Freescale Semiconductor
Application Note

Document Number:AN4723
Rev 0, 07/2013

S12Z MagniV Bootloader

by: Arturo Inzunza

Contents

1 Introduction

This application note describes the boot operation of S12Z core and how to implement and use a bootloader application that can serially reprogram a device. The S12Z core is one of the key differences between the MagniV family and other 16-bit microcontrollers from Freescale. This document focuses on describing how this core manages the boot process, the memory layout and manipulation schemes, and the interrupt management. The bootloader application shared with this document can be matched with any serial communication protocol driver to provide flexibility to the implementation. Some popular alternatives for serial bootloaders are SCI, CAN, or LIN, even though I2C, SPI, or even regular GPIO (Bit-Banging) can be used. The bootloader application and the communications driver are clearly separated to provide the ability to replace and migrate the bootloader from one application (and/or MagniV device) to another.

The S12Z MagniV Bootloader aims to be a cross platform solution that can be easily ported to different MagniV devices. It takes the advantage that all the MagniV devices with an S12Z core share similar architectures. The bootloader was designed to be small so that it can be placed on smaller devices without consuming most of the storage resources. The following sections will focus on describing the operating characteristics of the S12Z core, the S19 record format, considerations when developing code to be loaded with the

bootloader and a quick description of the PC application server. The PC application can be used to stream an S19 file through a serial RS-232 port to a target.

2 S12Z core operation

The S12Z core has several differences with respect to other 16-bit cores from Freescale. The most important being the 24-bit addressing scheme. Previous 16-bit cores had 16-bit addressing and could only support 64 KB of direct memory addressing. This limitation forced the use of pages to implement larger memories and control of these memories became more complex. The 24-bit addressing capability of the S12Z core allows up to 16 MB of direct memory addressing. This means that pages are no longer required, but addresses are 3-byte long instead. The S12Z core also features a slightly modified boot mechanism that is simpler than the one on older cores.

2.1 Memory layout

The S12Z MagniV devices come in several memory ranges but all share a common layout. Taking advantage of the larger addressing capability, all memories: RAM, Program Flash (P-Flash), and EEPROM can be addressed directly without pages. EEPROM will not be analyzed on this document because the bootloader does not erase, program, or read this memory.

P-Flash memories range from 128 KB on the larger devices to 8 KB on the smaller ones. The P-Flash memory is placed at the end of the addressing space finishing on address 0xFFFFFF regardless of its size. The figure below shows the example of the P-Flash distribution and memory protection options of a 32-KB memory.

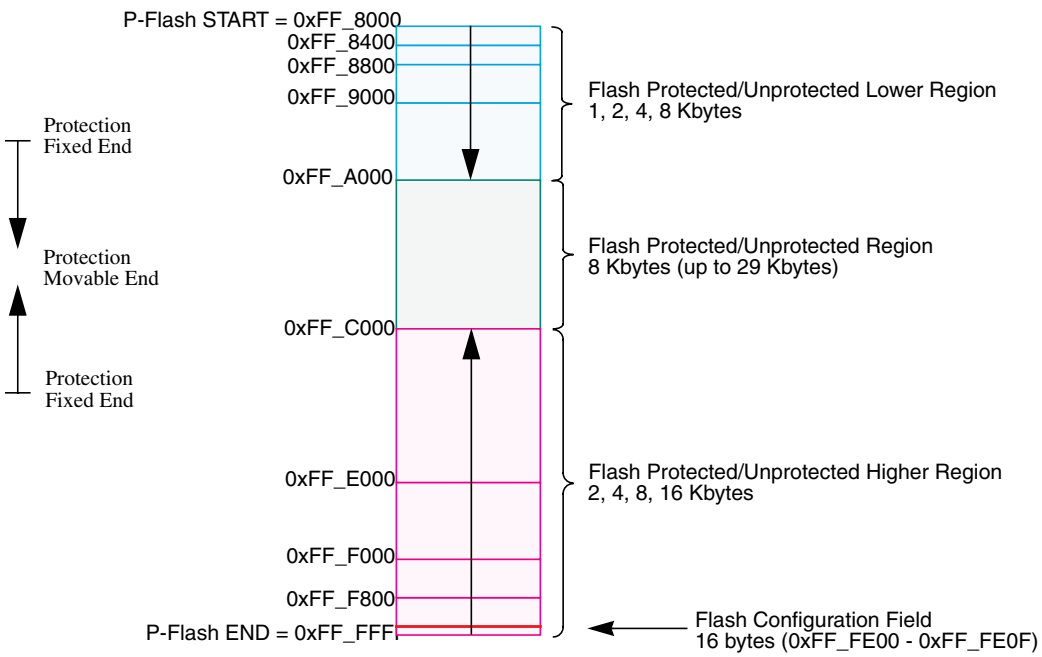

Figure 1. 32-KB P-Flash Layout

The memory technology used on the MagniV devices is composed of a single block divided into 512 bytes sectors. A sector is the smallest unit that can be erased by the memory controller. Table below shows the P-Flash memory layout data of different memory sizes on MagniV devices.

Table 1. P-Flash memory layout information

Memory size	Start address	End address	Sector number
128 KB	0xFE0000	0xFFFFFF	256
64 KB	0xFF0000	0xFFFFFF	128
32 KB	0xFF8000	0xFFFFFF	64
16 KB	0xFFC000	0xFFFFFF	32
8 KB	0xFFE000	0xFFFFFF	16

The last sector of the P-Flash memory holds the vector table and the NVM controller configuration information. This sector and its details are analized in detail in further sections of this document. Besides the P-Flash memory, the RAM memory is included on the general addressing map as well.

