FUJITSU

Application Note

Microcontroller interface to external Flash Memory (e.g. MB90670/75 series)

20.3.1997 Vers. 1.0 by E.Bendels

Fantastic FLASH

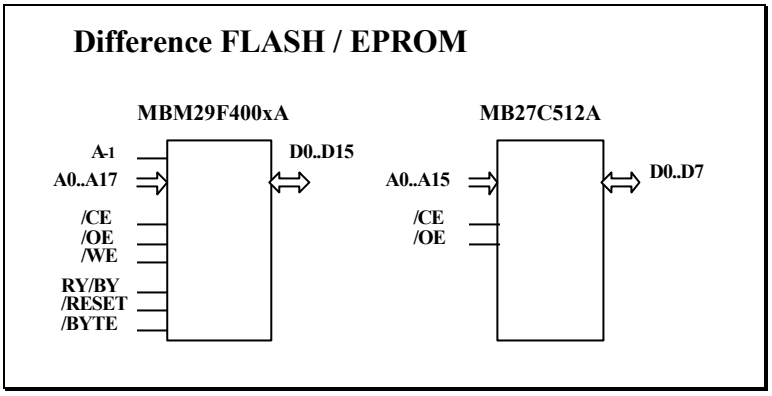
Flash Memories are the ideal code storage memory for microcontroller designs,
Mainly due to the fact, that the program content can be simply modified by the controller itself without having to physically open the system and replace a component.
Thus, the nightmare of errors in the firmware has lost its horror.

The following article intends to give some simple but important hints on hardware and software issues related to using flash memories together with microcontrollers in embedded applications.
It is based on a practical example of using the MBM29F400TA/400BA or MBM29F800T/MBM29F800B in 16-bit mode, but can also be easily adapted for other devices and configurations. For further information please also refer to the data-sheet.

Background

FLASH memory can be directly connected to any microcotroller bus in a similar manner to ROM or EPROM devices. Flash memories can also be programmed on dedicated programmers similar to EPROMs, but the real advantage is, that they can be re-programmed in-circuit under the appropriate software control.

One could assume that a flash would require some additional and special control signals, but that is not really the case. (At least with the FUJITSU flash memories which require only a single supply voltage). Actually, compared with a standard EPROM, there exist some special signals like /WE, RY/BY, /RESET, and BYTE but their function is quite trivial:



The /BYTE input is used to configure the device to operate in either 8-bit or 16-bit wide mode.
If it is configured as an 8-bit memory, the additional address line "A_{-1}" is needed to address the complete memory area. As mentioned above, for this application note we assume the 16-bit mode, so "A_{-1}" is not used.
"RY/BY" is an output status signal which could be used to check if the flash memory is doing something special ("embedded algorithm operations"), but since this status information can also be determined by software, it's not really necessary to connect this signal to a dedicated controller input, so we assume not used.

"/WE" is a simple write input and used to enable writing data to the flash memory.
But, do not assume that writing to a flash memory location would program this location with the written data. This would be to simple and more importantly to dangerous. A "corrupted" program could then overwrite and destroy the memory contents if it writes something into the flash memory space. To actually program a flash memory location, a "secret" command has to be sent to the flash devices internal state machine to put it into the "embedded command" mode. In this mode, you can then execute a write operation or one of the other embedded commands, such as erasing.

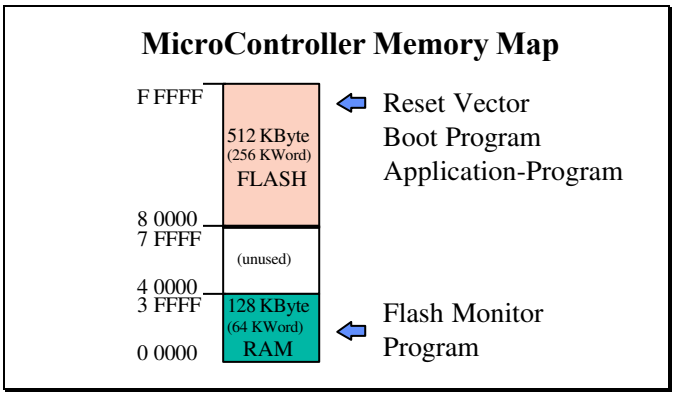
"/RESET" could be used to get the flash memory out of the "embedded command" mode, but as above, this can be done by software as well, so this input could be simply tied to Vcc.
In fact, the /RESET input could be useful in a different aspect mentioned later.

