Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Expert In-Circuit FLASH Programmer for SHARC® Processors

Contributed by Mitesh Moonat and Harshit Gaharwar *Rev 1 – August 6, 2012*

Introduction

SHARC® processors can be booted from an externally connected non-volatile memory. While the non-volatile memory used for booting can either be a PROM or a FLASH, the latter is most commonly used in recent designs. Both serial and parallel FLASH devices are usually hard-soldered on the board interfaced to the processor and can be programmed by a separate piece of software running on the processor itself. This method of programming the FLASH devices is often referred to as *"In Circuit FLASH programming"*.

This application note discusses how serial and parallel FLASH devices can be interfaced with SHARC processors: ADSP-2126x, ADSP-2136x, ADSP-2137x, ADSP-214xx processors.

For the parallel FLASH interface, this EE-Note does not cover processors with a *Parallel Port* (i.e., ADSP-2126x and ADSP-21363/4/5/6 processors), but only those with an *External Port* (ADSP-21367/8/9, ADSP-2137x, ADSP-214xx processors).

Furthermore, the associated `.ZIP` file provides simple and modular C code for interfacing both serial and parallel FLASH devices to the different SHARC processors. Along with this, the EE-Note also provides guidelines on how the provided C code can be modified for a different processor and/or a different FLASH device. In addition to using this code for programming the boot image to the FLASH, it can also be used for communication with the FLASH device for purposes other than the before mentioned boot process.

FLASH Devices Types

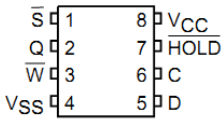
There are two most commonly used types of FLASH devices: serial and parallel.

Serial FLASH

As its name suggests, serial FLASH devices can be accessed over a serial interface. The most common serial interface standard used to communicate with serial FLASH devices is the *Serial Peripheral Interface (SPI)*. The most striking feature of this type of FLASH device is that all the communication, including memory addressing, data input and output, and control related commands is carried over serially with the help of a very few wires/signals, which in turn translates into very small package sizes. Generally, for single I/O SPI mode, this interface requires four signals, which include one clock signal to clock the data, two data signals - Master Out Slave In (MOSI) and Master In Slave out (MISO), and one chip select or slave select signal. However, in recent days,

there are “Multi I/O” SPI FALSH devices also available, such as dual (two bit data bus) or quad (four bit data bus) I/O mode, which can double/quadruple the access speed of the FLASH device.

This application note focuses on the single I/O mode only. Dual and Quad SPI modes are out of the scope for this revision of the document. Figure 1 shows the pin details of the M25P16 serial FLASH device.

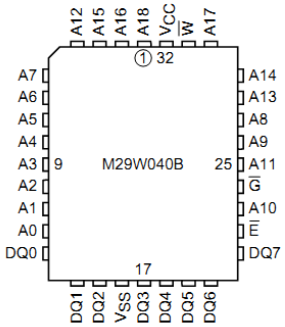


Signal name	Function	Direction
C	Serial Clock	Input
D	Serial Data input	Input
Q	Serial Data output	Output
$\overline{S}$	Chip Select	Input
$\overline{W}$	Write Protect	Input
$\overline{HOLD}$	Hold	Input
V_{CC}	Supply voltage	
V_{SS}	Ground	

Figure 1. Pin details of the M25P16 serial FLASH.

Parallel FLASH

Parallel FLASH devices are accessed over a parallel interface with the help of separate address, data and control lines. The parallel interface provides a relatively higher throughput at the cost of a larger pin count and bigger package size. The actual pin count depends upon whether the FLASH is byte (8-bit data) or word (16 bit data) accessible. The way a parallel FLASH can be accessed depends upon whether the device is CFI (Common FLASH Interface) compatible. CFI is an open standard developed by AMD, Intel, Sharp and Fujitsu and approved by the non-volatile memory subcommittee of JEDEC. The idea behind this concept is the interchangeability of current and future flash memory devices offered by different vendors. For a CFI compatible FLASH device, the developer can use a single driver for different flash products by reading identification information out of the flash chip itself. However, the CFI compatible command sets are out of scope for this revision of the document. Figure 2 shows pin details of the M29W040B parallel FLASH device.



A0-A18	Address Inputs
DQ0-DQ7	Data Inputs/Outputs
$\overline{E}$	Chip Enable
$\overline{G}$	Output Enable
$\overline{W}$	Write Enable
V_{CC}	Supply Voltage
V_{SS}	Ground

Figure 2. Pin detail of the M29W040B parallel FLASH.

Serial vs. Parallel FLASH

Using parallel FLASH devices result in a comparatively higher throughput. However, there are various other features of the serial FLASH devices, which make it a more popular and cost effective non-volatile memory solution, especially for the fast growing handheld and portable battery powered computer, personal communications, medical and industrial markets. Some of these features are:

1. Smaller footprint, reduced package and system costs.
2. Low power consumption and possibility of lower operating voltage.
3. Ease of programming.

Programmability of the FLASH devices

Reading from a NOR FLASH device, especially a parallel FLASH device, is very similar to reading from a Random Access Memory (RAM). However, programming the FLASH device is significantly different than writing data to RAM. The programming procedure can only change from a logical one to a zero. Bits that are already zero are left unchanged. Thus, to write a data in to the FLASH, it must be preceded by an erase operation which changes the corresponding bits back to one. Erasure usually takes place block wise where typical block sizes are 64, 128, or 256 KB.

Serial FLASH Programming

The following sections go into details of the hardware and software required to interface and program a serial/SPI FLASH device to SHARC processors. But first of all, let's examine some of the features of the commonly used devices.

Common Serial FLASH features

- **Typical memory size:** the serial FLASH devices are available in sizes starting as low as 512 Kbits to as high as 256 Mbits. Commonly used sizes are 2/4/8/16 Mbits.
- **Supply voltage:** typical supply voltages are 3.3 V and 1.8 V.
- **Speed:** the serial FLASH devices available these days can support SPI clock speeds of as high as 108 MHz. The typical speeds are 50-80 MHz.
- **Data bus width:** in addition to the traditional single data bus width, FLASH devices also support dual and quad I/O modes with 2x and 4x bus widths respectively. Using these modes, the SPI FLASH devices can now be accessed at double/four times the speed supported by earlier FLASH devices.

Serial FLASH Internal Memory Architecture

To be able to program a serial FLASH device, it is very important to understand its internal memory structure. Figure 3 shows a simple diagram of the internal memory architecture of a typical serial FLASH memory.

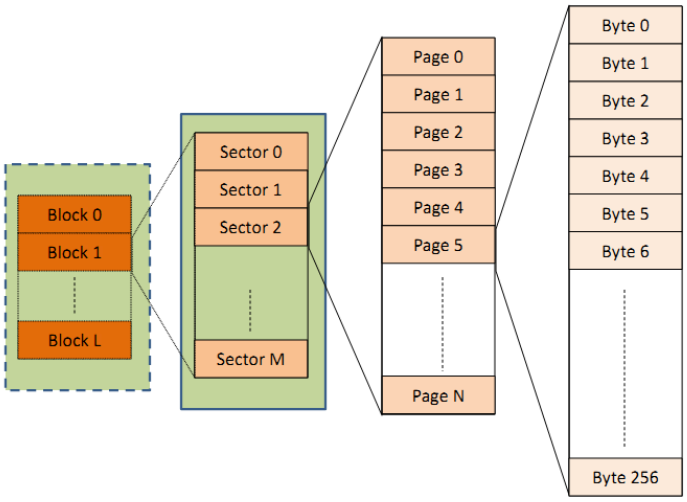


Figure 3. Typical serial FLASH memory architecture.

The most basic unit of the internal memory architecture is a byte. A fixed number of bytes (256) are grouped to form a page. A certain number of pages combine to form a sector. While the number of pages in a sector (N) may vary from one FLASH device to another, the typical values are 256 (64 KByte) and 16 (4 KByte). Generally, the size of two FLASH devices may differ either because of the number of sectors, or because of the number of pages per sector, or both.

For instance, M25P16 (16 Mbit) and M25P20 (2 Mbit) have the same number of pages per sector (256), i.e., same sector size (64 KByte), but different number of sectors (32 and 4 respectively). While the FLASH device SST25LF020A (2 Mbit) has less number of pages per sector (16), i.e. 4 KByte sector size. What it also means is that two serial FLASH devices might have the same total size but still a different internal memory architecture. Each page can be individually programmed (bits are programmed from "1" to "0"). The device is either "Sector" or "Bulk" erasable (bits are erased from "0" to "1") but not "Page" erasable.