The RAM size, as the P-Flash size, varies from device to device. The largest MagniV devices can have up to 8 KB of RAM while the smallest has 1 KB. The RAM memory on the MagniV devices always starts from address 0x1000. Table below shows the RAM memory layout information for different implementations.

Table 2. RAM memory layout information

Memory size	Start address	End address
4 KB	0x1000	0x2000
2 KB	0x1000	0x1800
1 KB	0x1000	0x1400

The flash and RAM start and end address information is used by the bootloader to determine which memory sections to erase and what RAM areas can be used to store data.

2.2 Interrupt vectors and NVM configuration

The last sector of the P-Flash is used to store the interrupt vector information, and the P-Flash protection configuration. The protection bytes on this NVM section are copied to the Memory Controller Register Space at Power-On-Reset (POR). Table 3 shows the address and names of the protection bytes on the last sector of the P-Flash.

At reset, the FPROT, DFPROT, FOPT, and FSEC bytes on the last P-Flash sector are copied to the registers with the same name placed on the Memory Controller register space. The actual memory protection functionality depends on the registers on the Memory Controller, so that a running application can change the protection configuration at runtime. At the next reset, the default values are copied from P-Flash to the registers again. Changing the protection values on P-Flash will ensure that the new configuration is loaded by default after each reset.

Table 3. P-Flash configuration bytes

Global address	Size (Bytes)	Description
0xFFFE00-0xFFFE07	8	Backdoor Comparison Key
0xFFFE08-0xFFFE09	2	Protection Override Comparison Key

Table continues on the next page...

Table 3. P-Flash configuration bytes (continued)

0xFFFE0A-0xFFFE0B	2	Reserved
0xFFFE0C	1	P-Flash Protection Byte (FPROT)
0xFFFE0D	1	EEPROM Protection Byte (DFPROT)
0xFFFE0E	1	Flash Non-Volatile Byte (FOPT)
0xFFFE0F	1	Flash Security Register (FSEC)

Besides the P-Flash protection bytes, the last P-Flash sector also holds the default interrupt vector table space that goes from 0xFFFE10 to 0xFFFFFF. The interrupt vector table holds the pointers to each of the functions that must service an interrupt request. Although the function pointers are 24-bits long, the interrupt function pointers are stored as 32-bit addresses (4 bytes) for compatibility. The S12Z core can address up to 124 interrupt sources although not all are implemented. The availability of an interrupt source depends on the implementation of each particular MagniV device. Consult the “Interrupt Vector Locations” table at reference manual of a particular MagniV microcontroller for details on specific interrupts. The interrupt table can be moved from its default position by changing the IVBR register.

Although each MagniV microcontroller can have different interrupts available, all of them have the same Startup Vector. The last interrupt vector of the interrupt table is reserved for this startup vector. After a POR, the MCU will go to the last vector, on the last P-Flash sector (0xFFFFFC-0xFFFFFF [4 bytes]) and will automatically jump to that pointer. Even though the interrupt table can be moved, after a reset the table goes back to its default position (the last P-Flash sector), so that the startup vector will always be the last 4 bytes of the P-Flash array. On previous 16-bit cores, the reset vector was placed on different locations depending on the reset cause, a Watchdog reset would jump to a different location than an External Pin Reset. The S12Z only has one reset vector regardless of the reset cause.

2.3 Memory manipulation

The Memory Controller on the MagniV devices, known as FMTRZ, is capable of erasing and writing Non-Volatile memory sectors on runtime, without extra power supplies. Some of the operating characteristics of the FTMRZ module are:

- The minimum erase unit is a sector (512 bytes)
- A sector must be erased before it can be written
- The minimum write size is 8 bytes
- Writes must be aligned to an 8-byte boundary
- Flash can’t be read-from and written-to at the same time

Erasing a sector will set all its bits to 1. After erasing a sector, the FTMRZ can try to write data into it but this can’t be done while executing code from Flash, since Flash can’t be read-from and written-to at the same time. To write to a Flash sector, the application must be executing code from RAM. Special care needs to be taken into consideration regarding write boundaries. The address of the write operation must always be aligned to an 8-byte boundary, which means the last three bits must be 0, for example 0xFFFE08. This is because the smallest write size is 8 bytes, in this example the write would modify bytes from 0xFFFE08 to 0xFFFE0F and the next write would be on 0xFFFE10, which is aligned to an 8 byte boundary again.

3 S-Record format

The bootloader application uses program files on S-Record format, also known as S19, SREC, or Sx. The S-Record format was developed by Motorola to represent the memory contents of a microcontroller. Several compilers are able to generate an S-Record file directly after compilation, so no extra tools are needed. The S-Record is the preferred program representation because it is a simple text file that can be easily parsed by a tool to program a microcontroller.

An S-Record file is organized in lines all starting with the letter ‘S’. The first two characters of the line identify the purpose of that line, for example:

- Identification
  - S0: Could contain project name and vendor information, depends on the compiler
- Data Sequence
  - S1: Actual memory data with 2-byte addressing (16-bits)
  - S2: Actual memory data with 3-byte addressing (24-bits)
  - S3: Actual memory data with 4-byte addressing (32-bits)
- Line Count
  - S5: The amount Data Sequence lines on the file (S1,S2, or S3)
- Program Starting address
  - S7, S8, or S9. Although it is not normally used

An S-Record file can have only one of each type of lines, for example, there can be no S1 and S2 records on a single file. Since the MagniV devices have 24-bit addresses, an S-Record file will have S2 lines representing the data to transfer.

Each of the record lines is composed of the following elements:

- Identifier: The first two characters of the line (S0, S2, S9, etc…).
- Line Byte Size: Next two characters represent the amount of bytes (in hexadecimal) on the line excluding the Identifier and itself.
- Address: Next 2, 3, or 4 bytes represent the address of the data, depends on the record type.
- Checksum: The last byte of the line. Calculates the checksum of all bytes except the identifier.