Programming Philosophy

So far it should be clear, that a flash memory is a quite simple device and that it is possible to perform in-circuit programming using some "secret" commands.
To make things a little bit more complicated, we have to pay attention to some futher considerations:

1.) First of all, it should be noted, that if the flash memory is the only program memory in a system, the flash memory must contain some instruction code, so the system is able to "boot-up" at power-on or reset. Thus the flash device has to be programmed once before it is soldered into the system.
(See the example Microcontroller Memory Map. Note that in this example we assume controllers like Fujitsu´s, which read its reset-vector from the upper or top memory area. There are other architectures, which would require the FLASH to be at the bottom of the memory area.)

2.) Before re-programming the flash memory, it has to be either erased completely, or just certain memory areas of the flash, called "sectors". This would be done by issuing an embedded erase command. (Please refer to the data sheet on the flash memory sector areas.)
The problem is, that while a erase procedure is in progress, the complete flash memory can not be accessed. So even if just a sector is erased, the controller software can not run from a different sector in that flash memory.
Thus, before erasing the flash, a so called "Flash Monitor" program has to be installed in a different memory area, for example by copying some code from the Flash to the RAM area.
When the controller is executing code from that RAM area, it can initiate the erase operation, and make sure that new program data is written into the flash memory.



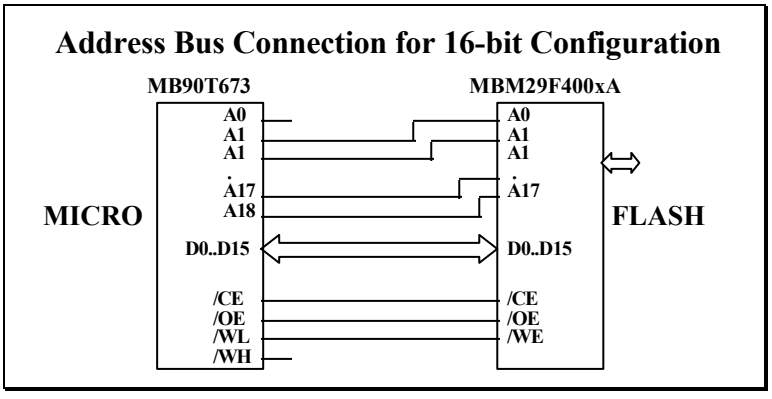
Now, what would happen if the power-supply fails during that process.
Well, if the complete chip erase was executed, basically bad luck.
The flash program memory would be empty or inconsistent and the system is out of use.
If just an individual sector was erased, and it was not the sector which contains the reset-vector, boot program and the install-routine for the Flash Monitor, the system could be re-programmed again.

To make sure such an important "boot sector" can never be erased, it is indeed possible to "erase-protected" each individual sector. Note, that this protection can only be done by dedicated programming equipment and not by software control.
A drawback of this protection is of course, that the flash program can not be re-programmed completely anymore. The application software has to be structured into a fixed part which will never change, and a variable part which could be updated.
To be able to keep this fixed part as small as possible, the flash memory have a couple a smaller sectors in the "boot area", so the required protection area can be specified in rather small area steps.
Note also, that due to the different processor architectures, there exist two type of flash devices.
One, where these smaller sectors are located at the top of the memory area, and the others where they are located at the bottom.

At this point I would also like to mention a nice trick using the /RESET input.
If the microcontroller bus is accessible by an external connector, it would be possible to temporarily connect an external memory board which contains a program memory with some boot-code, which is mapped to the same flash memory address space.
Usually this would lead to a bus-conflict, if two memory devices are selected within the same memory area, but if the flash /RESET is activated, then the on-board flash memory is disabled.
Now, the external program could install a "Flash Monitor" into ram. Then the external memory board could be disconnected or made inactive, the /RESET could be released again, and the flash memory can be re-programmed with the updated code, so the /RESET input is effectively used as a second chip select input.
This way, even an empty flash memory could be programmed inside a system for the first time.

Bit Assignment, Even/Odd Address, Byte/Word Accesses

To understand the software procedures which can activate the "embedded commands", there are some more hardware related issues to discuss, since they have some impact on the software.
The main reason for this is because we have chosen the 16-bit wide example.
The typical connection scheme between a Fujitsu microcontroller and a 16-bit wide flash memory is shown in the following figure. Other Architectures might look slightly different.