In some serial FLASH devices, the sectors are further grouped into a bigger memory structure called a "Block", as shown by the dotted line in Figure 3. For example, the W25X10BV/20BV/40BV FLASH devices, contain a group of 16 sectors, which in turn form a 64 KByte block. W25X10BV devices have 2 blocks, while the W25X20BV devices have 4 blocks, and the W25X40BV devices 8 such blocks. These devices are all sector erasable (4 KByte) and block erasable (64 KByte). The example codes provided with this application note however, use only sector erase operations.

Various SPI FLASH commands

The host communicates with the SPI FLASH device with the help of various commands/instructions sent over the SPI interface. All instructions, addresses and data are shifted in and out of the device, most significant bit first. Serial Data Input (D) is sampled on the first rising edge of Serial Clock (C) after Chip Select (S) is driven Low. Then, the one-byte instruction code must be shifted into the device, most significant bit first, on Serial Data Input (D), each bit being latched on the rising edges of Serial Clock (C). Every instruction sequence starts with a one byte instruction code. Depending on the instruction, this might be followed by address bytes, or by data bytes, or by both or none. The following sections provide an overview of some of the common instructions. For more detailed information on the various commands supported by the FLASH device, refer to the corresponding datasheet.

Write Enable (WREN)

This instruction can be used to set the Write Enable Latch (`WEL`) bit of the status register. It is required prior to every program/erase/write status instruction.

Read Identification (RDID)

This instruction is used to read the unique manufacturer ID (1 byte) and device ID (2 bytes) information of the FLASH device. JEDEC READ ID (`0x9F`) instruction is supported by most serial FLASH devices, and thus can be used for "Auto Device Detection".

Read Status Register (RDSR)

This instruction allows the status register to be read. It provides status of the Write Enable Latch (`WEL`) and Write In Progress (`WIP`) bits.

Read Data Bytes (READ)

This command is used to read the data byte-by-byte from the serial FLASH device.

Page Program (PP)

The Page Program (`PP`) instruction allows bytes to be programmed in the memory (changing bits from "1" to "0"). A maximum of 256 Bytes can be programmed at a time with a single `PP` instruction. It should be preceded by a `WREN` instruction. `WIP` bit of the status register can be read to check the progress of this operation.

Sector Erase (SE)

This instruction can be used to erase (set to "1" – `0xFF`) all bits inside the chosen sector. It should be preceded by a `WREN` instruction. `WIP` bit of the status register can be read to check the progress of this operation.

Interfacing and Programming Serial FLASH Devices with SHARC processors

Hardware

Figure 4 shows the hardware connections required to interface a serial FLASH device to a SHARC processor. A serial FLASH device is usually an 8-pin IC. Four of these pins (clock, data input, data output, and chip select) are used for SPI communication. These signals can be connected to the corresponding SPI related pins on the SHARC side, which can be either dedicated pins (ADSP-2126x and ADSP-21362/3/4/5/6 processors) or DPI (Digital Peripheral Interface) pins (as for ADSP-21367/8/9, ADSP-2137x, and ADSP-214xx processors).

Figure 4 shows the default DPI pins (`DPI1, DPI2, DPI3, DPI5`) routed to the SPI port used for master booting. The Write Protect (`/WP`) pin is used to protect the SPI FLASH device against writes and it is pulled inactive (`HIGH`) permanently in this particular example with a 10 KOhms resistor, as it's not used. The Hold (`HOLD`) signal can be used to pause serial communication with the device without deselecting it, and it is pulled inactive (`HIGH`) permanently in this example with a 10 KOhms resistor, as it's not used.

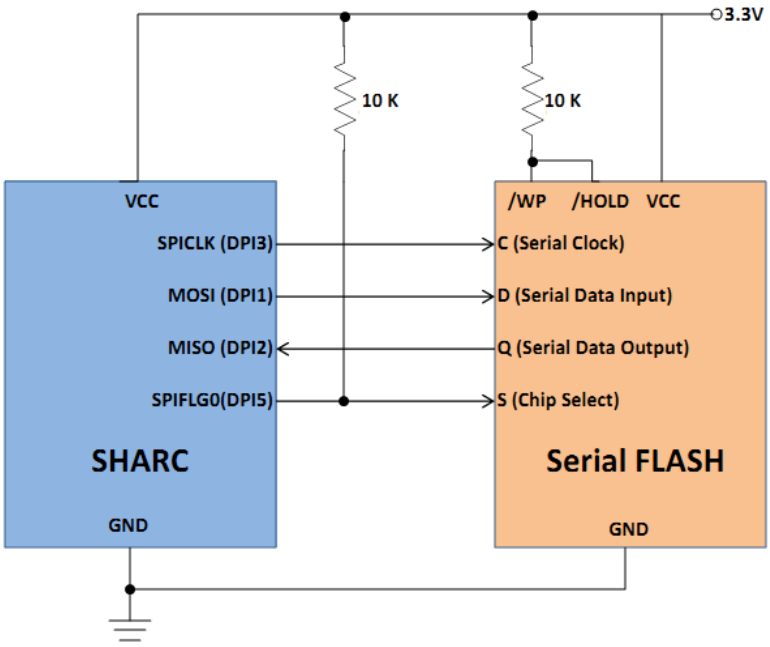


Figure 4. Serial FLASH SHARC interface.

Software

The software required to read/write/erase the FLASH device involves integration of three different categories of various functions:

1. *System Initialization functions* – these functions are one time initialization functions called at the beginning of the code. These are used to initialize PLL, DDR2/SDRAM controller, SRU, SPI baud rate, among other configuration settings. These are *processor dependent* but *FLASH independent* functions and thus need to be modified only when using a different processor.

2. *SPI related functions* – these functions are low level drivers written for communicating with FLASH device over SPI protocol. For example, one of such functions can be "Send N 8 bit words over MOSI". These are *processor dependent* but *FLASH independent* functions and thus need to be modified only when using a different processor.

3. *Serial FLASH related functions* – these functions use a combination of the low level SPI functions mentioned above and other functions of the same category in a specific sequence (as provided in the serial FLASH device data sheet) to communicate with the FLASH device. These are *processor independent* but *FLASH dependent* functions and thus need to be modified only when using a different FLASH device. Having said that, almost all serial FLASH devices use the same sequence/flow with the exception of the command values, which might differ. Thus, the only required change for a different FLASH device would usually be the modification of the command names.

Reading Data from Serial FLASH

Reading from FLASH involves only SPI functions. Figure 5 shows the software flow required to read N bytes from the FLASH from an arbitrary byte offset. As we can see, it involves only basic SPI functions (highlighted in **blue**).

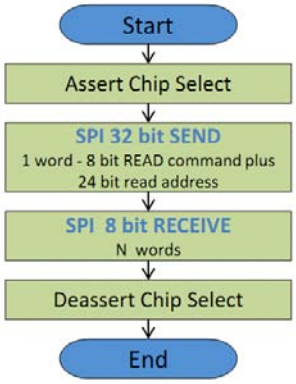


Figure 5. Reading N Bytes from serial FLASH.

Erasing Data from Serial FLASH

This process involves converting the memory location values from "0's" to "1's". An erase is always required before trying to program a data into the FLASH device. Figure 6 shows the software flow required to erase a serial FLASH sector. It involves both SPI and other Serial FLASH related functions (highlighted in **red**).

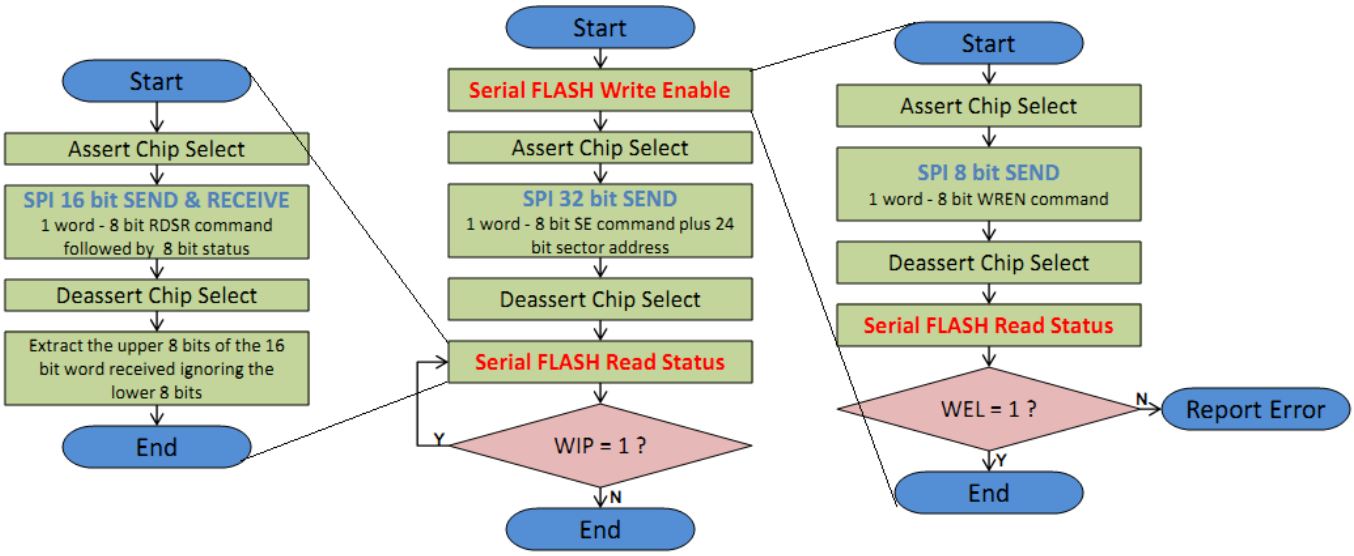


Figure 6. Erasing a serial FLASH sector.

