CIRCUIT CELLAR INK®
THE COMPUTER APPLICATIONS JOURNAL

FEATURE ARTICLE

Ethan Bordeaux
& Stefan Hacker

Integrating Flash Memory in an Embedded System

Turn on to the power and ease of flash! Ethan and Stefan show us the best uses for flash memory, and create an interface between a DSP and flash. It's easy to tie into your system, and you can't beat having in-system programmability.

ntil recently, the most flexible external boot memory was an EPROM. But, if you needed to erase or update the data, you had to remove the EPROM from the system, expose it to ultraviolet light, place it in an EPROM burner, and insert it back into the design.

A better solution is to update code and data while the nonvolatile memory is in the system. This is the type of functionality built into flash memory.

We're sure you can imagine lots of applications and system configurations where such functionality is useful.

One application is an embedded speaker-independent voice-recognition unit, as in a hands-free car kit or voice-activated appliance. The user programs a number of keywords to perform operations like dialing a phone number or turning on and off a device. The processor would need to store these voice patterns in external nonvolatile memory and be able to retrieve them for comparison purposes.

Flash memories are also an asset in systems that need to save data during a power outage or brownout. The processor moves its code and data contents from volatile internal memory to an external nonvolatile memory and, on revival of the system, continues at the last saved state.

In this article, we explain some of the benefits and basic functionality of flash memories. We cover an example interface between an ADSP-218*x* DSP and an Am29F040 flash memory as well as hardware and software structure.

FLASH VS. EPROM

Even with the in-system programmability (ISP) of flash memory, there are times when a conventional EPROM may be a better choice for your design. Table 1 is a partial list of considerations for choosing a byte memory.

EPROMs cost less than flash memory with similar storage capabilities, but there are many reasons to consider using flash memory. ISP is an obvious advantage. They can potentially offer all of the functionality of an EPROM and an SRAM in a single package.

Because flash memory is such a hot commodity in today's semiconductor market, many manufacturers are focusing research and development, along with their advanced manufacturing processes, on the flash market. This translates into flash memories with low power consumption during operation and powerdown, more aggressive operational voltages, and faster access times (e.g., Intel's Strato flash family).

So, if your design requires the absolute lowest power consumption, a flash memory may be the best nonvolatile solution. And because operational voltages on flashes have now reached 1.8 V, they can gluelessly interface to systems that operate below LVTTL levels.

FLASH ROBUSTNESS

Flash memories are given a rating for the number of write cycles they can sustain before the part is not guaranteed to operate properly. Flash memories

from a few years ago sustained ~10,000 write cycles. Assuming your design is supposed to last 10 years, this translates to ~2.7 memory rewrites a day.

Some systems are deterministic enough to guarantee this rate, but often it's unknown exactly how many times the flash memory will be programmed or erased in its lifetime. Some flash memories now guarantee upward of 1,000,000 write cycles (or 270 rewrites a day) before failure. This alleviates some of the flash programming concerns and enables them to fit into a wider variety of designs.

INTERNAL VS. EXTERNAL

There's been a lot of talk recently about embedding flash memory on processors, microcontrollers, or DSPs and the increased level of system flexibility that would provide.

The strongest argument for including flash memory on a processor is for prototyping. If a manufacturer provides a processor with both a ROM and flash-memory variant, the user can use the flash device for prototyping and the ROM-coded processor for production.

But, processors with embedded flash memory cost 2–10× more than their flash-less counterparts and are aimed at prototyping environments. And, flash memories frequently can't operate as fast as onboard SRAM or ROM. The entire processor speed can be compromised by including flash memory.

Lastly, the size of a flash memory embedded with a processor is typically 10–30k words. Although this is often adequate for a portion of program or data memory, many times, a design needs access to a much larger memory space, where it can grab many code and data overlays or use a nonvolatile memory source as a virtual hard drive.

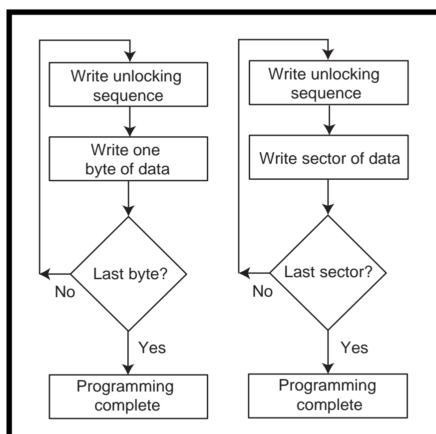

Figure 1—*A tale of two programming models: the left-hand side (Am29F040B) requires an unlocking sequence after every byte is programmed, and the right-hand side (AT29C040A) requires an entire sector of memory to be programmed each time you write to the flash.*

