Storing variables that will survive program updates

By Rick Mainhart

It seemed simple enough … I wanted to be able to store some processor-specific values in Flash EEPROM that would be read during startup, yet would survive scrubbing program memory and a subsequent program reload. I originally thought I'd have to work out a routine that saved all the data in case we upgraded ISOMAX, but this is no longer the case.

Since this was a minor issue (compared to getting my products out the door) I put it aside several times. Well, I finally cleaned the to-do list up, and am left with storing the processor ADC offset and gain values as well as some other specific system parameters.

A bit of background is in order. I'm using the ServoPod processor in an industrial application. I anticipate having upward of 1000 ServoPods on one factory floor. Having to maintain an individual program listing for each and ever processor is unwieldy at best, and is a constant source of error (load the wrong calibration values and you're in trouble). Further, I had wanted to give the customer the ability to handle their own calibration, but didn't want to give them my source code.

Quite simply put, I needed to store my configuration data into Data Flash, and ensure that it survived rebooting as well as a program memory SCRUB and reprogramming cycle. This data will survive a firmware upgrade with a simple addition to the command line text, as shown below:

Original command line to flash ISOMAX V082 into a ServoPod

```
flash_over_jtag flash807.cfg V082-7.S
```

To prevent overwriting the User Data Flash area, enter the command line text as shown below:

```
flash_over_jtag flash807.cfg V082-7.S –page
```

By adding the –page switch, we only erase the pages used by the S Record (in this case V082-7.S), and ONLY those pages. Our User Data Flash information remains unchanged.

The following was tested on an IsoPod V2 and a ServoPod with IsoMAX V0.82 firmware installed.

Hex versus Decimal

I don't think well in Hex, although I do see the advantages of Hex math. As such, I'll be taking care to identify the number base as we go along.

The User Data Flash memory is described in Isomax Memory Maps.pdf (available from the downloads section of the New Micros website), and shows the IsoPod User Data Flash memory starting at $1800 (6144 decimal), and the ServoPod User Flash memory starting at $3000 (12288 decimal). Each page (and this is important) is $100 (256 decimal) words long. When you perform an erase function, you erase a full page at a time … for example:

```
HEX
1800 EEERASE  (3000 EEERASE for the ServoPod)
```

will erase all memory in Data Flash from $1800-18FF ($3000-30FF for the ServoPod), by writing all ones ($FFFF or 65535 decimal) to each address location. You can verify this by the command DUMP as follows

```
HEX
1800 100 DUMP (3000 100 DUMP for the ServoPod)
```

This gives you a screen full of addresses with each containing $FFFF (65535 decimal). Note there are only 4 places displayed, no matter the number base chosen, so 65535 decimal is actually shown as 5535 decimal and FFFF Hex.
Now, if we want to write values to this erased memory, we'll use the EE! word:

```
HEX
1 1800 EE! (1 3000 EE! for the ServoPod)
```

this places the value $1 (1 decimal) at address $1800 or 6144 decimal ($3000 or 12288d for the ServoPod). Verify by using the DUMP command we discussed earlier:

```
HEX
1800 8 DUMP (3000 8 DUMP for the ServoPod)
```

You'll get 1 FFFF FFFF FFFF FFFF FFFF FFFF FFFF as the result.

Now cycle the processor power. Here's where I started making mistakes (and why I put the word HEX in front of all my commands).

```
HEX
1800 8 DUMP (3000 8 DUMP for the ServoPod)
```

again, you will get 1 FFFF FFFF FFFF FFFF FFFF FFFF FFFF
as your result.

Issue the command SCRUB. This clears PROGRAM memory, so the Data Flash is left alone. Verify with:

```
HEX
1800 8 DUMP (3000 8 DUMP for the ServoPod)
```

again, you will get 1 FFFF FFFF FFFF FFFF FFFF FFFF FFFF
as your result.

Now you can issue the erase command for this page again:

```
HEX
1800 EEERASE  (3000 EEERASE for the ServoPod)
```

When you run the same dump command

```
HEX
1800 8 DUMP (3000 8 DUMP for the ServoPod)
```

you will get FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
as your result.

That's it. Pretty simple. You can store any 16-bit value directly by writing it to the address with EE!, read it with @, erase it with EEERASE (but remember that you erase a whole page at a time), and it survives SCRUB and rebooting.

Letting IsoMax work for you

To make things much easier to use, what I'm going to work on next is letting ISOMAX perform all my memory management for me. Why? Simply put, not everything I want to store is a single precision value. I want double and floating point values stored as well.

Define all of the variables you want to be non-volatile in a group, such as:

```
VARIABLE  ADC0_OFFSET  EEWORD
VARIABLE  ADC1_OFFSET  EEWORD
VARIABLE  IP_ADDR_A    EEWORD
VARIABLE  IP_ADDR_B    EEWORD
VARIABLE  IP_ADDR_C    EEWORD
VARIABLE  IP_ADDR_D    EEWORD
2VARIABLE MAX_BUS      EEWORD
FVARIABLE ADC0_GAIN    EEWORD
FVARIABLE ADC1_GAIN    EEWORD
VARIABLE CALFLAG       EEWORD
```

Now you have a group of variables, the first six take one 16-bit address, the next three take two 16-bit address slots and the last takes one 16-bit address. I recommend the last variable be a single to ensure you grab all the addresses you need.