Writing Data to Serial FLASH

Writing/Programming converts "1's" to "0's". Before programming data to a location of the FLASH device, it must be erased using sector erase command, as discussed earlier. Figure 7 shows the software flow required for writing a block of data of arbitrary length to an arbitrary serial FLASH memory region.

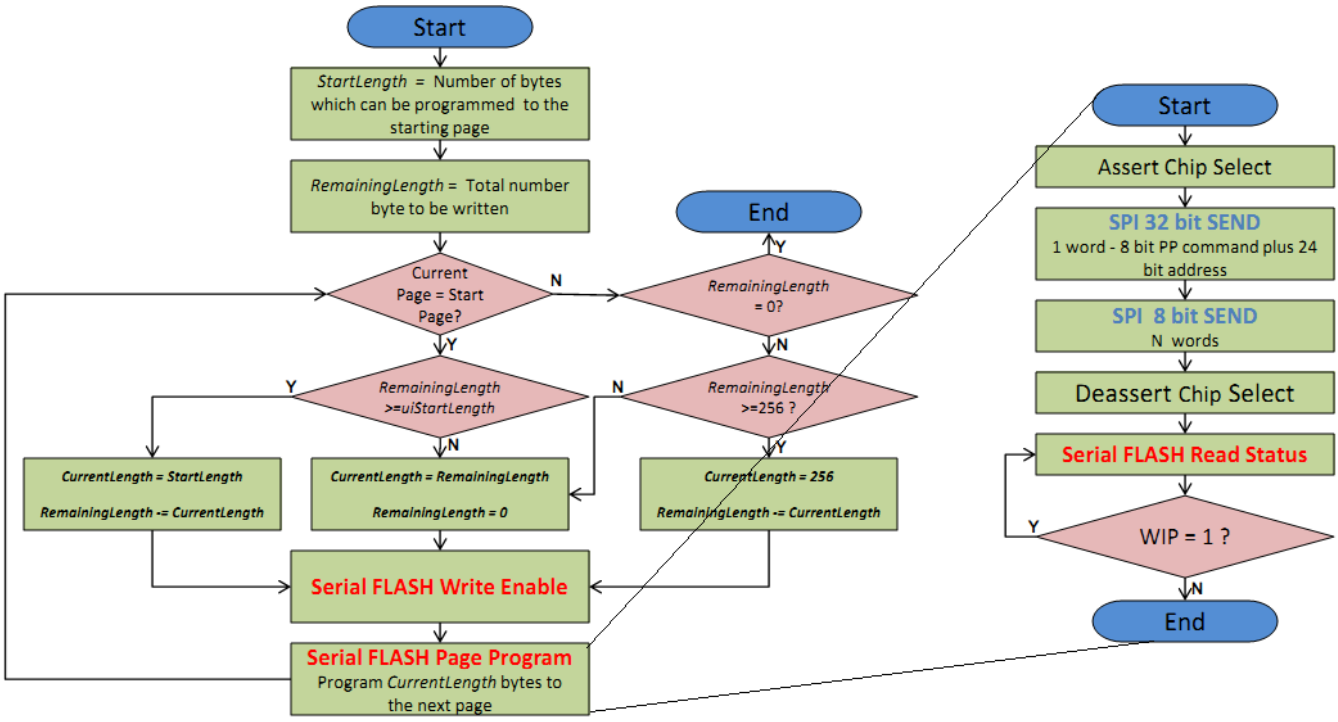


Figure 7. Writing bytes to serial FLASH.

Programming an LDR file into the FLASH

This section discusses the framework that makes the calls to various functions discussed earlier to program the loader file (`LDR`) into the FLASH. Figure 8 shows the flow of the main function.

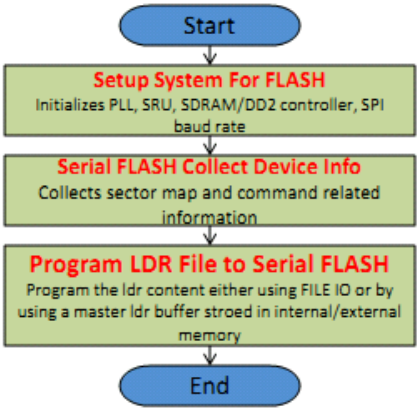


Figure 8. Main function for programming LDR file to serial FLASH

The first function it uses sets up the system (PLL, DDR2/SDRAM controller, SRU, SPI baud rate, etc.) for the SHARC-FLASH interface. Then, the *"Serial FLASH Collect Device Info"* function is called, as shown in Figure 9.

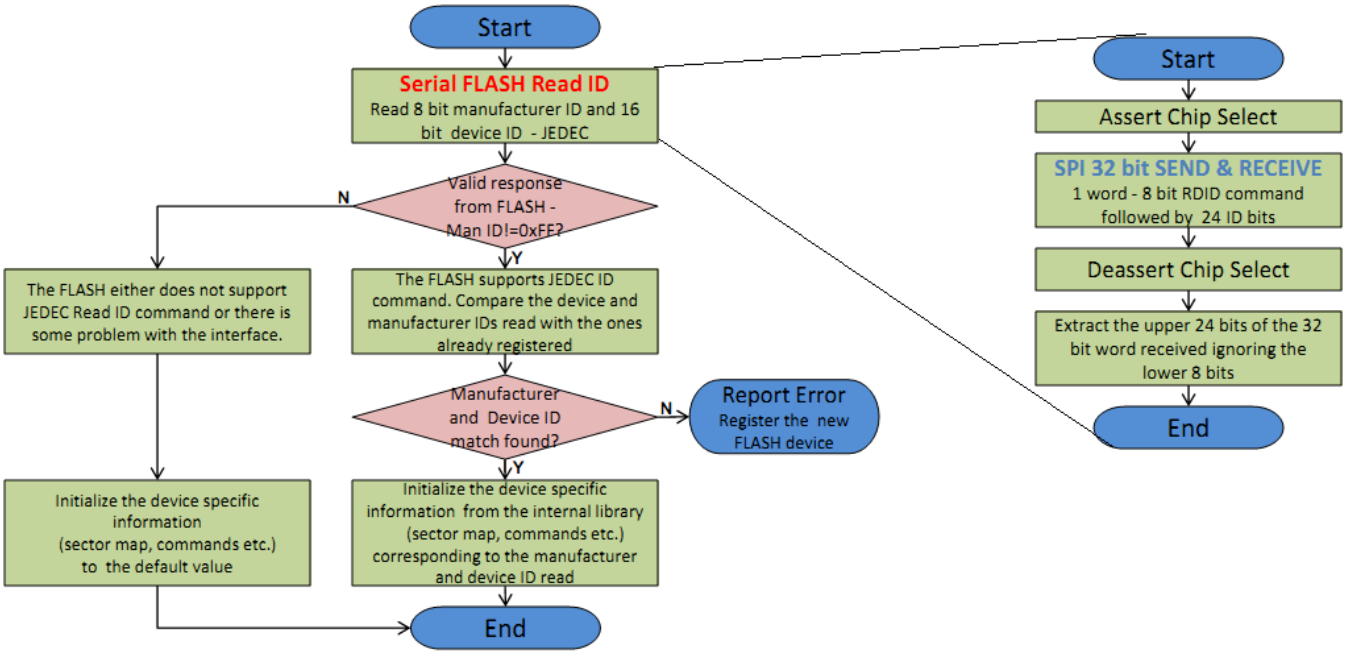


Figure 9. Serial FLASH collect device info

This function performs an *"Auto Device Detection"* to initialize various device specific parameters, such as device manufacturer's name, device name, size, sector map info, command set, etc. This is achieved by sending the JEDEC Read ID (`0x9F`) command to the FLASH device, which is supported by most FLASH devices. If the FLASH device supports this command, it responds with a valid manufacturer and device ID. The code then tries to match the read manufacturer ID with the ones already registered inside the code. If a match for both manufacturer and device ID occurs, it means that the device specific information for that FLASH device is already available in the existing library and thus it will be initialized accordingly. Else, these parameters will be initialized with default values (defined statically before loading the code). Hence, the following two possibilities exist:

- The device connected does support the JEDEC Read ID command, but is not registered in the code. In such a case, the user will have to register the FLASH ID by entering the device specific information as specified in the data sheet.
- The device connected does not support the JEDEC Read ID command or there is some problem with the hardware interface. The latter case should be debugged separately. For the former one, such device cannot be auto-detected. Thus, for such devices, one will have to manually make sure that before building the code the default device specific settings are changed as per the connected device.

