ebFlashSetup

User Manual

Version 1.1

Peer Georgi

March 1, 2006

Contents

Chapter 1

Introduction

ebFlashSetup is designed for processing file contents for programming them into the Read Only Memory (e.g. Flash Memory) of an embedded system. The program generates all necessary data for configuring the ROM. The program coordinates the file transfer and works together with the file transfer program („*ebStartUp*"). It uses the project file to generate a configuration file for „ebStartUp".

1.1 Program structure

„ebFlashSetup" is part of a group of programs that work together to start an embedded system out of flash memory. The following programs are required:

- The boot loader of the target system.
  It copies programs and data from ROM to RAM and executes the program.
- The ROM configuration program.
  It runs on the target system and programs the flash memory.
- The transfer program „ebStartUp".
  It transfers program and data into the RAM of the target system and executes the program.
- The data processing program „ebFlashSetup".

EbFlashSetup performs memory management tasks for ROM and RAM of the target system and configures the transfer program. It manages the target RAM for the following two situations:

1. The startup process.
2. The data transfer with „ebStartUp".

During startup, specified data blocks have to be copied from ROM to RAM. The target addresses in the RAM are given in the configuration file.
During transfer, the target RAM is used for temporarily buffering the data. „ebFlashSetup" takes care of avoiding address and data conflicts.

Additionally, the ROM is managed for minimizing unused areas - gaps - between stored data blocks. A memory allocation table (similar to a **F**ile **A**llocation **T**able) is created and used by the boot loader for reading the data blocks.

After defining all required memory addresses for every situation, the ebStartUp configuration file is created. The file is used for the setup process when data are transferred to the target system and written into flash ROM. In order to communicate with the program that writes data into the flash ROM, a table is needed that contains information about the RAM and ROM location of the data blocks. The structure of this table, which also is created by ebFlashSetup, is described here.

1.2 Usage

„ebFlashSetup” is a console program, set up by a project file. The program behavior is controlled by command line parameters and environment variables.

1.2.1 Command line parameters

Command line parameters are passed to the program at execution time. Some parameters are optional. If such a parameter is missing, default settings are used. Parameters have a short and long notation. The short notation ("-p") is for lazy typists, while the long notation ("–project") offers better documentation.

1.2.1.1 Configuartion

Projct file name
At program start the configuration file "setup.prj" is read from the current directory. A different file name can be given with the command line option "-p filename" or "–project filename".

ebStartUp configuration file name
„setup.ebs” is the default file name for the ebStartUp configuration file to be generated. A different file name can be given with the command line option "-o filename" or "–output filename".

Output
By default, only errors, warnings or essential messages are printed to the standard output device. The parameter "-q" resp. "–quiet" restricts the output to errors only. The parameter "–verbose" causes printout of all sorts of information.

1.2.1.2 Help

The parameter "–help" prints a list of command line parameters with brief descriptions. This list may be more up to date than this manual.

1.2.1.3 Documentation

The parameter „-d” prints a documentation in Latex format. Please find details under „Extras”.

1.2.2 Environment variables

Environment variables can be used for configuring the project file. They are treated like normal variables. See also 2.3.5.

1.3 Supported target platforms

The program was designed for data conversion and configuration for ARM9 based embedded systems. It is suited for all embedded systems that provide a boot loader and transfer programs.

1.4 Supported operating systems

- Linux

At the moment only Linux is supported.

Chapter 2

The project file

The configuration file is a plain text file that can be edited with any text editor. The line break codes of "MacOS", "Windows" and "Unix" are supported. The file is hierarchically organized and uses keywords and blocks for structuring.

2.1 Syntax

The syntax is similar to C or C++. Statements end with a semicolon ";". Comments start with "//" and end at the end of the line. Blocks are opened with "{" and closed with "}". Keywords are **case sensitive**.

2.2 Structure

The project file consists of two sections, the configuration block and the instruction section. The instruction section contains instruction blocks. An instruction block specifies the file, the transfer method, and the type and address properties of the file that affect the address generation.

2.2.1 Configuration block

This block begins with the keyword „GlobalSettings" and defines several global variables used by the program. Local variables can also be defined and used, however only within the configuration block.

2.2.2 Instruction section

This section begins with the keyword „TransferSetup" and contains one or several instruction blocks.

2.3 Variables

They are heavily used in the project file. There are different variable types:

- Simple variables
- Function variables
- Configuration variables

2.3.1 Simple variables

They are what you expect variables to be. They are defined by their useage. Values are assigned like this:

```
// Simple assignment
Filename = „datei.bin”;
// Assigning a variable content
Imagename = $(Filename);
// Assigning an environment variable content
UserName = $$(USER);
```

If an assignment is not possible, for instance when an environment variable is undefined, an error message will be issued. Finite recursive assignments are possible.