	EPROM	Flash memory
Price per megabit	$1–3	$2–10, depending on features
Typical packages	PDIP, PLCC, BGA	BGA, PDIP, PLCC, TSOP
Typical operational voltages	2.5 V, 3.3 V, 5 V	1.8 V, 2.5 V, 3.3 V, 5 V
Power consumption	30–200 mW	15–50 mW (read), 45–200 mW (write/erase)
Typical access time	60–200 ns	18–150 ns

Table 1—*Even though flash memories are typically more expensive than EPROMs, a wide range of voltages, fast access times, low power consumption, and inherent ISP can make them powerful and flexible memory ICs.*

Applications like digital cameras require many megabytes of nonvolatile storage for saving each "roll" of digital film. Algorithms such as voice recognition need large external memories for voice look-up tables. To store 30 words for a speaker-independent voice-recognition system, up to 512 KB of external memory is needed.

These applications can't be supported by processors with on-chip flash memory. These systems require a simple and flexible method of gluelessly connecting an external flash for the optimal solution.

EMBEDDED ALGORITHMS

The simplest way to understand a flash memory is to think of an SRAM programmed via a finite state machine (FSM). When reading information from the flash memory, you have normal access to all memory locations, much like an SRAM or EPROM.

But, when specific operations need to be performed on the flash, whether it's erasing, writing information, or protecting memory segments from erroneous erasure, the processor must use the flash's embedded programming algorithms (EPAs).

A series of commands are written from the processor to the flash. These commands unlock the flash so it can accept data, erase sectors, or perform other programming tasks. We explain a few of the common functions here.

BYTE/SECTOR WRITE

Byte/sector write lets the host processor place data into flash memory. The processor first writes the unlock sequence to the flash and then writes the address and data for the first value to be programmed.

Depending on the programming methodology implemented on the flash, additional unlocking commands need to be written for each word, or a sector (typically between 64 bytes and 64 KB) of memory can be written in a burst fashion, as you see in Figure 1.

In both types of flash memories, the processor writes the unlocking sequences and data to be programmed into the flash and the flash latches the data into memory. This sequence enables the processor to continue executing algorithmic data while the flash memory handles programming itself.

Keep in mind that the contents of the block you want to program must be erased beforehand. A convenient feature of sector-programmed flash memories is that they usually erase the sector before programming it with data.

One methodology is not inherently better than another, but each is better suited for certain systems. Consider these points before choosing a flash-writing architecture.

First, decide what type of data you're writing. Will your data be partitioned as a large chunk of information (e.g., a JPEG file for a digital camera) or will the external memory save single bytes of information?

If you're only making small incremental changes to the flash memory, it might make sense to choose a flash with a byte-programming protocol. But, if you write large pieces of data from the processor, a sector-programmed flash may be more efficient.

Second, what kind of processor is writing to the flash? Low overhead DMA and fast byte-port accesses are available in some embedded processors.

For example, the ADSP-218*x* DSPs

integrate a byte-wide DMA port that supports a variety of ICs, including flash memories. Knowing how your processor will interface with a flash and its EPAs is useful.

SECTOR/CHIP ERASE

Flash memories are partitioned into a number of sectors, which enables the programmer to erase one sector of the memory at a time.

In general, a series of commands is written out to the flash to start the erasing procedure. Even though the commands are latched into the flash in a few microseconds, actually erasing the flash takes a couple seconds.

Some flash memories don't allow the processor access to the flash during a sector erase. Others, however, have simultaneous read/write architectures that permit memory accesses to one block while the other block is erased.

AUTOSELECT/PRODUCT ID

This EPA determines specific information about the flash in your system. The information includes the manufacturer ID (necessary because many companies make pin-for-pin–compatible flash devices), device ID, and sector-protection feedback. This data is useful for external flash programmers/burners.

SECTOR PROTECT

The sector-protect EPA disables both the programming and erasing of a particular memory sector. This feature is useful in systems where you both boot and continually write/erase the flash. You can set the boot sector to be protected and leave other memory segments unprotected for a chip erase.

DATA POLLING

The time it takes a specific EPA to finish varies greatly, even on the same device. For example, the Am29F040B flash memory takes from 7 to 300 µs to program a byte of data, and sector erase time can vary from 1 to 8 s. There are provisions for the flash memory to signal when an operation completes.