The important thing is, that the address line A1 of the controller is connected to A0 of the flash memory. The controller would need its A0 line only to address single bytes in an external memory, but since we configured our system to be 16-bit wide, the smallest unit the microcontroller can read is a 16-bit word. Note, that also the controller has to be configured such that it knows that the flash memory area is 16-bit wide.
This ensures that even if the controller software reads a byte from the flash area, the controller hardware will read a 16-bit word and multiplex the higher data bits D8..D15 onto D0..D7 internally if the access was to an odd memory location.)
This ensures also, that if the controller would read a 16-bit word from an odd-address, the micro hardware has to read two 16-bit words and assemble the data by using the hi-byte of the 1st and the lo-byte of the 2nd access.

If the flash memory would not be a flash, but a 16-bit wide RAM, the RAM would have two write inputs, so the micro could write individual bytes into the RAM (/WL and /WH).
But the flash is not such a RAM and has just one write input which is connected to WL, so we can only write 16-bit words into the flash memory.

Coming back to the “secret embedded commands”.
As said before, writing data to the flash memory does not program a flash memory location,
the write cycle is simply ignored.
But if you know the “secret”, you can write one of the special codes into the relevant memory locations in the right order (like a pass-word) and you will get into the embedded command mode.

Now, the above implications on AddressBit Assignement and 16-Bit Bus Mode imply some additional rules to be followed, otherwise the “password” will not work. These are:

a) The software must be written in a manner, such that all read/write accesses to the flash memory (for embedded commands) are 16-bit wide and reference only even microcontroller-addresses. Otherwise there is a risk that the micro-hardware generates two-cycle access. Of course this might require special care when using a C-Compiler such that the C-Compiler does not generate strange executable code.

b) The special “secret” memory locations to write to, are not the ones mentioned in the data-sheet. This means physically yes, but due to the A1 => A0 bit assignments, the address values used by the controller software are actually the mentioned ones shifted left by one bit.

Example Program

After all these hardware considerations its time to look at the software.
An example program was developed which is able to demonstrate the embedded functions.
The program is structured into two source code modules. One provides the “embedded command functions”, the other is a user interface or monitor type program, which calls these functions to display some information about the flash memory (Vendor-, Device-Code), and which allows to do some basic thinks like chip-erase, individual sector erase, program and display memory locations.
To save space, we have listed just the first source code module in this article.