2.3.2 Function variables

They are built-in functions, used for assigning contents calculated by ebFlashSetup in real time.

```
// The next free address in the transfer RAM
TransferLoadAddress = $(@FIRST_POSSIBLE_TRANSFER_SDRAM_ADDRESS);
```

This assignment copies the next free RAM address into the „TransferLoadAddress” variable. The real value is calculated at assignment time.
The following variables are function variables. They are valid within transfer blocks only.

- ***@FIRST_POSSIBLE_TRANSFER_SDRAM_ADDRESS***
  Next unused address in the target RAM during the data transfer.
- ***@FIRST_POSSIBLE_FLASH_ADDRESS***
  Next free target ROM address.
- ***@FIRST_POSSIBLE_BOOTLOAD_SDRAM_ADDRESS***
  Next free target RAM address during startup.
- ***@FLASH_FILE_MAP_IMAGE_NAME***
  Name of the file containing the allocation table to be generated.
- ***@FLASH_SETUP_MAP_IMAGE_NAME***
  Name of the file containing the setup table to be generated.
- ***@IMAGE_FILE_SIZE***
  Size of the current image file.

Using function variables greatly simplifies the project file because start addresses can be calculated by ebStartUp itself.

2.3.3 Configuration variables

For calculating memory addresses, the program must be fed with information about the used memory. Those informations must be defined in built-in configuration variables within the configuration block of the project file. The following variables have to be defined:

- ***SDRAM_SIZE***
  Gives the target RAM size in bytes.
- ***FLASH_SIZE***
  Gives the target ROM size in bytes.

- ***FLASH_FILE_MAP_IMAGE_NAME***
  Gives the file name containing the generated allocation table for the boot loader.
- ***FLASH_FILE_MAP_ITEMS***
  Gives the maximum number of entries in the allocation table. If the number is too small, an error message is issued.
- ***FLASH_SETUP_MAP_IMAGE_NAME***
  Gives the file name containing the setup table for the flash setup program.
- ***FLASH_SETUP_MAP_ITEMS***
  Gives the maximum number of entries in the setup table. If the number is too small, an error message is issued.
- ***SDRAM_BOOTLOAD_START_ADDRESS***
  The initial value for the first free address in the target RAM at system start.
- ***SDRAM_TRANSFER_START_ADDRESS***
  The initial value for the first free address in the target RAM during transfer.
- ***FLASH_START_ADDRESS***
  The first used target ROM address.

All integer values are given in hex format with leading „0x". Strings, like file names, are given in double quotes. Memory area conflicts are solved or will be indicated with an error message.

2.3.4 Scopes of validity

Built-in variables used for the configuration are valid within the configuration block as well as within transfer blocks. User defined variables are only valid within their block of definition. Function variables are only valid within instruction blocks.

2.3.5 Environment variables

Environment variables can be used whenever a constant or variable content can be assigned. If they are undefined at runtime, an error message is issued. They can be used to make a project externally configurable.

2.4 Instruction blocks

An instruction block contains the following information:

- „***Image***" = [file name]
  The file name.
- „***BootLoadAddress***" = [integer]
  The start address at system start.
- „***TransferLoadAddress***" = [integer]
  The start address during transfer and setup.
- „***StoreToFlash***" = [yes | no]
  Whether to write the file into ROM or not.
  - „***FlashAddress***" = [integer | ignore]
    The ROM start address.

- „***Execute***” = [yes | alternate | no]
  Whether to execute the file at system start or not.

► „***Protocol***” = [USB_DFU | USB_LDBA | ...]
The protocol to be used for the data transfer.

► „***ProtocolParam***”
Begins a block whose content is passed directly to the ebStartUp configuration file, without further affecting ebFlashSetup.

Those statements can be given in any order within an instruction block.

A typical instruction block might look like this:

```
Transfer
  {
    StoreToFlash = yes;
    Image = "LdDataFlash";
    FlashAddress = 0x0;
    BootLoadAddress = ignore; // the cpu will load this automaticaly //
    TransferLoadAddress = $(@FIRST_POSSIBLE_TRANSFER_SDRAM_ADDRESS);
    Execute = no;
    Protocol = "USB-LDBA";
    ProtocolParam
      {
        Comment = "Loading LdDataFlash...\\\\n  ";
         Filename = "$(Image)";
         LoadAddress = $(TransferLoadAddress);
      }
  }
```

This block describes a file („LdDataFlash”) to be written into target ROM. The flash ROM start addess is explicitely set at 0. The start address at system start however is omitted. As the MCU automatically loads the first block from ROM, this entry is ignored by the boot program (by the way, the first block is the boot program itself).
The start address during transfer does not matter and is left to ebFlashSetup to calculate. The „Execute = no” field means that the boot program shall not execute this entry (because the MCU will execute it anyway).