Data polling is a common method for determining when an EPA is finished. Once the processor writes the data into the flash, it is latched inside the part and the processor isn't needed to control the actual writing of data into memory. But, the flash still needs to place the new information into its memory bank and signal when it's ready to handle another EPA.

One method is to poll the status of one of the data bits to see if the EPA has completed. For example, the Am-29F040 provides data-polling capability on data pins DQ7 and DQ6.

By reading back DQ7's value during a byte write, you can determine if the programming is done. If DQ7 has the inverted value of the programmed value (e.g., the byte value programmed into the flash is 0x0F [DQ7 = 0] and DQ7 is read back as a 1), the EPA is not complete. If DQ7 reads back as the value programmed on that bit, the EPA is done and the next byte can be programmed.

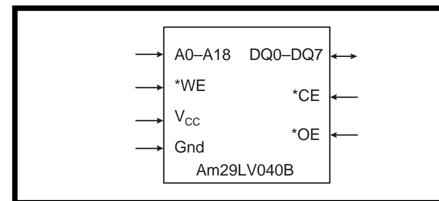

Figure 2—*The Am29LV040B has memory strobes similar to an EPROM, with the exception of a write pin (WE).*

DQ6 can be used in a similar way. While the flash is still in a byte-programming EPA, successive reads of DQ6 cause the value to toggle. When the EPA finishes, DQ6's value stops toggling and reads as the same data value. The data-polling scheme you choose depends on your processor and whichever method is easier to implement in software.

READY/*BUSY PIN

A lower-overhead method of determining EPA completion is via a READY/*BUSY pin. Some newer flash memories include this pin to signal the status of the flash at any moment. Essentially, the pin is at a logic low when the flash is in any EPA and at a logic high when it's ready to read data or in standby.

This method is useful on processors where the status of external flag pins can be easily tested. ADSP-218*x* DSPs provide a variety of I/O pins and support for externally generated interrupts, which can be connected to test the status of READY/*BUSY.

The tradeoff is that this method ties up an additional pin and increases the total number of signals in your design.

FLASH TO DSP

Now that you're acquainted with some typical flash EPAs, here's an example using an ADSP-2184L DSP and Am29LV040B flash memory.

The Am29LV040B (see Figure 2) has a basic set of memory strobes. Other flashes may provide READY/*BUSY strobes, hardware pins for locking the contents of the memory, or methods of reading information for synchronous burst transfers. We chose this memory primarily for its simple hardware and software interfaces.

Listing 1—*This code shows you the function* `PROG_BYTE`.

```
* Flash application server
* int error PROG_BYTE(char c_byte, char d_byte, int addr_lo, int
*    addr_hi);
* Byte program:
*   c_byte : 0xA0              // Third input to AMD EPA
*   d_byte : value to program  // holds data to be programmed to flash
*   addr_lo: low 16bit         // lower part of 22-bit address
*   addr_hi: high 6bit         // higher part of 22-bit address
* Register usage summary:
*   modify : addr_lt, addr_ht, d_btmp
*   destroy: ar, ax0, ay0, af
*   calls : init_seq, cmd_write, calc_adr, DQ7_poll

prog_byte:
  call init_seq;               // EPA unlock sequence
  call cmd_write;              // EPA command word write
  ar = dm(d_byte);             // fetch byte from input register
  dm(d_btmp) = ar;             // store byte in destination register
  call calc_adr;               // compute registers from address
  call DQ7_poll;               // check for internal completion
  rts;                         // return from function call
```

The ADSP-2184L is a 16-bit fixed-point DSP. This processor family contains a number of external interfaces including an external byte-wide DMA port that can be configured to support 8-bit memories, including flash. Figure 3 shows the pins that the DSP uses to connect to an external memory.

There are 14 address and 24 data lines available externally on the DSP. This configuration enables direct external-program memory execution because its opcodes are 24 bits wide. But, when the DSP accesses external byte space, only D8–D15 are connected to the flash memory data bus, allowing D16–D23 to become address bits and creating a total address reach of 4 MB.

The DSP can supply the necessary chip-, read-, and write-select strobes to the flash. Because the ADSP-2184L operates at speeds up to 40 MHz and flash memory may be just 5–6 MHz, the on-chip wait-state generation logic enables a flash interface without external glue logic.

The ADSP-2184L does not operate out of external byte-wide memory. That memory is typically too slow to support the processor's operational speeds. Instead, the DSP transfers the information from the byte-wide memory into internal SRAM and operates from this memory space.