Figure 10 shows the flow for programming an `LDR` to a serial FLASH device.

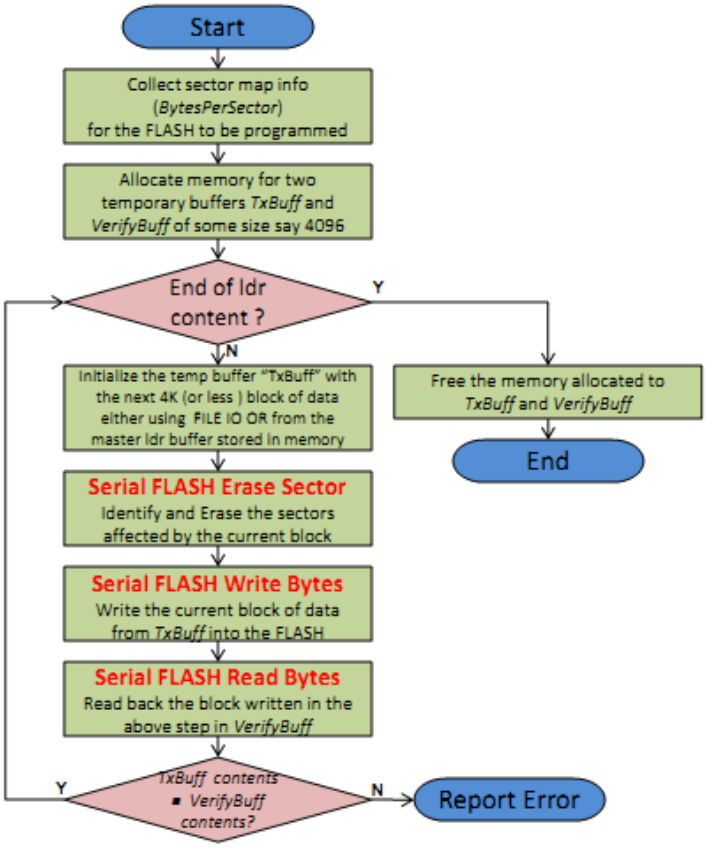


Figure 10. Programming the LDR file contents to serial FLASH

How to use the example code

The code example *"SHARC_Expert_Serial_FLASH_Programmer"* provided with the associated `.ZIP` file has been developed based on the above described software architecture in such a way that the same project can be used for different SHARC processors and different SPI FLASH devices with minimal modifications. The file *"SystemInit.C"* contains system initialization functions, *"SPIInit.C"* contains SPI related functions, and the file "Serial_FLASH.C" contains FLASH related functions. To be able to build and run the provided example code for a serial FLASH device connected to a SHARC processor, the following items need to be taken into consideration:

- **Processor Specific Settings:** The example code supports all SHARC processors. The files *"SPIInit.c/h"* and *"Serial_FLASH.c/h"* are *processor independent*, thus no changes are required for these files. The file *"SystemInit.c/h"* contains all the *processor dependent* code such as PLL, DRAM controller, and Signal Routing Unit (SRU) initializations. The `CCLK` and `SDCLK/DDR2CLK` are by default programmed to their maximum values possible for the corresponding EZ-Kits. The SRU is programmed to use the DPI pins for SPI master booting. Therefore, these settings would need to be modified by the user according to the specific system requirements.

- **FLASH Specific Settings**: The example code supports the M25P16, M25P20, and W25X40BV serial FLASH devices. For programming a new FLASH device, the user would need to register the new device as per the following screenshot, which shows the registration entries for M25P16 and M25P20 devices.

```
void Serial_FLASH_Collect_Device_Info(SERIAL_FLASH_DEVICE_INFO* serial_flash_device_info)
{
    //Read device ID and manufaturer ID
    Serial_FLASH_Read_Identification(&(serial_flash_device_info->IDInfo));

    //Manufaturer name and Device name
    switch(serial_flash_device_info->IDInfo.cManID)
    {
        //Numonyx
        case 0x20:
                //Initialize the manufaturer name here
                strcpy(serial_flash_device_info->ManName,"Numonyx");

                //Compare the device ID to check if the device is M25P16 - 214xx EZ-Kits
                if(serial_flash_device_info->IDInfo.sDeviceID==0x2015)
                {
                    //Initialize the complete device name here
                    strcpy(serial_flash_device_info->DeviceName,"M25P16 - 16 Mbit, Low Voltage, Serial Flash Memory with 75 MHz SPI Bus Interface");

                    //Initialize the sector map info here
                    serial_flash_device_info->SectorInfo.uiBytesPerPage=        256;
                    serial_flash_device_info->SectorInfo.uiPagesPerSector=      256;
                    serial_flash_device_info->SectorInfo.uiTotalSectors=        32;
                    serial_flash_device_info->SectorInfo.uiTotalBytes=          2097152;
                }

                //Compare the device ID to check if the device is M25P20 - 21364/21369 EZ-Kits
                else if(serial_flash_device_info->IDInfo.sDeviceID==0x2012)
                {
                    //Initialize the complete device name here
                    strcpy(serial_flash_device_info->DeviceName,"M25P20 - 2 Mbit, Low Voltage, Serial Flash Memory with 75 MHz SPI Bus Interface");

                    //Initialize the sector map info here
                    serial_flash_device_info->SectorInfo.uiBytesPerPage=        256;
                    serial_flash_device_info->SectorInfo.uiPagesPerSector=      256;
                    serial_flash_device_info->SectorInfo.uiTotalSectors=        4;
                    serial_flash_device_info->SectorInfo.uiTotalBytes=          262144;

                }

                //Initialize the command set info here - common for same different FLASH devices from the same manufaturer
                serial_flash_device_info->CommandSet.WREN       =   0x06;           //Write Enable
                serial_flash_device_info->CommandSet.RDSR       =   0x05;           //Read Status Register
                serial_flash_device_info->CommandSet.READ       =   0x03;           //Read Data Bytes (f <= 20 MHz)
                serial_flash_device_info->CommandSet.PP         =   0x02;           //Page Program
                serial_flash_device_info->CommandSet.SE         =   0xD8;           //Sector Erase

                break;
```

Figure 11 Device Registration of the Serial FLASH Supporting JEDEC READ ID Command

Both devices have the same Manufacturer ID (`0x20`), and hence have the same command sets, while having different device IDs (`0x2015` and `0x2012`) and sector map information. So, to support a new FLASH device from the same manufacturer, the user will have to add another "else if" structure with four new entries: *BytesPerPage*, *PagesPerSector*, *TotalSectors*, *and TotalBytes*. However, to support a new FLASH device from another manufacturer, the user will have to add another "case" structure with FLASH command set entries (`WREN`, `WRDI`, `RDID`, `RDSR`, `WRSR`, `READ`, `FAST_READ`, `PP`, `SE`, and `BE`) in addition to the sector map information previously discussed.

Figure 12 shows the default entries for the AT25F2048 FLASH device along with the extra line of code *"#define DEFAULT_AT25F2048"*, which would need to be uncommented only when using the AT25F2048 device.

```
#define DEFAULT_AT25F2048
//#define DEFAULT_SST25LF020A


#ifdef DEFAULT_AT25F2048

    //Default definitions for AT25F2048
    #define WREN_DEFAULT                0x06    //Write Enable
    #define RDSR_DEFAULT                0x05   //Read Status Register
    #define READ_DEFAULT                0x03   //Read Data Bytes (f <= 20 MHz)
    #define PP_DEFAULT                  0x02   //Page Program
    #define SE_DEFAULT                  0x20   //Sector Erase                --> Different

    #define MANUFATURER_DEFAULT         "Atmel"
    #define DEVICE_NAME_DEFAULT         "2Mbit High Speed SPI Serial Flash Memory"

    #define BYTES_PER_PAGE_DEFAULT      256
    #define PAGES_PER_SECTOR_DEFAULT    256
    #define TOTAL_SECTORS_DEFAULT       4
    #define TOTAL_BYTES_DEFAULT         262144

#endif
```

Figure 12 Device Registration of the Serial FLASH NOT Supporting JEDEC READ ID Command

