.B.PSL = 0x7;
```

Next up, we tell the QADC that the converter should be internally multiplexed. You can perform your own external multiplexing if you need more channels than internal multiplexing can provide. After that we set the QADC to use the system clock divided by 20. The User's Manual doesn't seem to give any clear cut reason why you would want to do this, other than a brief note about backwards compatibility. The reason why I did this in our project was merely because it was done in example in the User's Manual. For more information about the Prescaler High (PSH) and the Prescaler Low (PSL) see page 13-26 in the User's Manual.

As a note, if you need an interrupt to be thrown whenever a scan queue has finished converting, you can specify this behavior in the CIE1 and PIE1 fields of the QACR1 register and the IRLx field of the QADC64INT register.

Now that we have setup all of the general QADC stuff we need to configure each Conversion Command Word (CCW) that we plan on using. A CCW is a register in the QADC's RAM that can be used to request conversions from a particular pin and represents a single ADC in the queue.

```
QADC_A.CCW[x].B.P    = 0;      //does not pause between CCW
QADC_A.CCW[x].B.BYP  = 0;      //does not bypass amplifier
QADC_A.CCW[x].B.IST  = 0x11;   //set to highest sampling time for
                                // accuracy
```

In the first line of code we are telling each CCW not to enter the pause state after the conversion for this CCW has been performed. The next line of codes tells the QADC not to bypass amplifying this CCW's signal. Doing this affects the timing of the QADC, but not in any way enough to be a problem for our project. Then, we tell the CCW to use the highest sampling time for accuracy.

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

Finally we set the channel that the CCW uses. The channel represents the pin that the CCW uses to read the voltage from in the conversion. In the above example, we are setting CCW0 to channel 0 which results in it converting the signal from pin AN0. Table 13-20 on page 13-48 shows all of the values that CHAN can be set to and the pin assignment that results.

Once all this has been done for each CCW you plan on using, you need to configure the CCW immediately after that last CCW that you are using. As an example, say you are trying to perform an ADC on 5 different CCWs, then the following CCW (5 in this case) has to be set this way:

```
QADC_A.CCW[5].B.CHAN = 63;     //End of queue code
```

The number 63 denotes the end of queue code and tells the QADC that this is the last CCW you are using. When the QADC reaches a CCW whose CHAN has been set to 63, it will wrap back around to CCW[0] and begin converting from there again. The very last thing that we do is set the MQ1 field of the QACR1 to the appropriate operating mode:

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

After this has been set, the QADC immediately begins converting. A list of all the operating modes can be found in table 13-13 on page 13-38 of the User's Manual. Please note that all of the values in the table are in binary not hex as the above code is.

Alright, getting the QADC setup is all well and good, but how do we actually *access* the results of the conversion? For that you use the Right Justified Unsigned Result Register (RJURR). The following line of code gives you the result of CCW[0].

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

Using the built in PWM module

The built in Pulse-Width-Modulation module on the 555 can be very useful. It is a good alternative to using timers and generating your own PWM signal. The trouble with using the PWM lies in the setup. Getting the period you want for your signal will likely be the hardest part of your task. However, there are several prescalers at your disposal, and now we'll describe where to find them.

```
USIU.PLPRCRK.R = 0x55CCAA33;    // Unlock the PLPRCR register
USIU.SCCR.B.MFPDL = 0;          // Unlock MF and DIVF fields
```

The above code is going to do exactly what the comments say. This allows you to change the system clock by making it possible to multiply and divide by some prescaler.

```
USIU.PLPRCR.B.MF = 0;           // Multiply input clock by 1
USIU.PLPRCR.B.CSRC = 0;         // Normal clock source
USIU.PLPRCR.B.LPM = 0;          // Normal power mode (not low power)
USIU.PLPRCR.B.DIVF = 0x1;       // Divide system clock by 2
```

The code above is going to do all of one thing, and that is half the frequency of the system clock. If needed, you could multiply the clock by using a value other than 0 for the MF field. Or you could further divide the clock by changing the DIVF field. The LPM should be zero for the normal power operation; the other values are for low power modes. The CSRC is going to set the source of the system clock – I'm not sure what changing this value does, so I wouldn't recommend changing it.

At this point we've changed the system clock, but that doesn't help us much for fine-tuning our desired PWM period. At this point, you need to make a choice as to what PWM pin you're going to use. After looking over several previous examples of how people have implemented PWM stuff, we're deciding that PWM19 is a good place to start. Note that each PWM pin has its own period and prescaler settings.

```
MIOS1.MPWMSM19PERR.R = 0x5f;   // Set period
MIOS1.MPWMSM19SCR.B.CP = 0x7c;  // Set prescaler
MIOS1.MPWMSM19SCR.B.EN = 1;     // Enable prescaler
```

We never figured out what settings for the period correspond to, so we have listed our method as follows: try out different values of PERR and the prescaler and observe the

waveform generated on an oscilloscope. It's a guess-and-test methodology, but it works, and it doesn't take very long.

Finally, you're probably going to want to set the pulse width of your signal. There is one register that you write to and the mpc555 will take care of the dirty work for you.