Figure below shows an example of an S-Record file for a MagniV microcontroller

```
1 S0580000433A5C53564E5C47726F75705C41727475726F5C4B6E6EF785C536F6674776172655…
2 S224FF81E0A407850A6105ABFE00111E1B901CE7F70B867D05000000010001100FFFF0100012A
3 S213FF8200000000000000001030002C000000000A5
4 S207FFFFFDFF80007E
5 S90380007C
```

Figure 2. Example of an S-Record file

In the above figure, line 1 represents the file identification and description (due the S0 identifier) and line 5 indicates the startup address (S9 identifier). The bootloader application has no need for these records to operate. Lines 2, 3, and 4 are S2 records, which represent data that must be written to memory and are of importance to a bootloader application. Table below separates each of these lines (S2) into its elements.

Table 4. Example of data fields on S2 records

Line	Id.	#Bytes [Hex]	Address [Hex]	Data [Hex]	Checksum [Hex]
2	S2	24	FF81E0	A407850A6105AB FE00111E1B901C E7F70B867D05000 00010001100FFFF 010001	2A
3	S2	13	FF8200	000000000000010 30002C000000000	A5
4	S2	07	FFFFFD	FF8000	7E

An S2 line can be at most 37 bytes long excluding the identifier (1 size + 3 address + 32 data + 1 checksum). The S2 line is the only S-Record line relevant to the MagniV Bootloader application.

4 Bootloader software

The bootloader software that comes with this application note was designed to receive an S-Record program and load it to P-Flash on the run. The S-Record program must be streamed one line at a time. The bootloader processes the data on each line before replying with an acknowledge message indicating that a new line can be sent. The bootloader software must manage P-Flash erase and write while receiving the next S-Record line serially.

4.1 General architecture

The bootloader application was designed independently from the communications protocol used to receive the S-Record data. The communications driver can be changed and adapted to a custom application while the bootloader software remains unmodified. This architecture offers flexibility and simplicity when implementing the bootloader on different applications and devices.

The bootloader application only relies on three peripheral modules of the MCU: The Clock and Power Management Unit (CPMU), the Memory Controller (FTMRZ), and the GPIO module. The communications driver is independent of the bootloader. Figure below shows the relationship between these different modules.

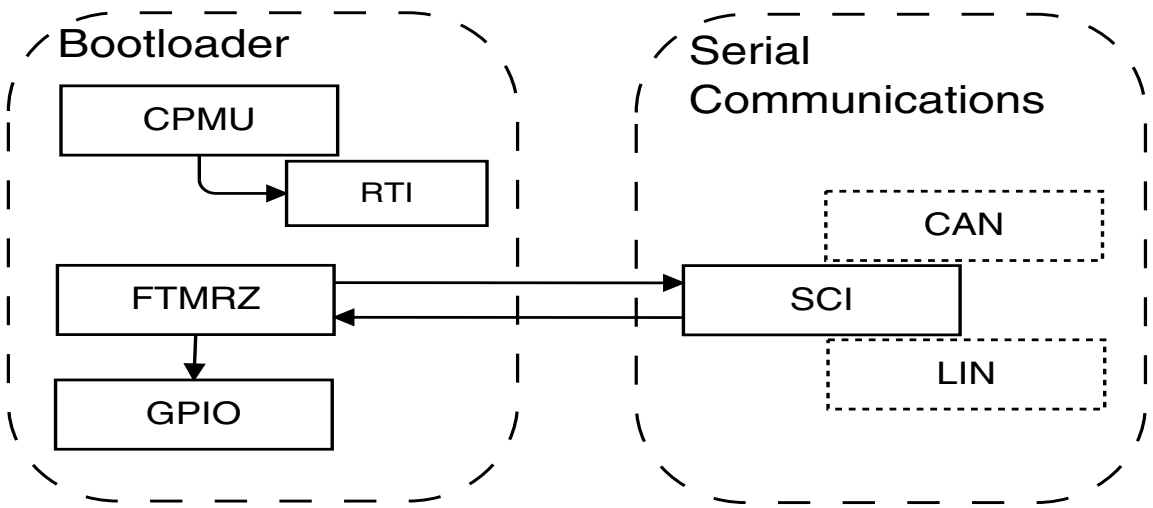

Figure 3. Bootloader application dependency diagram

At reset, the microcontroller starts running the bootloader application and the CPMU and RTI modules are configured first. The CPMU module configures the PLL to set the BusClock at 32 MHz. The RTI is used to keep track of time. After a configurable timeout, the bootloader stops waiting for an incoming program and jumps to the main application.

The FTMRZ module is the Memory Controller and is in charge of processing the received serial data and writes it into P-Flash. This module is also in charge of erasing all the P-Flash sectors on which the bootloader is not located, including the interrupt vector section. The GPIO is only used to configure a single “Activity” LED that flashes each time a new S-Record line is received. It is just a visual indication of activity and can be disabled.

The Serial Communications section of the application is independent of the rest of the bootloader. The S-Records can be received from any communication protocol and then sent to the FTMRZ module for processing. Data must be stored in a specific way for the FTMRZ to be able to process it. It is the responsibility of the communications driver to ensure that this formatting is met. This format is explained in more detail on the next section.

The bootloader application was designed to be small and therefore it does not use interrupts. Since the default interrupt vector needs to be erased, the bootloader application interrupt vector would require relocation, which impacts in size since more space is needed to allocate it. The data reception is ensured by polling the communications driver. Figure below shows the program flow of the complete bootloader application.