- **LDR Programming Method Selection:**
    - Using memory to store the LDR file contents
        - To store the complete LDR file contents in internal SRAM, uncomment the macro definition *"USE_MEMORY_STORAGE_FOR_LDR_PROGRAMMING"* and comment out the definitions *"USE_EXTENAL_MEMORY_FOR_LDR_STORAGE"* and *"USE_FILE_IO_FOR_LDR_PROGRAMMING"*. This is the preferred method but can only be used if the LDR file is small enough to fit into the processor's internal memory.
        - If internal memory is not enough, to store the complete LDR file contents in external SDAM/DDR2RAM/SRAM, uncomment the macro definition *"USE_MEMORY_STORAGE_FOR_LDR_PROGRAMMING"* and *"USE_EXTENAL_MEMORY_FOR_LDR_STORAGE"* and comment out the definition *"USE_FILE_IO_FOR_LDR_PROGRAMMING"*. This method is slower than the above method, but faster than the FILE I/O method described next. This method, however, cannot be used with processors with no external memory connectivity.
    - Using File I/O to store the LDR file contents (it does not apply for ADSP-2137x processors, because of the lack of sufficient internal memory resources for the file I/O library), uncomment the macro definition *"USE_FILE_IO_FOR_LDR_PROGRAMMING"* and comment out the definitions *"USE_MEMORY_STORAGE_FOR_LDR_PROGRAMMING"* and *"USE_EXTENAL_MEMORY_FOR_LDR_STORAGE"*. This is the slowest method of all the three, but is not limited by the size of the LDR file.
- Change the *"Processor"* in *"Project Options"* to the corresponding processor being used.

Parallel FLASH Programming

Common Parallel FLASH Features

- **Typical Memory Size:** The parallel FLASH devices are available in sizes of as low as 2 Mbits to as high as 2 Gbits. SHARC processors with 24 bit external address lines can support devices of up to 128 Mbits (16 Mbytes). Commonly used sizes are 4/8/16/32 Mbits.
- **Supply Voltage:** Typical supply voltages are 5, 3.3 and 1.8 Volts.
- **Access Time:** The read access time for parallel FLASH devices is usually between 45 to 110 ns. Typical values are between 70 to 90 ns, which equals 16 to 20 wait states for `DDR2CLK` speed of 225 MHz (ADSP-2146x) and 12 to 15 wait states for `SDCLK` speed of 166 MHz for SHARC Asynchronous Memory Interface (AMI) controllers.
- **Data Bus Width:** Parallel FLASH devices are available with 8/6/32 bit data bus widths. However, the commonly used bus widths are 8 and 16 bits. For runtime accesses, the ADSP-21367/8/9 and ADSP-2137x processors support bus widths of up to 32 bit, while the ADSP-2147x and ADSP-2148x processors support bus widths of up to 16 bit, and ADSP-2146x processors support 8 bit only. However, when it comes to booting, all these processors support 8 bit parallel FLASH boot mode only. Therefore, parallel FLASH devices connected to

SHARC processors are typically either 8 bit devices (e.g. AM29LV081B on ADSP-21369/21375 EZ-KIT boards), or 16 bit devices used in byte mode (e.g. M29W320EB used on ADSP-214xx EZ-KIT boards).

Parallel FLASH Internal Memory Architecture

Figure 13 shows typical FLASH internal memory architectures.

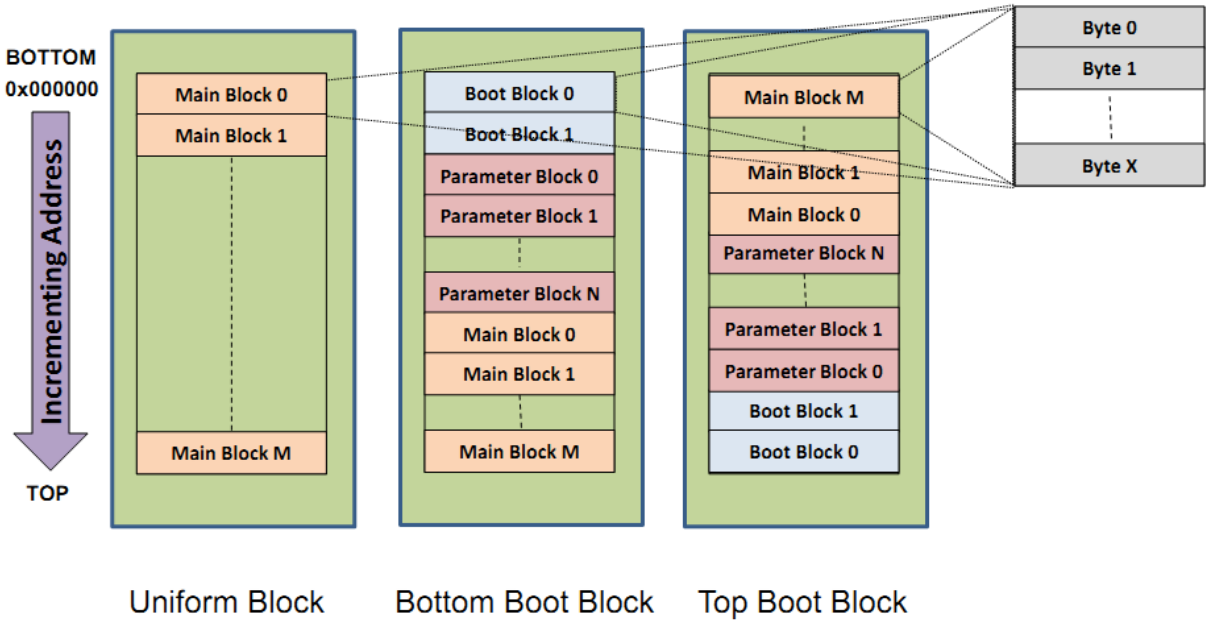


Figure 13. Typical parallel FLASH internal memory architectures

At a high level, there are typically two types of memory architectures in which parallel FLASH devices are available:

1. **Uniform Block Architecture** – All the memory blocks (sectors) are of the same size. Typical block sizes are 4/8/16/32/64 Kbytes. This type of architecture is used for applications that use all blocks for the same purpose (i.e.; a FLASH file system). Examples of such FLASH devices are AM29LV081B (8 Mbits), which has 16 uniform sectors each of size 64 Kbytes, and SST39LF040 (4 Mbits),which has 128 uniform sectors each of size 4 Kbytes.

2. **Boot Block Architecture** – The memory blocks are categorized as boot blocks, parameter blocks, and main blocks, all having different sizes. Some FLASH devices may as well have one smaller main block with the other main blocks. Many applications have no need to use this kind of architecture. However, some applications may make good use of this architecture. For such applications, the information that is stored in the FLASH device can be categorized into Boot Code, Application Code, User Parameters and User Data. Boot code is the code and data used for booting purposes. This is usually present in the "Boot Block(s)". This code is the heart of the application, and thus the FLASH devices provide a dedicated external pin, which can be used to protect the boot blocks from erase/write operations. The application code is the actual code, which is loaded by the boot code and which the user sees running. Since the application code does not need different blocks and it's only erased once, this is typically placed in the bigger main blocks. User

Parameters are the configuration parameters, which configure the system based on user requirements. Since these are small in size, they are usually stored in smaller parameter blocks. User Data is usually bigger in size that File Systems, Databases, etc., and hence, it's typically stored in bigger uniform blocks.

This memory architecture can be further categorized as *"Top Boot Block Architecture"* and *"Bottom Boot Block Architecture"*, depending upon whether the boot block is present at the bottom or top of the of the memory architecture. Example of such architectures is the M29W320EB (32 Mbit) FLASH device used on ADSP-214xx EZ-KIT boards, which is a bottom boot block device. It contains 8 parameter blocks of 8 Kbytes in size, out of which the first two are the boot blocks, and 63 main blocks of 64 Kbytes in size.

Various Parallel FLASH commands

Similar to serial FLASH devices, the host processor can communicate to parallel FLASH devices over a command interface. A command is nothing but a sequence of one or more Bus Write operations with a specific address and data sets. These long sequences help maximize data security. Following are few important commands required for basic read/write/erase communication with parallel FLASH devices.

Read/Reset

The Read/Reset command returns the memory to its Read mode. It also resets the errors in the Status register.

Auto Select

This command can be used to auto detect the FLASH device by reading the unique manufacturer and device codes.

Program

This command can be used to program an specific value to an address location of the memory array.

Block Erase

This command can be used to erase a list of one or more blocks. It sets all of the bits in the unprotected selected blocks to '1'. All previous data in the selected blocks is lost.

Interfacing and Programming Parallel FLASH Devices with SHARC processors

Hardware

Figure 14 shows the hardware connections between the SHARC external port and an 8 bit parallel FLASH device. While, Figure 15 shows the hardware connections for a 16 bit FLASH device interfaced in 8 bit mode.