LDBA is used for the transfer protocol, and its parameters are stored in the ProtocolParam block.

Chapter 3

Examples

The examples can be used as templates for individual projects. However the variables have to be adapted in most cases.

3.1 Configuration block

This configuration block is intended for a target system with 16MByte RAM and 8 MByte flash ROM. RAM starts at physical address 0x20000000.
Thus the first valid start address at system start is 0x20000000. Using the function variable *„FIRST_POSSIBLE_BOOTLOAD_SDRAM_ADDRESS"* does not make sense in most cases because normally start addresses are predefined for the system start.

```
GlobalSettings
  {
    SDRAM_SIZE = 0x1000000; // 16MB
    FLASH_SIZE = 0x800000;  // 8MB
    FLASH_FILE_MAP_IMAGE_NAME = "filemap.img"; // the name of filemap-image
    FLASH_FILE_MAP_ITEMS = 0x10; // number of entries in file_map
    FLASH_SETUP_MAP_ITEMS = 0x10; // number of entries in setup-map
    FLASH_SETUP_MAP_IMAGE_NAME = "setupmap.img";
    // start values //
    SDRAM_BOOTLOAD_START_ADDRESS = 0x20000000;
    SDRAM_TRANSFER_START_ADDRESS = 0x20000000;
    FLASH_START_ADDRESS = 0x0;    // start with this address //
  }
```

3.2 A real example

The target system is a MCU module by Conitec Datasystems Corp. It's an ARM9 based embedded system comprising the AT91RM9200 microcontroler. The system contains 128 MByte RAM and 8 MByte ROM.
After the setup process described here, a program has to be executed automatically on the target system. For this it has to be copied from ROM to RAM after reset, and then executed. The target ROM does not contain a valid program at first.
In that case the MCU waits after reset for a XMODEM or USB-DFU connection for receiving and starting a program. For details see the MCU data sheet.

The following programs are available:

- LdDataFlash
  The boot loader.
- StDataFlash
  The ROM setup program.
- LdBigAppUSB
  The LDBA transfer program.
- The test program to be executed at system start.
  Call it „ping.bin".

3.2.1 Prerequisites

At first it has to be determined at which RAM addresses the programs are to be executed and which other settings are required.

LdDataFlash

This program is automatically loaded in the cache by the MCU. For this it has to be located at ROM address 0. The program also expects the allocation table in the ROM, which describes the further ROM content. The allocation table address is predefined at 0x4000, as the MCU loads 16 kB from the ROM and the boot loader must not exceed this size. The allocation table therefore begins immediately after the boot loader.
Additionally the program start address is required. This address - 0x200000 - is defined when the program is linked. 0x200000 is the MCU cache address where the program is copied to.
Therefore we have the following informationen:

- **Execution start addresse:** ***0x200000***
  However as the program is executed by the MCU and not by the boot loader, the address can be ignored in the ebFlashSetup project file.
- **ROM start address:** ***0x0.***
- **Allocation table start address in ROM:** ***0x4000.***

StDataFlash

The program's task is configuring the ROM. It is automatically executed after the configuration is finished. It expects the setup table at a RAM address specified in the program. This table has the structure described in the chapter 5.2.

- **Execution address:** ***0x20080000***
  This address results from the linker script.
- **setup table start address:** ***0x20070000***
  This address is defined by the program itself.
- The program is not loaded into ROM as it has to configure it.

ping.bin

This program has to be started automatically. The execution address is defined by the program developer and given in the linker script. The ROM address is not predefined. It is determined by ebFlashSetup in order to minimize ROM gaps. There are no further data location requirements.

LdBigAppUSB

This program is the counterpart to „ebStartUp" and offers fast data transfers via USB port. It transfers the data to the target system and initiates the StDataFlash startup for running the configuration process.

Because the ROM is empty at startup, the program is transferred by USB-DFU protocol and started by the MCU. After program start *LdBigAppUSB* offers a transfer channel for the further data transfers to the target RAM, using the **LDBA** (**Lo**a**d** **B**ig **A**pplications) protocol. The execution address does not matter because the program is transferred by DFU protocol, which uses an address predefined by the MCU and set by the link process.