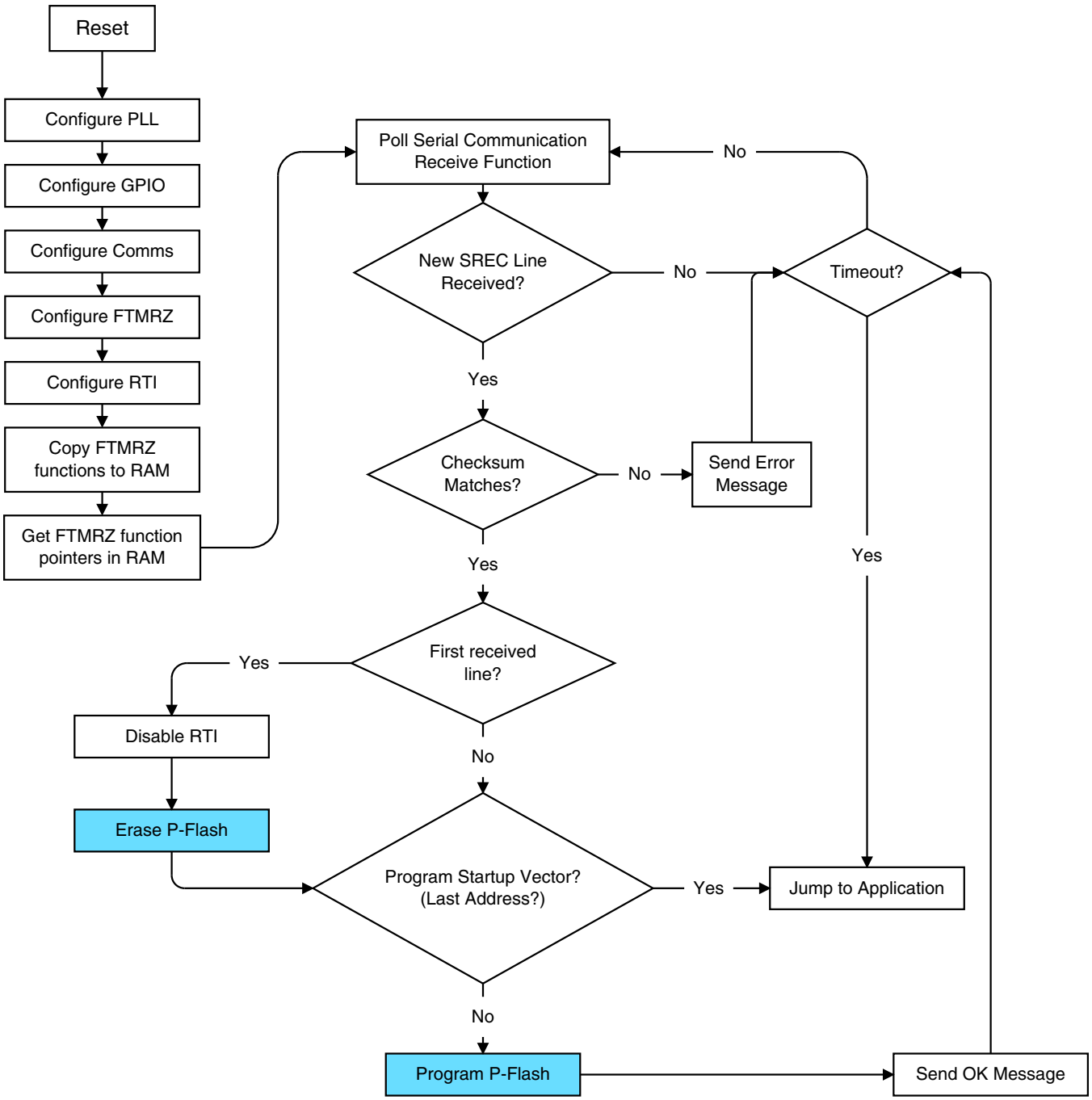

Figure 4. Bootloader application flow diagram

The bootloader application waits for data to be received. If data is received within the timeout, the FTMRZ module erases the P-Flash and proceeds to write the data. After the first S-Record line is received, the RTI module is disabled and a timeout cannot occur. Doing this ensures that the timeout will not be issued in the middle of the programming operation. When the FTMRZ needs to write the startup vector (the last four bytes of the P-Flash), it is considered that the programming has finished and no more data will be received.

NOTE

The startup vector of the received program (the S-Record) will NOT be written on the last address of the P-Flash. This address is reserved for the bootloader startup vector so that the bootloader starts on the next POR.

When either a timeout occurs or the received program has reached the last address (and therefore finished as well), the application will jump to the main application. The main application must start at a fixed address configurable on the bootloader. This address is fixed on compile time and cannot be changed once the bootloader has been loaded to the target device.

On Figure 4, the operations “Erase P-Flash” and “Program P-Flash” are colored blue to indicate that they are executed from RAM. The Flash memory can’t be written-to and read-from at the same time, so these two operations, that modify the Flash, need to be executed from RAM. At the program startup, these functions are copied to RAM and their new RAM pointers are generated. After the erase or write operation is finished, the execution returns to Flash until a new S2 record needs to be programmed.

4.2 Data reception

As mentioned before, the data reception section of the bootloader is independent from the rest of the application. Any communications protocol can be used to receive the new program and the user is given flexibility to implement his own (or third party) drivers. The communications driver is called from the bootloader to perform different tasks; the following functions must be implemented:

- ***void Comms_Init(void)*** – Communications initialization routine. This function must initialize the peripheral, configure clocks, and set the GPIO configuration, if necessary. It is called after the PLL has been configured (to 32 MHz) and is only called once.
- ***void Comms_Tx_Ack(enum ACK_OPTS ack)*** – Communications Transmit Acknowledge routine. This function is called to transmit the acknowledge message to the transmitting device. The acknowledge message is only one byte long. The enumeration ACK_OPTS can only have two values ‘OK’ or ‘CRC_Error’.
- ***void Comms_Rx_Data(void)*** – Communications Receive Data routine. This function is polled constantly to trigger the data reception mechanism. Received data must be stored in the ‘BootPhraseStructure BP’, which can hold up to 37 bytes (the maximum S2 line size).
- ***void Comms_Reset_Reg(void)*** – Communications Reset Registers routine. This function is called when the bootloader is ready to jump to the main application. The function must ensure that all the registers that were modified by the driver are returned to their default values. This is to ensure that the main application will receive the MCU on a known state as if it were starting from reset.