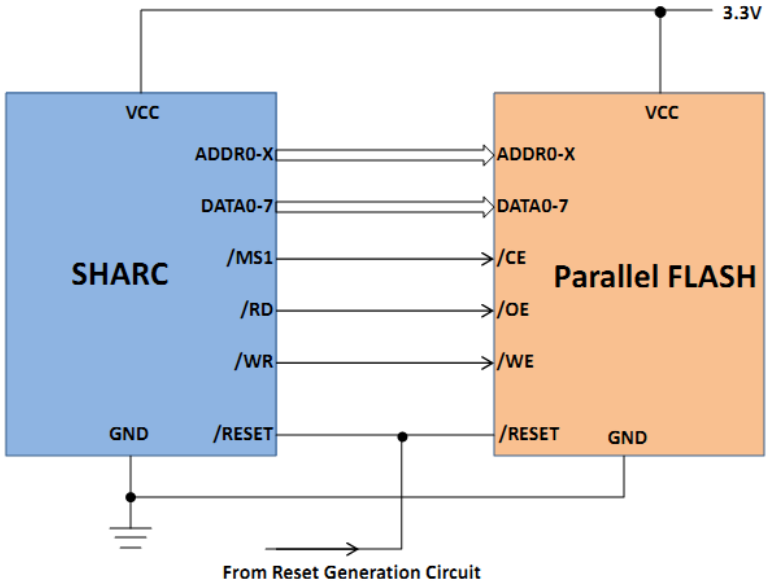


Figure 14. SHARC interface to an 8 bit parallel FLASH device

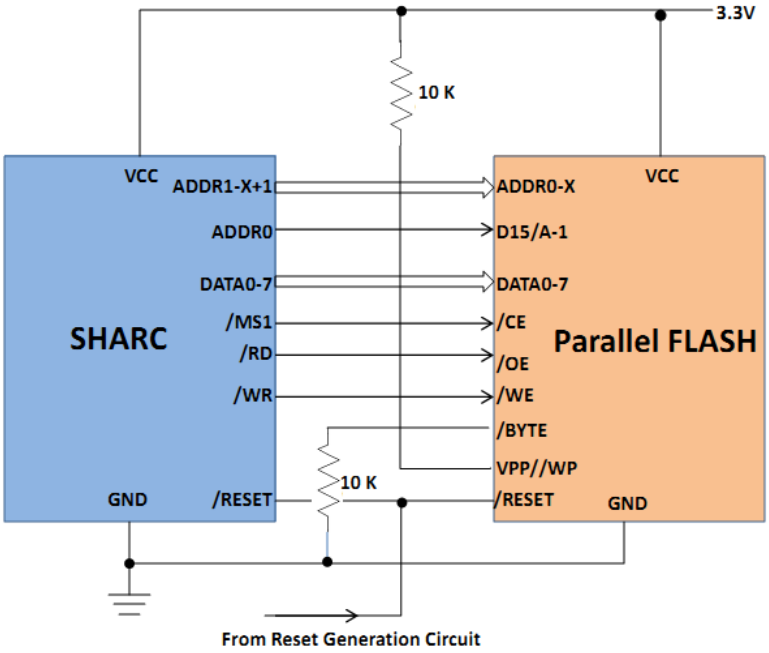


Figure 15. SHARC Interface to a 16 bit parallel FLASH device used in 8 bit mode

For an 8 bit FLASH device, `ADDR0` of the DSP is connected to `ADDR0` of the FLASH device. While, for a 16 bit FLASH device, `ADDR1` of the DSP is connected to the `ADDR0` of the FLASH device and `ADDR0` of the DSP is connected to the `D15/A-1` of the FLASH device. As shown in Figure 15, some FLASH devices may have an additional `VPP//WP` pin used to write protect the boot blocks.

Software

Similar to serial FLASH devices, the software required to be able to read/write/erase to the parallel FLASH devices involves integration of three different categories of various functions.

1. *System Initialization Functions* – these functions are one time initialization functions called at the beginning of the code. These are used to initialize PLL, DDR2/SDRAM controller, External Port, and AMI controller. These are *processor dependent* but *FLASH independent* functions and thus need to be modified only when using a different processor.

2. *External Port /AMI Related Functions* – these functions are low level drivers written for communicating with the FLASH device over the External Port. These are *processor dependent* but *FLASH independent* functions and thus need to be modified only when using a different processor.

3. *Parallel FLASH Related Functions* – these functions use a combination of the low level External Port functions mentioned above and other functions of the same category in a specific sequence (as provided in the FLASH device data sheet) to communicate with the FLASH device. These are *processor independent* but *FLASH dependent* functions and thus need to be modified only when using a different FLASH device. For each command, the parallel FLASH devices use the same number of write bus cycles. The only difference would be the address and data values used in the sequence. Thus, the only change required for a different FLASH device would usually be the modification of the address and data values.

Reading Data from Parallel FLASH

Once the External Port and AMI controller are initialized, reading data from the parallel FLASH device is as simple as reading data from internal/external SRAM.

Erasing Data from Parallel FLASH

Erase operation converts "zeros" to "ones". This operation involves six bus write cycles followed by a routine to poll and wait for the FLASH device to indicate that the erase operation has been successfully completed. Though there are various polling methods possible, the code supplied with this EE-Note uses *"DQ7 Polling Mechanism"*. Figure 16 shows the software flow required to erase a block/sector of a parallel FLASH device.

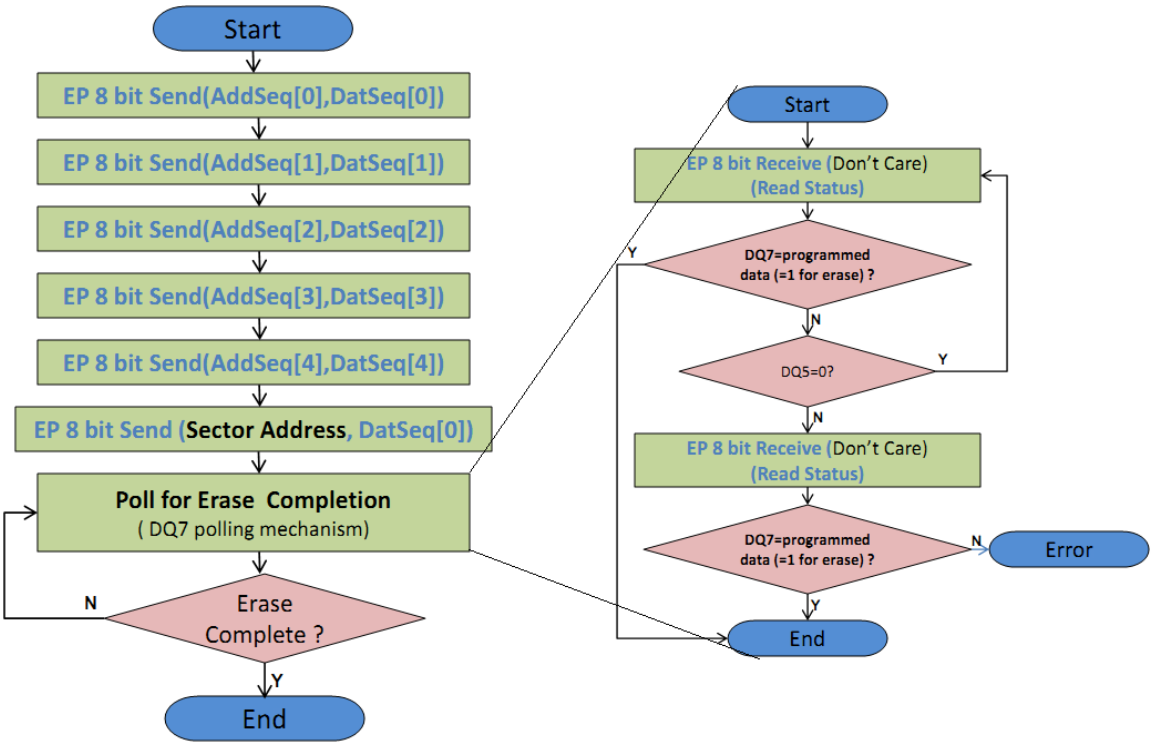


Figure 16. Parallel FLASH sector erase and DQ7 polling mechanism

Writing Data to Parallel FLASH

Writing converts "ones" to "zeros". This operation involves four bus write cycles followed by a routine to poll and wait for the FLASH device to indicate that the write operation has been successfully completed. Though, there are various polling methods possible, the code supplied with this EE-Note uses *"DQ7 Polling Mechanism"*. Figure 17 shows the software flow required to program a single byte to a particular address of a parallel FLASH device.