The preparations are finished now and the project file for ebFlashSetup can be created.

3.2.2 Creating the project file

At first the global parameters must be defined. This is basically the same as in the example in 3.1.

GlobalSettings

```
{
  SDRAM_SIZE = 0x10000000; // 256MB
  FLASH_SIZE = 0x800000;  // 8MB
  FLASH_FILE_MAP_IMAGE_NAME = "filemap.img";
  FLASH_FILE_MAP_ITEMS = 0x4;
  FLASH_SETUP_MAP_ITEMS = 0x10;
  FLASH_SETUP_MAP_IMAGE_NAME = "setupmap.img";
  // start values //
  SDRAM_BOOTLOAD_START_ADDRESS = 0x20000000;
  SDRAM_TRANSFER_START_ADDRESS = 0x20000000;
  FLASH_START_ADDRESS = 0x0;
}
```

Next, the instruction section:

```
TransferSetup
  {
    // alle Anweisungsblöcke befinden sind innerhalb
    // dieses Blockes...
```

Next step is the initialization of the communication channel. LdBigAppUSB must be transferred at first. It is then immediately executed by the target MCU.

```
    Transfer
      {
        TransferLoadAddress = 0x200000; // internal SRAM (CPU)
        Image = "LdBigAppUSB";
        StoreToFlash = no;
        Protocol = "USB-DFU";
        ProtocolParam
          {
            Filename = "$(Image)";
          }
      }
```

In the following the definitions for the start program (boot loader) are given. The ROM address is specified, while the RAM address does not matter because the program is stored only temporarily in RAM. The Execute field is required for the boot loader. Because the boot loader is anyway automatically started by the MCU, this field is set at „no”.

```
Transfer
  {
    TransferLoadAddress = $(@FIRST_POSSIBLE_TRANSFER_SDRAM_ADDRESS);
    Image = "LdDataFlash";
    StoreToFlash = yes;
    FlashAddress = 0x0;
    BootLoadAddress = ignore;
    Execute = no;
    Protocol = "USB-LDBA";
    ProtocolParam
      {
        Filename = "$(Image)";
        LoadAddress = $(TransferLoadAddress);
      }
  }
```

Next follows the allocation table for the boot loader. It is stored to the address 0x4000 in ROM.

```
Transfer
  {
    TransferLoadAddress = $(@FIRST_POSSIBLE_TRANSFER_SDRAM_ADDRESS);
    Image = $(FLASH_FILE_MAP_IMAGE_NAME);
    StoreToFlash = yes;
    FlashAddress = 0x4000;
    BootLoadAddress = ignore;
    Execute = no;
    Protocol = "USB-LDBA";
    ProtocolParam
      {
        Filename = "$(FLASH_FILE_MAP_IMAGE_NAME)";
        LoadAddress = $(TransferLoadAddress);
      }
  }
```

Now the program that is to be executed.

```
Transfer
 {
   TransferLoadAddress = $(@FIRST_POSSIBLE_TRANSFER_SDRAM_ADDRESS);
   Image = "ping.bin";
   StoreToFlash = yes;
   FlashAddress = $(@FIRST_POSSIBLE_FLASH_ADDRESS);
   BootLoadAddress = 0x20780000;
   Execute = yes;
   Protocol = "USB-LDBA";
   ProtocolParam
     {
       Filename = "$(Image)";
       LoadAddress = $(TransferLoadAddress);
     }
 }
```

All preparations concerning the ROM are now finished. The next step is preparing the program for flash memory programming (StDataFlash).

At first the setup table must be transferred. This table is created by *ebFlashSetup*. The target address is given by the StDataFlash program (see „Prerequisites”)

```
// Setup-List for StDataFlash
Transfer
  {
    TransferLoadAddress = 0x20070000;
    Image = $(FLASH_SETUP_MAP_IMAGE_NAME);
    StoreToFlash = no;
    Protocol = "USB-LDBA";
    ProtocolParam
      {
        Filename = "$(FLASH_SETUP_MAP_IMAGE_NAME)";
        LoadAddress = $(TransferLoadAddress);
      }
  }
```

After defining all those transfers, the setup process has to be specified. The program „StDataFlash” must be transferred and executed.

```
Transfer
  {
    TransferLoadAddress = 0x20080000;
    Image = "StDataFlash";
    StoreToFlash = no;
    Protocol = "USB-LDBA";
    ProtocolParam
      {
        Filename = "$(Image)";
        LoadAddress = $(TransferLoadAddress);
        Execute = yes;
      }
  }
} // closing the block „TransferSetup”
```

This file is available as configuration example (example1.prj). Just enter:

ebFlashSetup -p example1.prj

After running ebFlashSetup, the following files were created in the current directory:

- filemap.bin - the allocation table
- setupmap.bin - the setup table
- ***setup.ebs - the project file for ebStartUp***

All left to do for configuring the target system is entering:

ebStartUp

As soon as the target system is connected via USB and is reset, the configuration starts automatically. After the configuration process is finished, the target system is bootable.