The bootloader cannot use interrupts and therefore the Comms_Rx_Data function is polled. On each call, this function must check if new data was received, and if it was, store it on the BootPhraseStructure. This structure has bitfields that match the standard S2 S-Record line. This structure is 37 bytes in size since that is the maximum size that an S2 record can have (without the S2 identifier). The architecture of the BootPhraseStructure allows access to either the individual byte level via an array or via a structure that has bitfields defined. It is expected that the communications driver writes the information as an array, since there is no need to interpret the information on the data reception phase. The skeleton of the BootPhraseStructure is:

```
typedef union
{
     UINT8 Byte[37];    /* Byte level access to the Phrase */
     Struct
     {
     UINT8 PhraseSize; /* Phrase size (wihtout including the size on the count) */
     UINT8 PhraseAddress[3]; /* Address, on S12Z devices its 24-bit addressing */
     UINT8 PhraseData[33]; /* Maximum 32 data bytes. The last byte is the CRC */
     }F;
}BootPhraseStruct;
```

The bootloader application creates an instance of the BootPhraseStruct structure called ‘BP’. Once a complete S2 line was received and stored on the ‘BP’ structure, the communications driver must set the ‘u8BootPhraseRcvd’ flag. Note that ‘BP’ and ‘u8BootPhraseRcvd’ are external variables that the communications driver must modify. The bootloader will check if the ‘u8BootPhraseRcvd’ flag is set and if it is, the flag is cleared and the received data on ‘BP’ is processed for writing. After the data has been written, the bootloader will call the Comms_Tx_Ack function signaling the S-Record transmitter to send a new S2 line. Note that the ‘BP’ structure does not need to be completely filled to be processed. S2 records can be of different sizes. The bootloader will only use the data size indicated by the S2 record itself (the size byte). The communications driver must check if the current S2 line has finished being transmitted by checking the ‘size’ field. There is no signal from the transmitter to indicate that a line transmission was finished.

4.3 Memory erase

When the first S2 line is received, the checksum is verified, and if it matches, the P-Flash memory is erased. Calling the erase P-Flash function will erase all the sectors of the P-Flash (except the sectors where the bootloader is). The erase process includes the last sector, which holds the memory protection bytes, interrupt, and startup vectors. The bootloader must be placed on the last sectors of the P-Flash array (right before the last sector). Table below indicates the sector use.

Table 5. Bootloader placement on P-Flash sectors

Start Address	End Address	Use	Size [Bytes]
...	...	...	...
0xFFF800	0xFFF9FF	Bootloader. Functions executable from Flash	1024
0xFFFA00	0xFFFBFF		
0xFFFC00	0xFFFDFF	Bootloader. Functions executable from RAM	512
0xFFFE00	0xFFFFFF	Interrupt and Startup Vector. P-Flash memory protection bits	512

The bootloader is separated into two main sections: The functions that must be executed from RAM, and the functions that can be executed from Flash. The only functions that must be executed from RAM are P-Flash Erase and P-Flash Write. The rest of the code, including the communications driver, can be executed from Flash. The bootloader is 1.5 KB in size when including a small SCI communications driver. This size can change depending on how large the implemented communications driver is.

All the P-Flash sectors before and after the bootloader are erased before any information is written. While the regular sectors (the ones before the bootloader) present no special considerations, the last sector requires that some sections are reprogrammed:

- The P-Flash Protection Bytes: When the last sector is erased, these bytes are default to 0xFF. Leaving the FSEC byte as 0xFF will secure the device the next time the MCU is reset. The rest of the protection bytes can be left as 0xFF as that means that write protections are not enabled. The Erase P-Flash function stores these protection bytes before erasing the last sector and rewrites them afterwards. This ensures that the MCU is not secured unintentionally.
- The Startup-Vector: When the last sector is erased, the startup vector defaults to 0xFFFFFF. Leaving the startup vector with this address would prevent the MCU from running any program, not even the bootloader would start. An external reprogramming is necessary to recover the MCU. It is critical that the startup vector is reprogrammed with the bootloader startup address as soon as possible. The Erase P-Flash routine stores the bootloader startup address before erasing the last sector and reprograms the startup vector immediately after.

NOTE

There is a 20.5 ms window from where the last P-Flash sector is erased to when the startup vector is reprogrammed with the bootloader startup address. The user must ensure that power is not lost during this time or the MCU will not be able to run on the next reset. The MCU would need to be reprogrammed externally to be able to run again.

After the P-Flash sectors were erased and the startup vector is reprogrammed, the MCU will return to regular execution from Flash and will prepare to write the first S2 line.

4.4 Flash programming

After an S2 record is received and the checksum is validated, the bootloader will prepare for the first write. As mentioned on section 3 “S-Record Format”, each S2 line is composed of three address bytes and at most thirty-two data bytes. Since the FMTRZ allows a maximum (and minimum) write of 8 bytes at a time, a single S2 record can be divided up to four separate write instructions. Each P-Flash write instruction must be aligned to an 8-byte boundary to be successful. This means that the FTMRZ will only write to addresses whose last three bits are 0. Each write instruction will write 8 consecutive bytes starting from this address.

