

# CIRCUIT CELLAR®

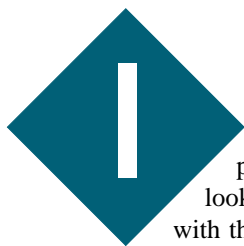
THE MAGAZINE FOR COMPUTER APPLICATIONS

## LESSONS FROM THE TRENCHES

George Martin

### Working With a Little EE

When it comes to data-logging projects, it's important to be able to store data and settings, even when the power cycles off. Electrically erasable memory offers a solution to this problem. This month, George shows us how to interface with an electrically erasable device to support your data-logger's requirements.



Last month, I promised that I'd look at how to chart with this data-logger program, but I ran into some problems. I thought I could use Microsoft's *Chart* command, but I got nowhere and Microsoft offered little (no) help. I started looking for other shareware charting stuff, but I haven't had any luck there yet either. And, I didn't want to just write about my troubles, so I switched gears.

Have you ever been asked to design an embedded system that needed to remember some data or settings even after the power cycles off? How about saving calibration constants or system configuration?

If you've ever faced these or similar system requirements, you've probably considered electrically erasable (EE) memory. Certainly, a disk or battery-backed-up SRAM would do the job, but not every system is equipped with those devices.

EE memory can be written to in the system and retain the data without power for 200 years. In the overall picture of system-cost allocation, EE

memory is expensive per bit (you find 32-Kb devices for less than \$2 in low volume). Some CPU devices now come with their own EE memory.

In this article, I'd like to present the overall picture of how you can add an external EE memory device to your design. I'll show you how to interface the device, along with the software, to support system requirements.

Let's start with the hardware interfacing details. I'll present the most involved design approach. If your hardware offers more support, you can simplify what I describe by using the delete key. I'm using Microchip's 24C32P. Figure 1 shows you the pinout for this

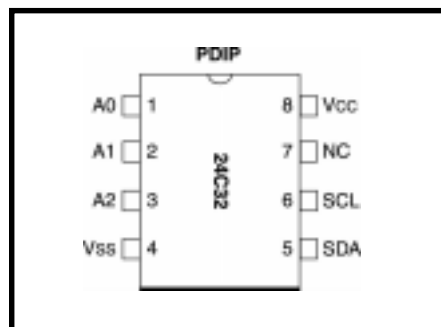


Figure 1—The Microchip 24C32P offers electrically erasable memory. Each chip contains three address lines that can be uniquely identified through values included in the address.

8-pin integrated circuit.

#### THE BASICS

The data exchange with the 24C32P is through the SCL (serial clock) and SDA (serial data) pins. The clock signal indicates when data should be transferred to or from the device, and the clock pin serves as a bidirectional output pin from the CPU. The data signal is used to read data from or write it to the device.

When data is written to the memory, the CPU outputs a one or zero to the memory. When data is read from the memory, the CPU writes a high to the

data pin and then reads it as an input. The memory drives the data pin either high or low for the CPU to read. Up to eight 24C32P devices can have their clock and data pins tied together and, by using address pins A0, A1, and A2 as hard-wired 1 of 8 decode-logic inputs, each chip's memory can be accessed individually.

Let me point out that, as you read and write this type of memory (flash memory works this way too), there's a lot going on in the background. A write command may actually perform a read-modify-write or an erase-write cycle. You only receive a busy/ready indication, signifying completion of the operation.

## THE READ/WRITE SEQUENCE

On powerup, put the device in standby mode by raising the clock and the data lines. In this state, no read or write activity takes place.

The first command written to the device is *Start Data Transfer*, which is activated by lowering the data line while the clock is high. After you've lowered the clock line, set the data line (either high or low), and then give the last command, *Stop Data Transfer*, which raises the data line. One bit is then transferred to or from the device. This serial sequencing continues until all the data is transferred.