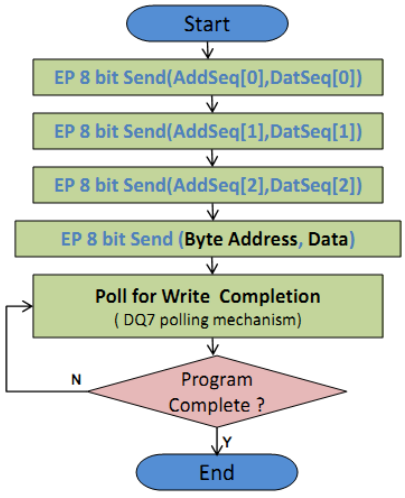


Figure 17. Parallel FLASH byte program

Programming an LDR file into the FLASH

This section discusses the complete framework, which calls the various functions discussed above to program an LDR file into the FLASH device. Figure 18 shows the flow of the main function.

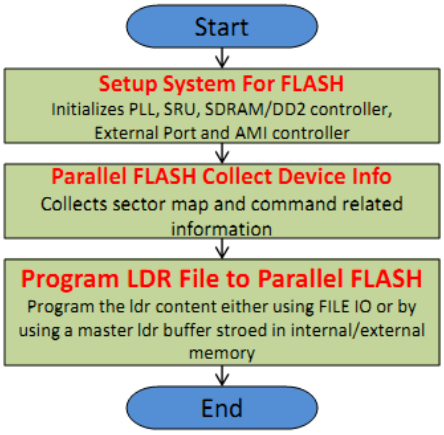


Figure 18. Main function for programming the LDR file to the serial FLASH device

The first function sets up the system (PLL, DDR2/SDRAM controller, External Port, AMI controller, etc.) for the SHARC-FLASH interface. It then calls the function "Parallel FLASH Collect Device Info", which program flow is shown in Figure 19.

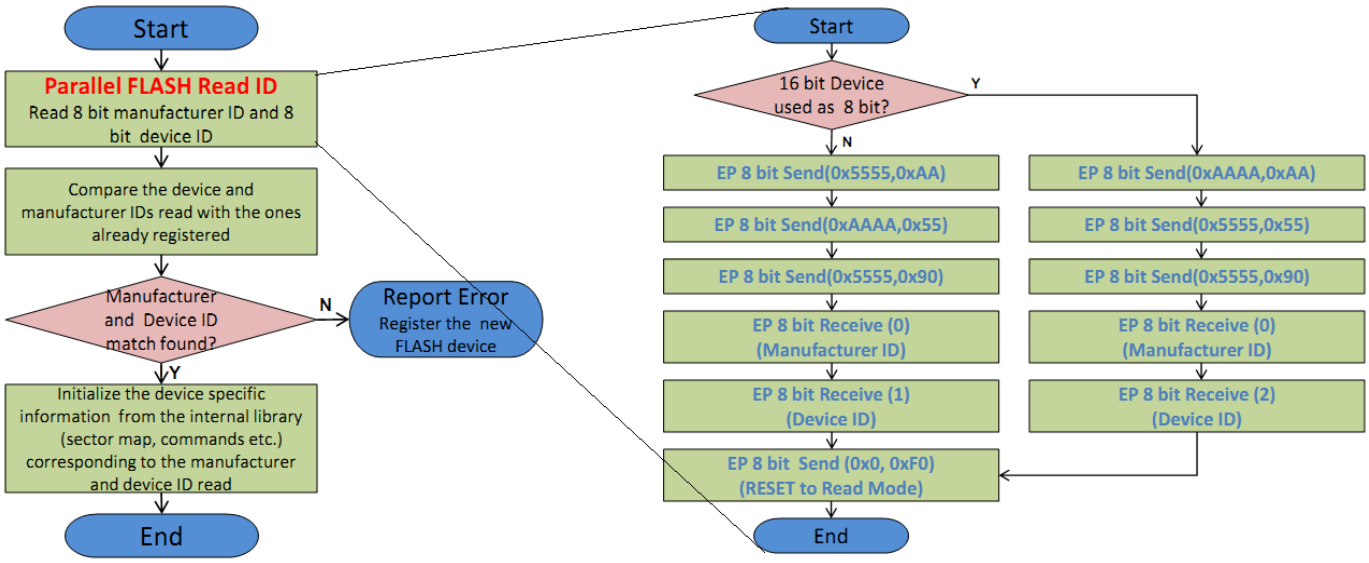


Figure 19. Parallel FLASH collect device info

This function performs an "Auto Device Detection" to initialize various device specific parameters, such as device manufacturer's name, device name, size, sector map info, command set, etc. This is achieved by sending the Auto Select command to the FLASH device. If a match is found for both manufacturer and device ID, it means that the

device specific information for that FLASH device is already available in the existing library, and thus it will be initialized accordingly. Otherwise, the user will have to register the FLASH ID by entering the device specific information, as specified in the data sheet.

Figure 20 shows the program flow required to program an LDR file into the parallel FLASH device.

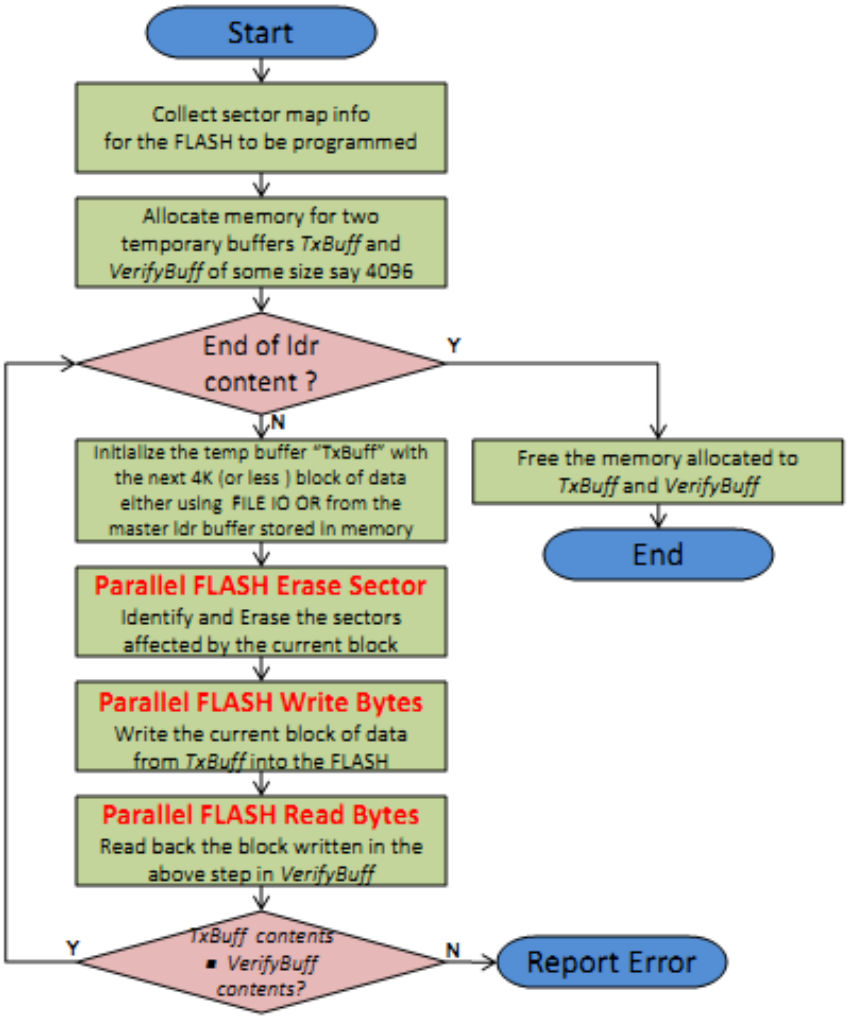


Figure 20. Programming LDR file contents into the parallel FLASH device

How to use the Example Code

The code example *"SHARC_Expert_Parallel_FLASH_Programmer"* provided in the associated `.ZIP` file has been developed based on the above described software architecture, in such a way that the same project can be used for different SHARC processors and a different SPI FLASH device, with minimal modifications. The file *"SystemInit.c"* contains system initialization functions, "*EPInit.c*" contains External Port related functions, and the file *"Parallel_FLASH.c"* contains FLASH related functions.

Before the provided example code can be built and run for a parallel FLASH device connected to a SHARC processor, the user should consider:

- **Processor Specific Settings:** The example code supports all SHARC processors. The files *"EPIInit.c/h"* and *"Parallel_FLASH.c/h"* are *processor independent*, so no changes are required for these files. The file *"SystemInit.c/h"* contains *processor dependent* code, such as PLL, DRAM controller, External Port, and AMI controller initializations. The CCLK and SDCLK/DDR2CLK are by default programmed to their maximum allowed values for the corresponding EZ-KIT boards. The External Port and AMICTL is programmed to use bank 1 with maximum wait states used at reset for AMI master booting. So, the user would need to change these settings, for a given custom system.