Example: A user wishes to write 0xAC to address 0xFFFE12

In this example, a direct write to 0xFFFE12 would generate an error because this address is not aligned to an 8-byte boundary (the last three bits are 0b010). The user must first find the 8-byte aligned address that corresponds to the memory address of interest, in this case 0xFFFE10. With the 8-byte boundary aligned address, the user can write to any of the 8 bytes consecutive to that address. In this case, the user would need to write:

This memory write modified the values of all the 8 bytes consecutive to the target address. In this case all these bytes where written with 0xFF. After writing, these bytes can’t be modified again unless the whole sector is erased.

In the bootloader application this process is automated and the data from the BootPhraseStruct transferred byte by byte to a smaller structure called the “ProgramStruct”. The ProgramStruct contains an 8-byte aligned 32 bit address and the 8 data bytes that must be written to that address:

```
typedef struct
{
         UINT8 Data[8];
         union
         {
             UINT8 Byte[4]; /* Byte level access to the Address */
             UINT32 DWord; /* DWord level access to the Address */
         } Add;
}ProgramStruct;
```

The bootloader application uses an instance of the ProgramStruct called PS. The PS structure is used by the P-Flash program routine to write data to Flash. The P-Flash Program routine must be executed from RAM, just as the P-Flash Erase routine. Once the programming is finished, execution returns to flash code and fills PS with the next data. The program routine is called either when the PS structure is full (the 8th byte value has been loaded) or when the BootPhraseStruct data finishes. All the bytes that don’t have an explicit value are programmed with 0xFF.

When the bootloader application attempts to write to the last address (the startup vector), it is considered that the programming has finished. This last write is ignored because the startup vector is immediately reprogrammed with the bootloader startup address immediately after the P-Flash is erased. This address can’t be reprogrammed unless erasing the complete sector.

4.5 Jump to main application

The bootloader application jumps to the main user application on two situations: timeout or attempt to write the startup vector. At reset, the MCU goes to the startup vector and looks for the address of the entry function. Since this address is always reprogrammed with the bootloader, the jump to the main application is done by creating a pointer to a programmable fixed address. Before jumping to the main application, some house-keeping activities are required.

To ensure that the main application receives the MCU in a known and expected state, the bootloader must return all the modified registers to its initial value. The bootloader code returns the registers from the CPMU, RTI, FTMRZ, and GPIO to its initial values. The communications driver must implement this register resetting functionality on the routine called "Comms_Reset_Reg" as described in Data reception.

After the communications drivers reset register routine returns, the MCU jumps to the pre-programmed pointer and starts executing code there. The next section describes how to configure this startup address and many other customizations.

5 Implementing the MagniV Bootloader

The MagniV Bootloader was designed to be scalable and easy to migrate between devices with an S12Z core. When migrating the bootloader application to a new S12Z device, the user must focus on three areas:

- Set the linker file addresses and sectors
- Review and set the bootloader configuration macros on "main.h"
- Implement or migrate a communications driver

5.1 File structure

The sources to the bootloader (shared with this application note) are separated in two folders: Bootloader and Comms. The "Bootloader" folder includes all the files needed for the actual data processing, memory erasing, and programming. The "Comms" folder includes some example of communications drivers. This folder includes a common "comms.h" file and specific folders implementing communications drivers for different modules and protocols. The "comms.h" folder must not be modified as it defines the interfacing functions between the bootloader and the communications driver. The "comms.c" file is the actual implementation of the data reception and transmission driver. This file can and must be implemented by the user depending on the communication protocol, channel, and other specific configurations for an application in particular. Figure 5 is an example of the bootloader application file structure on a CodeWarrior project.

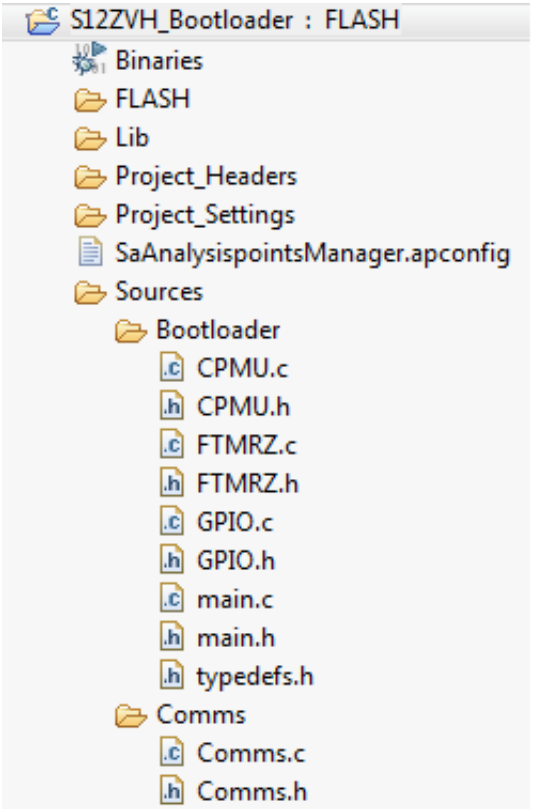

Figure 5. S12Z MagniV Bootloader file structure example

The Comms folder is reserved for the communications driver that can be much more complicated than only two files. The only restriction is that the functions defined on “Comms.h” must be implemented on the communications driver.

5.2 Linker file configuration

The linker file of the bootloader application must be modified. The new linker file must make sure the bootloader code is separated into two groups, one for functions executed from Flash, and one for functions executed from RAM. These sections are called ROM and SHADOW_ROM_S respectively. The address of these sectors must be modified so that they are placed at the end of the P-Flash array (right before the interrupt vector), refer to Table 5. SHADOW_ROM_S must be one sector long (512 bytes), while the ROM size may vary depending on the communications driver size (1k by default).