Chapter 4

Extras

A special feature of „*ebFlashSetup*" is its ability to create a Latex documentation about the memory content. The output can be in German or English. The three memory allocation tables are documented. For this the command line parameter „-d [delen]" must be given. In the current directory a file named „*memorymap.latex*" is created. For the above example the following command line will create an English documentation:

```
ebFlashSetup -p example1.prj -d en
latex memorymap.latex
dvips memorymap.dvi
ps2pdf memorymap.ps
```

This will create a file named „***memorymap.pdf***" in the current directory. For the above example this file is inserted in the following pages.

4.1 Memory content documentation

4.1.1 RAM content during startup

Memory region	Name	Position	Size
0x20780000 | 0x20784133	ping.bin	fixed	16692 Bytes

4.1.2 ROM content

Memory region	Name	Position	Size
0x00000000 | 0x00002b6b	LdDataFlash	fixed	11116 Bytes
0x00004000 | 0x0000421f	filemap.img	fixed	544 Bytes
0x00004224 | 0x00008357	ping.bin	variable	16692 Bytes

4.1.3 RAM content during configuration

Memory region	Name	Position	Size
0x00200000 | 0x00202a57	LdBigAppUSB	fixed	10840 Bytes
0x20000000 | 0x20002b6b	LdDataFlash	variable	11116 Bytes
0x20004138 | 0x20004357	filemap.img	variable	544 Bytes
0x2000435c | 0x2000848f	ping.bin	variable	16692 Bytes
0x20070000 | 0x200701db	setupmap.img	fixed	476 Bytes
0x20080000 | 0x20083427	StDataFlash	fixed	13352 Bytes

Chapter 5

Internal structures

The allocation table for the boot loader and the setup table for the ROM setup program are created within files that are transferred to the target system, additionally to the data. The files are described here
.

5.1 The allocations table for the system start

This struct contains all necessary information about the data stored in ROM. It's a static struct and can not be expanded at runtime. The maximum number of entries can be defined in the project file with the configuration variable „***FLASH_FILE_MAP_ITEMS***". The table contains entries of following structure:

```
#define DF_BTENTRY_FLAG_USE             (1 << 0) // use the item //
#define DF_BTENTRY_FLAG_COPY            (1 << 1) // copy this item to sdram //
#define DF_BTENTRY_FLAG_EXEC            (1 << 2) // execute this item by default //
#define DF_BTENTRY_FLAG_EXEC_ALTERNATE  (1 << 3) // execute this item alternate //

// size = 16+4+4+4+4 = 32 bytes.
struct tDFBlockTableEntry
  {
    char msItem[16];                // name of image inclusive trailing zero //
    unsigned int mDfBTEntryFlags;   // load / execution flags //
    unsigned int mDfBTEntryDFStart; // start in dataflash //
    unsigned int mDfBTEntryDFSize;  // size of image //
    unsigned int mDfBTEntryMemStart; // location in sdram where the image to be load //
  }  __attribute__((packed));
```

The last entry is detected by the boot loader due to its initialization with 0. All data fields of the last entry are to be filled with 0 therefore.

5.2 The setup table for the ROM setup program

This table is copied to the target RAM during setup, and used by the ROM setup program for reading the RAM location of the transferred data blocks. The ROM setup program then writes the data blocks to the flash ROM. The structure is static and similar to the allocation table. The number of entries must be set through the configuration variable „***FLASH_SETUP_MAP_ITEMS***". Because the table is not written intoflash ROM, the number of entries can be high. The maximum number depends on the configuration program and its target address settings.

```
struct cSetupItem
  {
    char msItemName[16];            // zero terminated msItemName[0] == 0 means end of list //
    unsigned int miDFStartAddress;  // start address in dataflash //
    unsigned int miLoadAddress;     // address where the data copied from //
    unsigned int miSize;            // size of data object //
  };
```

The configuration program calculates the number of entries from the last entry that contains 0 in all data fields.