There currently is no word that will take a range of addresses and copy them from one memory location to another (while the EEMOVE command states that the destination address should be in Data Flash, I have not tested this), so we can either read each address directly and write it to its new location, or we can create a word to perform the same function as EEMOVE (addr1 addr2 u --) only allowing movement from high memory to low. The question of what starting address we need can be found by:

```
HEX
ADC0_OFFSET .
```

Note I didn't fetch the data at the address pointed to by the variable ADC0_OFFSET, but rather fetched the address itself, and printed it. To verify all the variables are in one block, fetch and print all the variables you need to restore, and note their addresses.

Using the above example, I get the following:

```
SCRUB OK
VARIABLE  ADC0_OFFSET  EEWORD  OK
VARIABLE  ADC1_OFFSET  EEWORD  OK
2VARIABLE MAX_BUS  EEWORD  OK
FVARIABLE ADC0_GAIN  EEWORD  OK
FVARIABLE ADC1_GAIN  EEWORD  OK
VARIABLE CALFLAG   EEWORD  OK

HEX  OK
ADC0_OFFSET . 2BE  OK
ADC1_OFFSET . 2BF  OK
MAX_BUS . 2C0  OK
ADC0_GAIN . 2C2  OK
ADC1_GAIN . 2C4  OK
CALFLAG . 2C6  OK
```

Note that MAX_BUS starts at $2C0 (704 decimal) and that the next variable ADC0_GAIN starts at $2C2 (706 decimal) and NOT $2C1 (705 decimal). If you finish up with a double or floating point variable, you MUST be sure to increment the address by one to be sure you capture both halves of the variable word. Here, we end up with a single variable (CALFLAG) and so the last address we need concern ourselves with is 2C6. These values are identical in both the IsoPod (V0.82) and ServoPod (V0.82).

If we used $1800 as our starting address, and wanted to copy 12 locations from Data Flash, we could simply do:

```
HEX
1800 @ 2BE !
1801 @ 2BF !
1802 @ 2C0 !
...
```

and so on. That's a lot of typing, and you need to keep track. You could create a loop index that increments addresses and performs a read and write function as shown below. Please note that I am using a ServoPod (V0.82) and am working in DECIMAL. From the earlier example:

```
SCRUB
IsoMax V0.82
VARIABLE  ADC0_OFFSET  EEWORD    OK
VARIABLE  ADC1_OFFSET  EEWORD    OK
2VARIABLE MAX_BUS        EEWORD    OK
FVARIABLE ADC0_GAIN      EEWORD    OK
FVARIABLE ADC1_GAIN     EEWORD    OK
VARIABLE CALFLAG           EEWORD    OK
  OK
```

Now, let’s make sure of our starting address:

```
ADC0_OFFSET . 702  OK
```

Now, let’s see what’s in memory:

```
ADC0_OFFSET @ U. 0  OK
ADC1_OFFSET @ U. 0  OK
MAX_BUS 2@ D. 589874  OK
ADC0_GAIN F@ F. 6.4284E-39  OK
ADC1_GAIN F@ F. 6.4284E-39  OK
CALFLAG @ U. 0  OK
```

The single variables are all zero, but we’ll ignore that for now. Lets write a zero to the first 9 words at the start of User Data Flash ($3000 or 12288d):

```
0 12288 EE!  OK
0 12289 EE!  OK
0 12290 EE!  OK
0 12291 EE!  OK
0 12292 EE!  OK
0 12293 EE!  OK
0 12294 EE!  OK
0 12295 EE!  OK
0 12296 EE!  OK
```

Now for the hard part. We’re going to create a command that will read 9 addresses, starting at 12288, and write them starting at 702. We’ll use the ‘I’ word to get the loop value and add that to our base address:

```
: X 9 0 DO 12288 I + @ 702 I + ! LOOP ;  OK
```

Now, let’s execute the word:

```
X  OK
```

Now it’s time to read the variables:

```
ADC0_OFFSET @ U. 0  OK
ADC1_OFFSET @ U. 0  OK
MAX_BUS 2@ D. 0  OK
ADC0_GAIN F@ F. 0.0000  OK
ADC1_GAIN F@ F. 0.0000  OK
CALFLAG @ U. 0  OK
```

So far, so good. Now, lets erase the User Data Flash and write a value of 2 in each. Note that this will cause the double and float values to read rather more than 2:

```
DECIMAL  OK
12288 EEERASE  OK
2 12288 EE!  OK
2 12289 EE!  OK
2 12290 EE!  OK
2 12291 EE!  OK
2 12292 EE!  OK
2 12293 EE!  OK
2 12294 EE!  OK
2 12295 EE!  OK
2 12296 EE!  OK
```

Lets check the values again ... but we'll save time and use DUMP:

```
12288 9 DUMP
         0    1    2    3    4    5    6    7
12288:  0002 0002 0002 0002 0002 0002 0002 0002 ........
12296:  0002 5535 5535 5535 5535 5535 5535 5535 ........ OK
```

Yes, values are all 2 now.