The RAM memory also needs to be separated because the functions on SHADOW_ROM_S need to be copied to RAM as well. The SHADOW_RAM_S sector needs to be created and must be of the same size as its ROM counterpart (512 bytes). The linker file (the .prm file on CodeWarrior) should have the following layout:

```
NAMES END
SEGMENTS
  SHADOW_RAM_S = READ_WRITE 0x001000 TO 0x0011FF; // 512. Functions executed on RAM
  RAM = READ_WRITE 0x001200 TO 0x0013FF; // Rest for general RAM purposes

  EEPROM = READ_ONLY 0x100000 TO 0x10007F;

  ROM = READ_ONLY 0xFFF800 TO 0xFFFBFF; // 1K. Functions executed on Flash
  SHADOW_ROM_S = READ_ONLY 0xFFFC00 TO 0xFFFDFF; // 512. Functions executed on RAM
END

PLACEMENT
  _PRESTART, /* Used in HIWARE format: jump to _Startup at the code start*/
  STARTUP, /* startup data structures */
```

```
  ROM_VAR, /* constant variables */
  STRINGS, /* string literals */
  VIRTUAL_TABLE_SEGMENT, /* C++ virtual table segment */
  NON_BANKED, /* runtime routines which must not be banked */
  DEFAULT_ROM,
  COPY INTO ROM;
  SHADOW_ROM INTO SHADOW_ROM_S;

  SSTACK, /* allocate stack first to avoid overwriting variables on
  DEFAULT_RAM INTO RAM;
  SHADOW_RAM INTO SHADOW_RAM_S;
END

ENTRIES
END

STACKSIZE 0x100

VECTOR 0 _Startup /* reset vector: this is the default entry point for a C/C++ application.
```

Note that the PLACEMENT section includes two new lines, one that maps the SHADOW_ROM into SHADOW_ROM_S and another one that maps SHADOW_RAM into SHADOW_RAM_S. This ensures that the corresponding variables and functions are stored on the corresponding memory sector.

5.3 Bootloader configuration macros

Once the linker file is configured, the used can proceed to update the bootloader application macros on the “main.h” file. There are 11 macros that configure the bootloader application according to each device and application. These macros are separated into eight groups:

- Oscillator Frequency:
  - OSCILLATOR_FREQUENCY: This macro indicates the frequency of the oscillator on the target board in MHz. The possible values are 4, 8, 12, and 16. This value is used to configure the PLL to 32 MHz BusClk.
- Timeout:
  - TIMEOUT: This is the time the bootloader waits for a valid S-Record after POR. This value is in hundreds of milliseconds. A TIMEOUT of 1 will wait 100 ms before jumping to the main application.
- Erase Flash Macros:
  - FLASH_START_ADD: This is the address (in hex) of the first byte of the P-Flash array.
  - FLASH_SECTORS_TO_ERASE: This is the number of sectors to erase. Consider that a flash sector is 512 bytes. The sectors including the bootloader and the interrupt vector must not be counted here.

$$FLASH\_SECTORS\_TO\_ERASE = Total\_Sec - SR\_Sec - R\_Sec - IntVec\_Sec$$

Where:
Total_Sec: Total number of sectors on the P-Flash. One sector is 512 bytes.
SR_Sec: Number of sectors on the SHADOW_ROM_S segment. Must be 1.
R_Sec: Number of sectors on the ROM segment. 2 by default but may vary.
IntVec_Sec: Number of sectors for the Interrupt Vector. Must be 1.

- Shadow Segments Macros
  - SHADOW_ROM_ADD: This is the address (in hex) of the first byte of the shadow ROM sector.
  - SHADOW_RAM_ADD: This is the address (in hex) of the first byte of the shadow RAM sector.
- GPIO Macros for Visual indication
  - ACTIVITY_LED_ENABLE: Enable/Disable of an LED that toggles each time an S2 record is received. Used to indicate activity. Possible values 1 or 0. If this is 0 (disabled), the rest of the GPIO Macros don’t care.
  - ACTIVITY_LED_DDR: Pointer to the Data Direction Register of the LED that will be activity indicator. Example DDRP_DDRP0.

    - ACTIVITY_LED: Pointer to the Data register of the LED that will be activity indicator. Example PTP_PTP0.
    - LED_ON: Digital value that would turn the LED on. 1 for LEDs in source configuration, 0 for LEDs on sink configuration.
- Checksum Check
    - CHECK_PHRASE_CHECKSUM: Enable/Disable of the checksum calculation and compare routine. Disable this for communication protocols that ensure data delivery. When the CHECK_PHRASE_CHECKSUM is disabled, the check routine will always return OK.
- Application Startup Vector
    - APPLICATION_START_ADD: Address of the Main Application startup vector. The bootloader will jump to this address when a timeout occurs or when a new program has been loaded to the P-Flash. Once the bootloader is loaded to a target, this address will not be modifiable (unless the bootloader is recompiled and reprogrammed on the target). All the application programs loaded on this target MUST start at this address to be able to run. It is recommended that the startup address is the first address of the P-Flash as a common practice.

6 Developing code for the MagniV Bootloader

Once the bootloader application is loaded on a target, it will be ready to receive S-Record files and reprogram the P-Flash multiple times. Once a new program is written to the P-Flash and the bootloader is ready to jump to the main application, all the MCU resources are released. The main application has access to all the RAM memory and MCU peripherals.

Applications designed to run on an MCU with the MagniV Bootloader must take care of two aspects:

- Do not overwrite the bootloader: The user must ensure that the sectors where the bootloader is located are not used on the main application. This can be avoided by modifying the ROM segment on the main application's linker file. Set the ROM segment finish address to be just before the first bootloader sector.
- Match the startup function address with the preprogrammed startup address: The main application entry address must match the one preprogrammed on the bootloader already loaded to the target. It is recommended that the bootloader uses the first address of the P-Flash as the entry point to have a common practice, although any address on the P-Flash can be pre-programmed as the main application startup address.