Edmund Bendels

```
/*+------------------------------------------------------------------+*/
/*¦                          F U J I T S U                           ¦*/
/*¦              M i k r o e l e k t r o n i k   G m b H             ¦*/
/*¦                                                                  ¦*/
/*¦  Filename:  FLA.C                                               ¦*/
/*¦  Function:  Basic Routines for FLASH operations                  ¦*/
/*¦             (Tested with MBM29F400TA and MBM29F800B)             ¦*/
/*¦                                                                  ¦*/
/*¦  Series:    MB90670/5                                            ¦*/
/*¦  Version:   V01.00                                              ¦*/
/*¦  Design:    E. Bendels                                           ¦*/
/*¦  Change:    <name>  <date>                                       ¦*/
/*+------------------------------------------------------------------+*/

#include <typedefs.h>                   /* usefull type definitions */
#include "fla.h"                        /* Flash Functions */

#define AutoSelCmd 0x9090
#define ProgramCmd 0xA0A0
#define SecEraCmd1 0x8080
#define SecEraCmd2 0x3030
#define ChpEraCmd2 0x1010
#define LoAdr5555 0xAAAA                /* Address 5555 is AAAA physically !
*/
#define LoAdrAAAA 0x5554                /* Address AAAA is 5554 physically !
*/
#define LoAdrxx00 0x0000                /* Address xx00 is xx00 physically !
*/
#define LoAdrxx01 0x0002                /* Address xx01 is xx02 physically !
*/
#define LoAdrxx02 0x0004                /* Address xx02 is xx04 physically !
*/
                                        /* since A1 of Micro => A0 of Flash */
                /* Note: OddAddress would lead to a 2 cycle Access ! */

/*-------------------------------------------------*/
/*-------------------------------------------------*/
void GenerateSequence(long FAdr, WORD Cmd)
{
  WORD __far *pAAAA;                    /* some Pointers to Flash Memory */
  WORD __far *p5555;                    /* some Pointers to Flash Memory */

        p5555 = (WORD __far*) (( FAdr & 0xFFFE0000) | LoAdr5555);
        pAAAA = (WORD __far*) (( FAdr & 0xFFFE0000) | LoAdrAAAA);

        *p5555 = 0xAAAA;                    /* Data = AA */
        *pAAAA = 0x5555;                    /* Data = 55 */
        *p5555 = Cmd;                       /* Data=Command Byte (Word)*/
}

/*-------------------------------------------------*/
WORD FL_AutoSelect(long FAdr, WORD *MCode, WORD *DCode)
{
  WORD __far *pAdr;

        GenerateSequence( FAdr, AutoSelCmd);
```

```
        pAdr = (WORD __far*) (( FAdr & 0xFFFE0000) | LoAdrxx00);
        *MCode = *pAdr;                          /* read Device Code */

        pAdr = (WORD __far*) (( FAdr & 0xFFFE0000) | LoAdrxx01);
        *DCode = *pAdr;                         /* read Manufacture Code */

        FL_FastReset(FAdr);                     /* Terminate AutoMode */
        return 1;
}

/*-------------------------------------------------*/
WORD FL_GetSectProt(long FAdr)
{
  WORD __far *pAdr;
  WORD Flag;

        GenerateSequence( FAdr, AutoSelCmd);
        pAdr = (WORD __far*) (( FAdr & 0xFFFE0000) | LoAdrxx02);
        Flag = *pAdr;                           /* Protect Bit Status */
        FL_FastReset(FAdr);                     /* Terminate AutoMode */
        return Flag;
}

/*-------------------------------------------------*/
void FL_FastReset(long FAddr)
{
  WORD __far *pAdr;

        pAdr = FAddr & 0xFFFE0000;
        *pAdr = 0xF0F0;
}

/*-------------------------------------------------*/
WORD FL_ReadWord(long FAddr)
{
  WORD __far *pAdr;
  WORD Dat;

        pAdr =  (WORD __far*) FAddr;
        Dat = *pAdr;
        return Dat;
}

/*-------------------------------------------------*/
WORD FL_WriteWord(long FAdr, WORD Data)
{
  WORD __far *pAdr;
  WORD TDat;

        pAdr =  (WORD __far*) FAdr;
        GenerateSequence( FAdr, ProgramCmd);
        *pAdr = Data;
        do {
          TDat = *pAdr;
          if (TDat==Data) return 1;
          else if ((TDat&0x0028)==0x0028) return 0;     /* Failure Case */
        } while (1);
```

```
}

/*-----------------------------------------------*/
WORD FL_SectorErase(long FAdr)
{
  WORD __far *pAAAA;                    /* some Pointers to Flash Memory */
  WORD __far *p5555;                    /* some Pointers to Flash Memory */
  WORD __far *pAdr;

  WORD TDat;

        pAdr =  (WORD __far*) FAdr;
        p5555 = (WORD __far*) (( FAdr & 0xFFFE0000) | LoAdr5555);
        pAAAA = (WORD __far*) (( FAdr & 0xFFFE0000) | LoAdrAAAA);

        *p5555 = 0xAAAA;                    /* Data = AA */
        *pAAAA = 0x5555;                    /* Data = 55 */
        *p5555 = SecEraCmd1;                /* Erase Command 1 */

        *p5555 = 0xAAAA;                    /* Data = AA */
        *pAAAA = 0x5555;                    /* Data = 55 */
        *pAdr  = SecEraCmd2;                /* Erase Command 2 */

        do {
          TDat = *pAdr;
          if (TDat&0x0080) return 1;            /* if finnished */
          else if ((TDat&0x0028)==0x0028) return 0;     /* Failure Case */
        } while (1);
}

/*-----------------------------------------------*/
WORD FL_ChipErase(long FAdr)
{

  WORD __far *pAdr;
  WORD TDat;

        FAdr &= 0xFFFE0000;                     /* Make sure EvenAddress */
        pAdr =  (WORD __far*) FAdr;

        GenerateSequence( FAdr, SecEraCmd1);
        GenerateSequence( FAdr, ChpEraCmd2);

        do {
          TDat = *pAdr;
          if (TDat&0x0080) return 1;            /* if finnished */
          else if ((TDat&0x0028)==0x0028) return 0;     /* Failure Case */
        } while (1);
}
```