Let's execute the word X again and then see what the variables hold:

```
X  OK
ADC0_OFFSET @ U. 2  OK
ADC1_OFFSET @ U. 2  OK
MAX_BUS 2@ D. 131074  OK
ADC0_GAIN F@ F. 1.8367E-40  OK
ADC1_GAIN F@ F. 1.8367E-40  OK
CALFLAG @ U. 2  OK
```

Unsure? Lets use DUMP again, this time from address 702:

```
702 9 DUMP
       190  191  192  193  194  195  196  197
 702:  0002 0002 0002 0002 0002 0002 0002 0002 ........
 710:  0002 0004 0068 0085 0077 0080 0032 0065 ..DUMP A OK
```

You can rework the word X to allow for a source address, a destination address and a number of bytes and call the word EEREAD:

```
: EEREAD ( addr1 addr2 length -- )
  CR
  ROT      \ Pulls the start address to the top of the stack
  SWAP   \ Pulls the length to the top, above the start address
  0 DO    \ Uses the length as the start of the loop counter
  2DUP    \ Make a copy of the start and stop addresses for each loop
  I + .      \ Get the loop count and add to the source address, and print
  I + .      \ Get the loop count and add to the destination address, and print
  CR        \ print a carriage return to keep things clean
  LOOP    \ end of the loop command
  2DROP  \ clean the two remaining variables off the stack
;  OK
```

gives us:

```
12288 702 9 EEREAD
12288 702
12289 703
12290 704
12291 705
12292 706
12293 707
12294 708
12295 709
12296 710
 OK
```

Now issue the command

```
FORGET EEREAD
```

so we can rework it a bit. We'll remove the carriage returns and change the first print statement to a @ . Insert a SWAP, so that we have the destination address on top of the stack, and exchange the second print command to a ! as follows:

```
: EEREAD ( addr1 addr2 length -- )
  ROT     \ Pulls the start address to the top of the stack
  SWAP   \ Pulls the length to the top, above the start address
  0 DO    \ Uses the length as the start of the loop counter
  2DUP    \ Make a copy of the start and stop addresses for each loop
  I + @     \ Get the loop count and add to the source address, and print
  SWAP    \ Need the next address
  I + !     \ Get the loop count and add to the destination address, and print
  LOOP    \ end of the loop command
  2DROP  \ clean the two remaining variables off the stack
;
```

You could also assign individual words such as:

```
: EEADC0_OFFSET 2BE ; \ the value 2BE is placed on the stack whenever you call
                      \ EEADC0_OFFSET
```

and then perform the following:

```
EEADC0_OFFSET @ ADC0_OFFSET !
```

Each has it's good and bad points, but remember that you will only need to perform this at startup (I like to put stuff like this in MAIN).

Later on, you will need to update your Data Flash memory, such as when you recalibrate. This part gets easy. Simply have your calibration program store all your variables, such as:

```
HEX
8 ADC0_OFFSET !
```

Then, when you have finished calibrating, you will issue this command:

```
HEX
EEMOVE 2BE 1800 C  (EEMOVE 702 6144 12)
```

This copies twelve variables $2BE-2CA to address $1800-180C. You can reset, scrub, whatever, and your variables are safely stashed in User Data Flash. If you have read your variables prior to updating them, the variables that you don't update are saved and reloaded without having to keep track of them. That's letting IsoMax work for you.

Based on the IsoMax Memory Maps.pdf document (http://www.newmicros.com/store/product_manual/IsoMax%20Memory%20Maps.pdf), from V0.6 up there are 8 pages available, each with $100 (256 decimal) words for the 56F803/5 processors (IsoPod, MiniPod, TinyPod, and PlugaPod), and there are 16 pages available, each with $100 (256 decimal) words for the 56F807 processor (ServoPod). Other memory areas have changed, but the User Data Flash has remained stable from at least V0.6.

I highly recommend setting a variable that indicates whether any of the User Data Flash memory has been initialized by your program. If it hasn't, all values will be 65535d (or $FFFF or all ones in binary). Setting your variable to 0 will clear all the bits. Note that you CAN rewrite to this memory as long as you are clearing bits. Try erasing an address block, write $DDDD to one address. Now you can fetch the data at the address and see the value $DDDD. Reset the processor and switch back to HEX mode. Again, fetch the data at the address and verify it is still $DDDD. You can even scrub the processor, the value remains $DDDD. Now, write a zero to that location. Repeat the above exercises and find that the value is now 0. You can try writing FFFF to this location, but it will not be accepted. You MUST perform an EEERASE for the particular memory page you are working with. I HIGHLY recommend your application read this memory page before you erase. This way you have all your variables and simply update them in your program before writing them to the freshly erased memory. This allows you to selectively update variables without having to repeat calibration on all the ADC channels for example (and there are 16 ADC channels on the ServoPod).

I hope this helps you understand User Data Flash memory.

I want to express my great appreciation to the New Micros support staff for my many questions over the last few years.

Rick Mainhart