Failing to meet any of these requirements will not damage or erase the bootloader on the target. In these cases, the main application will not run correctly but it can always be reprogrammed by loading a new S-Record file. If a user wishes to start the bootloader execution from the main application, it is recommended to force a reset by software. This is commonly done by configuring the watchdog and providing an incorrect reset value. The watchdog module will immediately issue a reset and the bootloader application will start execution.

Once the application is ready, it needs to be compiled and linked into an S-Record file. Most compilers can generate S-Record files directly after compilation. The user must ensure that S-Record generation is enabled on the compiler configurations.

7 PC Application program

As part of the files shared with this application note is a PC Application that can be used to stream an S-Record file via RS-232 (SCI). This application is named S12Z MagniV Serial Bootloader Interface. It allows to select the COM port to be used for communications and also allows to specify the path of the S-Record file. Any text file can be used as an input file because there is no extension check. Note that the S12Z MagniV Bootloader expects only S2 lines, so the application will only transmit text lines that begin with 'S2', any other lines are ignored.

The PC application expects an acknowledgment from the target, indicating if the last sent line was processed correctly or a checksum error was detected. If there was a checksum error, the last line is retransmitted. When a line was received correctly, the next line is shared.

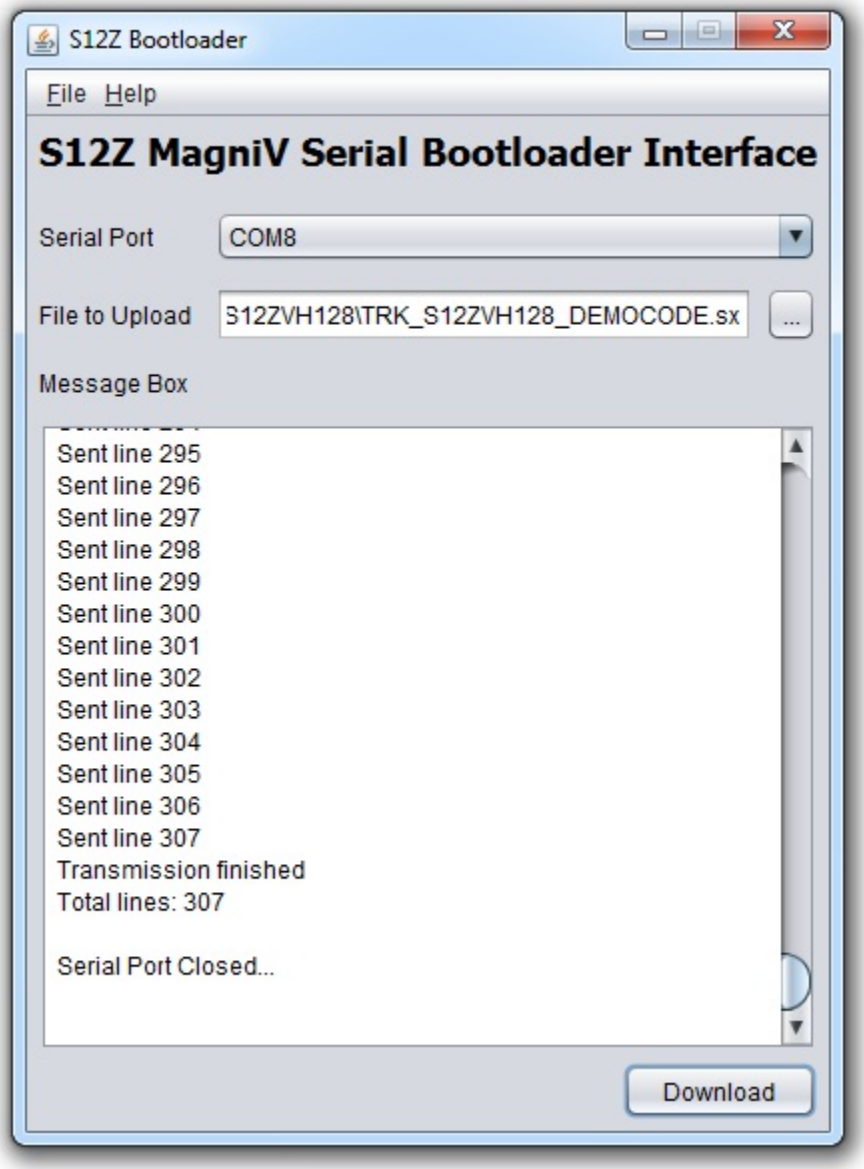

Figure 6. Screenshot of the S12Z MagniV Serial Bootloader Interface

The S12Z MagniV Serial Bootloader Interface is configured to communicate at 19,200 bauds with one stop bit and no parity. These configurations are not user-selectable. The application was developed on Java and requires the Java Runtime Environment (JRE) to be able to run. Currently this application only works on the Windows operating system because it uses the win32com.dll developed by Microsoft.

8 Conclusions

The S12Z MagniV Bootloader application is a scalable, cross-platform project that allows users to easily and quickly implement a bootloader for their application. The clear differentiation between the bootloader and the communications driver provides flexibility to be implemented on several different applications and devices without major modification to the code. The bootloader can run on a device with less than 1 K of RAM and uses in average 1.5 K of ROM space. When not in use it does not consume MCU resources leaving full access to the main user application. This software application and the PC

interface are provided by Freescale as it is. They are intended as demonstration code for development purposes only. The code shared is not intended to be used on production level applications and Freescale does not warrantee its proper functioning on such cases.

9 References

Please consult the latest MagniV Devices, news, resources, reference designs, and other information at: http://www.freescale.com/MagniV

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number AN4723
Revision 0, 07/2013