- **Parallel FLASH Specific Settings**: The example code supports parallel FLASH devices M29W320EB, M29W040B, AM29LV081, and SST39LF040. For programming of a different FLASH device, the user should register the new device by entering the device specific parameters, as described in the earlier section. For example, the FLASH devices M29W320EB, M29W040B, AM29LV081, and SST39LF040 are already registered in the subroutine *"Parallel_FLASH_Collect_Device_Info"* of the source file *"Parallel_FLASH.C"*. Figure 21 shows a screenshot of the registration entries for the M29W320EB FLASH device.

```
void Parallel_FLASH_Collect_Device_Info(PARALLEL_FLASH_DEVICE_INFO* parallel_flash_device_info)
{

    //Read device ID and manufaturer ID
    Parallel_FLASH_Read_Identification(&(parallel_flash_device_info->IDInfo));

    //Manufaturer name and Device name
    switch(parallel_flash_device_info->IDInfo.cManID)
    {
        //Numonyx/ST Micro
        case 0x20:
        default:
                //Initialize the manufaturer name here
                strcpy(parallel_flash_device_info->ManName,"Numonyx/ST Micro");

                //Compare the device ID to check if the device is M29W320EB - 214xx EZ-Kits
                if(parallel_flash_device_info->IDInfo.sDeviceID==0x57)
                {
                    //Initialize the complete device name here
                    strcpy(parallel_flash_device_info->DeviceName,"M29W320EB - 32 Mbit 3V supply Flash memory");

                    //M29W320EB does not have uniform sectors, so "false"
                    parallel_flash_device_info->SectorInfo.bUniformSectors          =   false;

                    //Initialzing the main sector info here
                    parallel_flash_device_info->SectorInfo.uiBytesPerSector         =   65536;
                    parallel_flash_device_info->SectorInfo.uiTotalSectors           =   63;


                    //Parameter blocks are at the bottom (starts from first location), so false
                    parallel_flash_device_info->SectorInfo.bTopParameterBlock       =   false;

                    //Initializing the parameter block info here
                    parallel_flash_device_info->SectorInfo.uiBytesPerParameterBlock =   8192;
                    parallel_flash_device_info->SectorInfo.uiTotalParameterBlocks   =   8;

                    //Initialize total device size in bytes
                    parallel_flash_device_info->SectorInfo.uiTotalBytes=                33554432;

                    //M29W320EB is a 16 bit device used in byte mode, so "true"
                    parallel_flash_device_info->CommandSet.bWordAsByte              =   true;


                    //Initialize Sector Erase Address and Data Sequence here
                    parallel_flash_device_info->CommandSet.uiSectorEraseAddSeq[0]   =   0xAAA;
                    parallel_flash_device_info->CommandSet.uiSectorEraseDatSeq[0]   =   0xAA;
```

Figure 21. Device registration of the M29W320EB parallel FLASH device

To register a new parallel FLASH device from an already registered manufacturer, the user would have to add the device specific information with a new "else if" structure. Alternatively, the user would have to add another "case" structure. Also, comment/uncomment the macro

"WORD_MEMORY_USED_FOR_BYTE" in *"Parallel_FLASH.h"* file based on whether the FLASH device is an 8 bit device or a 16 bit device being used in *BYTE* mode.

- **LDR Programming Method Selection**:
    - Using memory to store the LDR file contents-
        - To store the complete LDR file contents in internal SRAM, uncomment the macro definition *"USE_MEMORY_STORAGE_FOR_LDR_PROGRAMMING"* and comment out the definitions *"USE_EXTENAL_MEMORY_FOR_LDR_STORAGE"* and *"USE_FILE_IO_FOR_LDR_PROGRAMMING"*. This is the fastest and most preferable method, but it can only be used if the LDR file is small enough to fit into the processor's internal memory.
        - If internal memory is not enough, to store the complete LDR file contents in external SDAM/DDR2RAM/SRAM, uncomment the macro definition *"USE_MEMORY_STORAGE_FOR_LDR_PROGRAMMING"* and *"USE_EXTENAL_MEMORY_FOR_LDR_STORAGE"* and comment out the definition *"USE_FILE_IO_FOR_LDR_PROGRAMMING"*. This method is slower than the above method but faster than the FILE I/O method described next.
    - Using the File I/O to store the LDR file contents (not applicable to ADSP-2137x processors because of the reduced internal memory size), uncomment the macro definition *"USE_FILE_IO_FOR_LDR_PROGRAMMING"* and comment out the definitions *"USE_MEMORY_STORAGE_FOR_LDR_PROGRAMMING"* and *"USE_EXTENAL_MEMORY_FOR_LDR_STORAGE"*. This is the slowest method of all three, but is not limited by the size of the LDR file.
- Change the "Processor" in "Project Options" to the processor being used.

Conclusion

In this EE-Note, we have explored the various features and internal memory architectures of both, serial and parallel FLASH devices. Both, hardware and software aspects of how these devices and how they can be interfaced to SHARC processors have also been discurssed. The two example codes supplied with this application note can be used as a FLASH programmer tool for a number of FLASH devices as well as different SHARC processors, with very minimal modifications. Though the main function provided with these codes focuses on programming the LDR file into the FLASH device, the drivers associated with the combinations of the three other files (SystemInit, SPIInit/EPInit, and Serial_FLASH/Parallel_FLASH) can also be used to communicate with the FLASH device for other purposes.

References

[1] *ADSP-214xx SHARC Processor Hardware Reference*, Rev 1.0, February 2012. Analog Devices, Inc.

[2] *ADSP-2137x SHARC Processor Hardware Reference (includes ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371, ADSP-21375).* Rev 2.1, May 2012. Analog Devices, Inc.

[3] *ADSP-2136x SHARC Processor Hardware Reference (includes the ADSP-21362/ADSP-21363/ADSP-21364/ADSP-21365/ADSP-21366 processors*). Rev 2.0, June 2009. Analog Devices, Inc.

[4] *ADSP-2126x SHARC Processor Hardware Reference.* Rev 5.0, August 2010. Analog Devices, Inc.

[5] *ADSP-21261/ADSP-21262/ADSP-21266 SHARC Processor Data Sheet.* Rev F, August 2009. Analog Devices, Inc.

[6] *ADSP-21362/ADSP-21363/ADSP-21364/ADSP-21365/ADSP-21366 SHARC Processor data sheet.* Rev G, March 2011. Analog Devices, Inc.

[7] *ADSP-21367/ADSP-21368/ADSP-21369 SHARC Processors Data Sheet.* Rev E, July 2009. Analog Devices, Inc.

[8] *ADSP-21371/ADSP-21375 SHARC Processor Data Sheet.* Rev C, September 2009. Analog Devices, Inc.

[9] *ADSP-21467/ADSP-21469 SHARC Processor Data Sheet.* Rev A, December, 2011. Analog Devices, Inc.

[10] *ADSP-21477/ADSP-21478/ADSP-21479 SHARC Processor Data Sheet.* Rev B, April, 2012. Analog Devices, Inc.

[11] *ADSP-21483/21486/21487/21488/21489 SHARC Processor Data Sheet.* Rev A, April, 2012. Analog Devices, Inc.

[12] *EE-223-" In-Circuit Flash Programming on SHARC® Processors"*. Rev 2, February , 2007. Analog Devices Inc.

[13] *EE-231 - In-Circuit Programming of an SPI Flash with SHARC® Processors.* Rev 2, August, 2007. Analog Devices Inc.

[14] *M29W320EBData Sheet.* Rev 6, March 2008. Numonyx.

[15] *SST39LF040 Data Sheet.* Rev A, August, 2011. Silicon Storage Technology, Inc.

[16] *M29W040B Data Sheet.* September, 2005. STMicroelectronics, Inc.

[17] *AT25F2048 Data Sheet.* 2007. Atmel.

[18] *M25P20 Data Sheet.* Rev 14. March 2010. Numonyx.

[19] *M25P16 Data Sheet.* Rev 14. March 2010. Numonyx.

[20] *SST25WF040 Data Sheet.* June 2011. Silicon Storage Technology, Inc.

[21] *W25X40BVSNIG Data Sheet.* August 2009. Winbond.

[22] *AN551 – "Serial EEPROM Solutions vs. Parallel Solutions"*. 1993. Microchip Technology Inc.

[23] *AN1158 – "Uniform vs. Boot Block Flash Architectures"*. October, 1999. STMicroelectronics.

Document History

Revision	Description
Rev 1 – August 6, 2012 *by Mitesh Moonat and Harshit Gaharwar*	Initial Release