There are hardware provisions for packing data and program-memory words (16 and 24 bits) into internal memory. Byte memory space is intended to be a large region of memory where the processor can fetch instructions or data to be operated on or to save externally to the chip for later use. Software development tools let code and data be saved as memory overlays, which can be loaded into the DSP under control of an overlay manager.

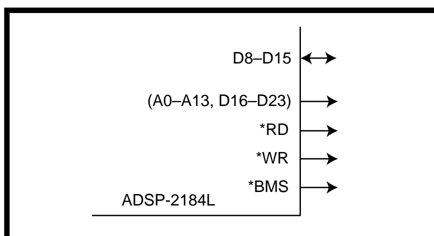

Figure 3—*The ADSP-2184L provides all the memory strobes required to control the flash memory.*

Function	Purpose
BDMA_SETUP	Initializes the byte-wide memory port for flash transfers
PROG_BYTE	Write one byte of information to flash
READ_BYTE	Read one byte of information
SECT_ERASE	Erase one sector of information
MEM_IDENT	Identify the manufacturer
AUTO_INC	Automatic incrementing of address for multiple-burst transfers

Table 2—*In our software, we adopted a function-calling scheme that hides much of the underlying protocols and architecture of the flash memory.*

FLASH SOFTWARE

The code modules of our ADSP-2187L flash software enable the programmer to access flash memory via function call APIs. Table 2 lists the base functions, along with their purpose.

Each function expects input data to be located in specific registers and output data to be placed into specific registers. Once you understand the rules, it's simple to tie the function calls into your software. An example of the assembly code used to program one byte of memory on the ADSP-2184L is in Listing 1.

The code shows the first layer of the software interface between the flash and DSP. Because many of the basic building blocks for each of the EPAs are similar, an additional set of functions (`INIT_SEQ`, `CMD_WRITE`, `CALC_ADR`, and `DQ7_POLL`) were written for `PROG_BYTE` to call.

This function assumes that the flash software housekeeping function (`BDMA_SETUP`), which configures the DSP for byte transfers, has already been called.

There are several steps to the basic program flow for `PROG_BYTE`. First, place the appropriate values into the addresses pointed to by `c_byte`, `d_byte`, `addr_lo`, and `addr_hi`.

Be sure that the values contained in the ar, ax0, ay0, and af registers are no longer needed. If they are needed after `PROG_BYTE` completes, the background register set on the ADSP-2184L can be used for programming operations and then switched back when the function ends.

`PROG_BYTE` calls two functions, `INIT_SEQ` and `CMD_WRITE`, which write the first three bytes of information into the Am29F040B's FSM to unlock the memory.

The third step is a memory transfer between the input data location and the API. This transfer is necessary to save the value of the data word for use later when polling the data.

`CALC_ADR` creates the address where the data is stored in the flash. It performs the last byte transfer to the flash.

Lastly, `DQ7_POLL` (which continually tests the status of the DQ7 bit) is called to determine when the flash has transferred the data into memory and when it's free to enter a new EPA.

NOW IT'S YOUR TURN

This flash interface is easy to tie into your application code. With a set of reference functions for interfacing these devices, you can add new functions as your system needs change and flash technology develops. ■

Ethan Bordeaux works for Analog Devices as a 16-bit DSP product line applications engineer. He has worked with embedded speech processing applications including speaker identification, speech recognition, and adaptive echo cancellation systems. You may reach him at ethan.bordeaux@analog.com.

Stefan Hacker is a DSP applications engineer at the European DSP support center for Analog Devices. His focus is on OEM accounts using 16- and 32-bit DSP products in consumer and industrial applications. You may reach him at stefan.hacker@analog.com.

SOFTWARE

The functions described here and instructions for linking them into your design are available via the Circuit Cellar web site.

REFERENCES

Advanced Micro Devices, *Am29-F040B*, Datasheet, 1998.

Analog Devices, *ADSP-2100 Family Users Manual,* 1995.

Atmel, *AT29C040A Datasheet*, 1998.

C. Leidigh, "Flash Memory Buyer's Guide," *Communications Systems Design*, March 1998.

SOURCES

Am29F040B, Am29LV040B
Advanced Micro Devices, Inc.
(408) 732-2400
Fax: (408) 732-7216
www.amd.com

ADSP-218*x*
Analog Devices
(781) 329-4700
Fax: (781) 329-1241
www.analog.com/dsp

AT29C040A
Atmel Corp.
(408) 441-0311
Fax: (408) 436-4200
www.atmel.com

©Circuit Cellar INK, the Computer Applications Journal. Reprinted by permission. For subscription information, call (860) 875-2199 or subscribe ©circellar.com