Let's look at the requirements for reading a byte of data. The read sequence consists of the following operations:

```
Send Start Data Transfer
Send Control Byte
Send Address Byte 1
Send Address Byte 0
Read Data Byte
Send Stop Data Transfer/
```

The write sequence looks very similar:

```
Send Start Data Transfer
Send Control Byte
Send Address Byte 1
Send Address Byte 0
Write Data Byte
Send Stop Data Transfer/
```

Function	Prototype
Start an EE transfer	void EE_start(void)
Stop an EE transfer	void EE_stop(void)
Read the acknowledge	char read_ack(void)
Write an EE bit	void wr_EEbit(char b)
Read an EE bit	char rd_EEbit(void)
Read a character	char EEread_char(int i)
Write a character	void EEwrite_char(int i, char c)

**Table 1**—All it takes is a few routines to manage Microchip's 24C32P. Download *EEemem.c* to see how I've implemented the routines into my code.

After every byte transfer, the 24C32P acknowledges the transfer, indicating that it went well. If you don't get an acknowledgement, you can assume there's been an error.

Other types of memory have different command sequences, so don't just blindly type in commands. Instead, look carefully at the datasheet. Also, the 24C32P supports block reading and writing. You only need to write the command and address once. After that, you can speed up transferring large blocks of data by using multiple read or write operations.

## GETTING THE ROUTINE DOWN

At a quick glance, it's encouraging to be able to write just a few routines to manage the device. Table 1 extracts some of the routines developed in *EEemem.c*.

These are basic bit operations, which can be built into larger functions. Check out the datasheet ([www.microchip.com](http://www.microchip.com)); there are some timing requirements to consider:

```
Clock high (min):      4000 ns
Clock low (min):      4700 ns
Output valid from clock (max):
                        3500 ns
Bus free time (min):  4700 ns
```

The EE memory I'm writing about has a vintage 1995 datasheet. The microprocessor I used is an 80C196, operating at about 10 MHz. As you can see in the code, I don't have to worry about going too fast for the device. Because 1995 is ancient history in today's world of technology, check and double-check your datasheets.

I didn't attempt to streamline these routines in any way. For my project, the EE memory's reads and writes were so

infrequent that it wasn't necessary. Also, I reserved a place for a delay routine that wasn't needed since the microprocessor did not exceed the timing requirements of the EE device.

## DEBUGGING YOUR READS AND WRITES

If you read my earlier columns in the Circuit Cellar Design Forum ([www.cir-cuitcellar.com](http://www.cir-cuitcellar.com)),

you know I love debugging embedded code on the desktop with Turbo C. When you see references to Intel and Turbo C, you know what I'm up to.

Also, when I need to pass a bit of data, I use an entire *char*. Don't forget that some of the new high-performance CPUs take just as much time and code, if not more, to pass *chars* as *ints*. Check the output of your compiler and make sure you're not adding unnecessary delays.

When I'm desk debugging with all Turbo C flags enabled, I'm simulating EE memory with a big character array. All the reads and writes are from that array:

```
Read a character char
  EEread_char(int i)
Write a character void
  EEwrite_char(int i,
  char c)
```

Use error checking. EE memory is supposed to acknowledge each byte it reads. If one of these is missing, it's an error. By giving each place a different error value, it greatly aids in debugging the design.

You should also notice in the write-a-character routine that there's a timer. EE memory takes 1 ms to write. In this project, I reset a timer that was incremented in the real-time interrupt routine every millisecond to increment at the value of 2. With this method, the code just waits for the timer to expire. It's not the most clever real-time effort, but it was all this project needed.

When you're accessing the EE memory, the address is an integer (16 bits in this design). Be careful if you're using bigger EE memory devices. The address might not fit into an integer, and you might need to use a 32-bit address.

The other routines found in this module just support the read and write character operations.

## COMING UP

Now that the EE memory is under control and performing basic operations, such as reading and writing one byte of data, I'll move on to cover the EE memory from a system point of view. Next month, I'll look at issues like how to save floating-point data and how to know if the data in memory is valid.

*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).*

## SOURCE

### 24C32P

Microchip Technology, Inc.  
(480) 786-7200  
Fax: (480) 899-9210  
[www.microchip.com](http://www.microchip.com)

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).