

# CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

## LESSONS FROM THE TRENCHES

George Martin

### Managing Your Memory

There's a right and wrong way to go about anything. And, far too often, especially in programming, you can get away with what's less than best. Having personally tried all the less-than-best methods to write memory management routines, George invites you to listen in as he shares the best way to accomplish the task.



Last month, I covered low-level interfacing to a serial electrically erasable (EE) memory. These devices are a low-cost, effective way to store data that needs to survive when the power is turned off. The device I wrote about was rather old, but the basics for that device extend to the latest and greatest. I described an interface that used two pins—one for the clock (output only) and one for bidirectional data. Your specific design may be different, but I think I selected the most general and it should be easy to adapt.

In last month's discussion, we ended up with two key routines that interface to the memory:

```
char EERead_char(int i) //  
  Read a character from EE  
  memory  
void EEwrite_char(int i, char  
  c) // Write a character to  
  EE memory
```

This month, I'd like to build memory management routines that control the EE memory and are at the

next higher level of code. I'll develop these routines so that the system can interface to:

```
SaveEEMemory // Save all the  
  data into EE memory  
RestoreEEMemory // Restore  
  all the data from EE memory
```

The basic routine reads and writes a character that involves 8 bits of data. But, what if we need to work with integers, which use 16 bits of data? Check out Listing 1 for a quick-and-dirty way to do it.

Although these routines work, they're ugly and full of pitfalls. Shifting the masked bits and converting them from *int* to *char* in the *EEWriteInt* routine is dangerous. They're not supposed to, but different compilers handle this routine differently. The parentheses help, but there are no guarantees. The same sort of problems exist in the *EEReadInt* routine. If the results of *read\_EE* need to be changed from a *char* to an *int*, the conversion needs to be done using unsigned characters, so signs are not promoted. There's usually a compiler switch set somewhere (not too obvious) to control these settings.

#### WORKING WITH FLOATING NUMBERS

The situation changes again if we need to work with floating-point numbers. Listing 2 illustrates one way to make that happen.

This approach is better since we're not converting and shifting data. There's less chance for the compiler to help us out and cause problems. Both these *float* and *int* routines could be cleaner if we used a union to map a *char* array and put the *float* or *int* into the same memory locations.

The requirements keep on coming: what if we need to store a double or a

string? Then we'd need a different routine for each type of data we wanted to read and write, and a different address for each variable. And, what do we do when a change comes down? (I guarantee you'll get one the Friday before your vacation starts.) Lots of work, lots of errors, and no vacation.

## STRUCTURED DATA

Consider a different approach. Let's put all the data into a structure like the one in Listing 3 (to see the entire program, download EEMec.c). The structure is a collection of all the types of variables required for your project. Strings and arrays are easily supported. You just let the language do all the bookkeeping. If data is added or removed, just change the structure, and all the details are taken care of.

By the way, the compiler will place multibyte data in the structure according to specific machine's preferences. Intel and Motorola, for example, use different-looking structures for exactly the same data. Also, pad bytes will be added to keep all the data in the structure on the proper boundaries. For instance, a character array of nine bytes would have a one-byte pad to get it to an even number of bytes. The pad bytes only take up room. They don't add any other obstacle.

I've set up the structure with some variables that aren't actually part of the data the project uses. The *DATA\_00* and *DATA\_AA* variables, for example, are markers to identify valid data in the structure. *DATA\_END* helps us find the end of the data. *chksum* maintains the checksum.

Listing 4 presents the routine *unsigned long calc\_EEchksum (void)*. Here, after initializing a pointer with the address of *EE.DATA\_AA*, we do a simple read and add the data up to, but not including, *DATA\_00*. You could use a more robust technique such as CRC, which would also indicate whether the data was valid.

The main interface to all this are the routines *saveEE* and *restoreEE* (see Listing 5). *RestoreEE* restores all your working variables with the contents of the EE memory. It calls *load\_EE*, which reads the EE

**Listing 1**—This method is not the best way to go about reading and writing an integer into EE memory, but it works.

```

a)
int EEWriteInt(int i, int d) {           // this routine writes an
                                        // integer to EE memory
    char c;

    c = (char) ((d >>8) & 0xff);        // get the 8 most significant
                                        // bits
    EEwrite_char(i, c);

    c = (char) (d & 0xff);              // get the 8 least significant
                                        // bits
    EEwrite_char(i, c);
}

b)
int EEReadInt(int i) {                  // this routine reads an integer from EE
                                        // memory
    int j;

    j = (int) read_EE(i++);            // get the 8 most significant bits
    j = j <<8;

    j = j + (int) read_EE(i);          // get the 8 least significant bits
    return(j);
}

```

**Listing 2**—This method is another way to handle integers and characters, but you'd better not need a string or double.

```

a)
int EEWriteFloat(int I, float f) {     // this routine writes FP to
                                        // EE memory
    unsigned char *p;                  // creates a pointer to an
                                        // unsigned character
    p = (unsigned char*) &f;          // points to the start of the
                                        // data

    write_EE(i++, *p++);               // write and advance the
                                        // pointers

    write_EE(i++, *p++);
    write_EE(i++, *p++);
    write_EE(i++, *p++);
}

b)
int EEReadFloat(int i) {               // this routine reads FP from EE
                                        // memory
    char[4]                             // create a four-byte array
    float *f;                            // pointer to a float
    float r;                              // a float

    f = (float *)&c[0];               // set the pointer to point to the
                                        // array

    c[0] = read_EE(i++);                // fill the array
    c[1] = read_EE(i++);
    c[2] = read_EE(i++);
    c[3] = read_EE(i++);
    r = *f;                              // read the data as a float

    return( r);                        // return the data
}