```
MIOS1.MPWMSM19PULR.R = 0x1;      // Set pulse width to ~1%
```

This seems pretty simple, and it is. The only problem we found was that the type of servo we used required a .7% pulse width which we had to approximate with the above code. It was a little shaky, but it worked. Whatever value you write to this register - that's going to be the % pulse width. We were mostly concerned with very small pulse widths, and writing a 1 to this register, as stated before, produced approximately a 1% PWM signal and writing a 2 produced approximately a 2% PWM signal. At this point you can feel free to experiment with various periods and prescalers until you get the signal you need. Don't forget to look on the pin-mapping document to find out where the PWM pin you're using is. I believe it's called PB555_SCH_revC.pdf.

As a side note, you even further fine tune your desired period/frequency by modifying the MIOS Counter Prescaler Submodule Register. It can be done as follows:

```
MIOS1.MCPSMSCR.B.PSL = 0x0;      // Set prescaler to divide by 16
MIOS1.MCPSMSCR.B.PREN = 1;      // Enable prescaler
```

This code is going to change the value that is loaded into the counter that powers the built in PWM generation. Different divide ratios are listed in the user's manual on page 15-13. In all honesty, using this probably isn't necessary, but it's there if you need it.

The Dreaded Flash Memory

In order to use interrupts you must be able to write instructions at address 0x500. This memory space (0x0 – 0x0006FFFF) is all held in flash memory. The advantage to using flash is that you can use the nonvolatile nature of this memory to run your program immediately without connecting to the BDM. This was, however, a feat that we were never able to accomplish. Despite this setback we were able to load instructions into the flash memory which would have enabled us to use interrupts. This section is intended to help you do the same.

Even though CodeWarrior provides you with default “Internal Flash” configuration settings for your project along with the “Internal SRAM” setting, switching to the flash configuration and hitting run will not return the expected results. Instead, you have to use the “Flash Programmer” tool to first write your program into flash memory.

The Flash Programmer is accessed by clicking the Tools menu in CodeWarrior and selecting Flash Programmer. Rather than writing about the correct configuration that must be used to correctly program the 555's flash memory we have provided a settings file. This file can be loaded by pressing the “Load Settings...” button. Once the settings

have been loaded, click on the “Erase / Blank Check” list item in the listbox on the left. Every time you want to program the flash memory you *must* erase the flash sectors that you will be using. This is done by pressing the “Erase” button.

Next, to actually load your program into flash memory click the “Program / Verify” list item in the listbox of the left. Check the “Use Selected File Field” and browse to yourprojectname/bin/ internal_FLASH.elf and then select “Elf Format” from the “File Type” drop down box. Once this has been done click the “Program” button.

Unfortunately should any error messages appear when you try to program the flash, determining what went wrong is rather difficult because the error messages are usually extremely useless. The only thing that I can really offer for help is to make sure that you have erased the sectors prior to trying to program the flash.

Once you have successfully written your program into flash memory, you’re not done yet. You can try to directly run the Internal Flash version of your project, but in most cases this did not work for us and no amount of resetting and power cycling could help us. Instead, we had to switch to the Internal SRAM version of the project, run that, stop it, switch back to the Internal Flash, and then finally run the Internal Flash. After you have correctly performed all of the above incantations and arm waving, your project should be ready to run from the flash.

One thing to note about this whole process is that you only have to program the flash memory once. As long as all of the changes that you have made occur only in the SRAM, you do not have to reload the flash memory. Now if you’re trying to get the thing to run straight from internal flash, that’s a different story.

Running From the Flash

As I mentioned earlier, we were unable to get our project run directly from flash memory. Instead we had to hook up the BDM cable, press run, and then quickly remove the cable. We did, however, spend some time trying to get our project to run from flash. There are two things that I would like to share that I think are important in accomplishing this feat.

First off, make sure to remove the PROG_EN jumper on the 555 board once you have loaded everything into flash memory. When the PROG_EN jumper is removed, the 555 is supposed to run from flash memory (from what I can tell, I believe that it is specifically supposed to start running from address 0x100 –the reset interrupt). Just remember to put the jumper back on when you make any changes to your code and go to load the revisions into the 555.

Secondly, I believe that the “correct” way to run from flash memory is not to run from flash at all. Instead, you are supposed to write a boot loader that reads memory from flash, copies those instructions into SRAM, and then run the program from SRAM itself. The sample code files on the “MPC-5xx Development Boards” CD seem to suggest that this is the right way to do it. For all I know, however, it may be possible to execute instructions directly from flash memory. This especially seems to be the case because if it were not true, where would your boot loader run from?