```

**Listing 3**—The best way to accommodate the different types is by providing a structure that takes all types into account.

```
#define EE_START 1           // EE starting location

typedef struct {           // Define EEPROM structure for saving
                           settings

    unsigned char DATA_AA; // this should be an 0xAA

    int var1;
    int var2;
    char var3;
    char MfgDate[12];
    char var4;
    char var5;
    int var6[2];
    float version;
    int var7;

    // add new elements here

    char DATA_00;           // this should be a 0
    unsigned long chksum;    // checksum of the data
    char DATA_END;         // this data value is never used
```

memory, places the data into the structure, and checks the data for validity. If valid data is present, then everything is OK. If invalid data is discovered, the default actions are determined by `load_EE_defaults`.

This design provides for unattended operation. In `restoreEE`, there is a flag called `fact_default`. If this flag is set, factory defaults are loaded, which in this particular design occurs through a key sequence or a serial port command. I left these features in to show you where you might want to hook in.

`load_EE_defaults` loads the default settings. The code I've shown is

not very enlightening, but it does reserve a placeholder for your actual code. These variables can be constants or calculated numbers.

The `saveEE` routine writes data into the EE memory. A couple of real-world and application-specific issues are dealt with here. First, it checks the pointers from the serial-data-input routine. Anyone remember that article? If serial data is received and pending, we back off checking EE memory for about 1 s. In this project, serial commands frequently control the device and all control changes are written to EE memory.

Next, there is a flag called

`ee_chnged`. The code that runs the unit looks at the work it does. Whenever that work changes a parameter saved in EE memory, the code sets the flag. The `saveEE` routine is constantly called in the main processing loop and, if the flag is set, then `saveEE` starts processing the data.

That processing verifies the data in the structure byte by byte with the contents of EE memory. If a difference is found, that byte is written into the EE memory. Once the structure has been verified as current, the flags are cleared.

I'm always amazed once I see how simple the answer to a complicated problem can be. I believe I've taken every wrong turn in the development of this code over the years. I hope you can use this code as a starting point for bigger and better work.

Next time, I would like to detect bad EE memory and change the starting address. This leads naturally into learning how to write data into memory in several places. Of course, this is a big saver. Then, if a power glitch gets me when I'm working on one of the copies, I can use another copy.

I'll use the same code in the next issue that I used here. I won't change the top-level routines, just the lower-level ones. All the changes should be transparent.

*George started his career in the aerospace industry in 1969. After five years at a real job, he set out on his own and cofounded a design and manufacturing firm. Typical systems that George designs include servo-motion control, graphical input and output, data acquisition, and remote control. George is a charter member of the Ciarcia Design Works Team and most recently, he's been working on the people-tracking system for Bill Gates's new house. You can reach him at [george.martin@worldnet.att.net](mailto:george.martin@worldnet.att.net).*

**Listing 4**—This routine calculates the checksum for the EE data starting with the location `EE.DATA_AA` and ending with `EE.DATA_00`. The checksum is the numeric sum of all the locations.

```
unsigned long calc_EEchksum(void)
{
    unsigned char *p;
    unsigned long temp;

    p = (unsigned char *) &EE.DATA_AA;
    temp = 0L;

    while (p != (unsigned char *) &EE.DATA_00)
    {
        temp = temp + *p++;
    }

    return(temp);
} // end of unsigned long calc_EEchksum(void)
```

**Circuit Cellar, the Magazine for Computer Applications.** Reprinted by permission. For subscription information, call (860) 875-2199 or [subscribe@circuitcellar.com](mailto:subscribe@circuitcellar.com).

**Listing 5a**—This routine restores variables from memory, while the second routine (b) saves variables to memory if they've changed. Only one variable is saved on each call, and the checksum is updated and saved

```

a)
void restoreEE(void)
{
    if ((load_EE() == 0) || (fact_default == ON)) { // EE failed or
        load_EE_defaults();                       //factory defaults re-
                                                    requested
        ee_chnged = 1;
    }

    /**** Put all the routines here *****/
} // end of void restoreEE(void)

b)
void saveEE(void) {
    unsigned char *s;
    int loc;
    int ck;

    if (com_in_pt != com_ot_pt) {                // any com data, don't
                                                do EE yet
        ee_timer = 0;
        return;
    }
    if (ee_timer < 100) {                       // just a long 1-s delay
        return;
    }
    ee_timer = 100;                             // hold it here

    if (ee_chnged == 0){                       // nothing to save;
                                                just exit
        return;
    }

    if (ee_chnged == 1){                       // this is the start so
                                                recalc the chksum
        EE.chksum = calc_EEchksum();
    }

    loc = EE_START;                            // start in EE at this
                                                location
    s = (unsigned char *) &EE.DATA_AA;        // point to the start
    ck = ee_chnged - 1;

    if (&s[ck] != (unsigned char *) &EE.DATA_END) {
                                                // has data changed?
    if (s[ck] != EEread_char(loc+ck)){        // yes; EEwrite_char
                                                has a 10-20-ms delay
        EEwrite_char(loc+ck, s[ck]);
        }
    }

    else{
                                                // all done */
        ee_chnged = 0;
    }

    ee_chnged++;
}

// end of void
saveEE(void)

```