



# Signal Ranger DSP board

SP2 version



**Bruno Paillard**

Rev 01      September 28, 2002

<b>INTRODUCTION .....</b>	<b>1</b>
<b>INSTALLATION.....</b>	<b>1</b>
<b>BEHAVIOUR OF THE SP2 VERSION .....</b>	<b>1</b>
<b>BOOTING FROM THE FLASH.....</b>	<b>2</b>
Boot table .....	3
Limitations on the code loaded in the boot Flash .....	3
<b>DETAILS OF THE FLASH INTERFACE WITH THE DSP .....</b>	<b>4</b>
<b>ACCESSING THE FLASH FOR GENERAL-PURPOSE USE.....</b>	<b>4</b>
<b>DSP FLASH DRIVER .....</b>	<b>5</b>
Overview .....	5
Setup of the driver .....	6
<b>Data structures .....</b>	<b>6</b>
_FB_WriteFIFO .....	6
_FB_WriteAddress .....	6
_FB_WriteEraseError.....	6
<b>User Functions .....</b>	<b>7</b>
unsigned short _FB_Init().....	7
unsigned short _FB_SetAddress(unsigned short FB_WAddress) .....	7
unsigned short _FB_FIFOState(unsigned short FB_FIFOCount, unsigned short FB_WAddress) .....	7
Void _FB_ErrorClear() .....	7
short _FB_Read(unsigned short FB_RAddress).....	7
unsigned short _FB_SectorErase(unsigned short FB_WAddress) .....	8
unsigned short _FB_EraseAll() .....	8
unsigned short _FB_WritePrepare(unsigned short FB_WAddress, unsigned short FB_WSize).....	8
unsigned short _FB_Write(short Data).....	8
<b>LABVIEW FLASH PROGRAMMING INTERFACE .....</b>	<b>9</b>
FB_Init.vi .....	9
FB_Write_Prepare.vi.....	9
FB_Write_Complete.vi .....	9
<b>SELF-BOOT CODE EXAMPLE.....</b>	<b>10</b>

# Introduction

The SP2 (Self-Powered-Mk2) version of the Signal Ranger board was developed to not only allow the operation of the DSP even when the PC is powered off, or when the USB cable is disconnected from the board, but also to allow the DSP to boot from code in a Flash ROM, with no USB connection present.

With this modification, and if a properly programmed “SR\_FlashBoot” board is present in the expansion connectors of Signal Ranger, the code it contains is downloaded into RAM when the board is powered, even if no USB connection is present at power-up.

At all times, whether the DSP has booted up from Flash ROM or not, the PC can gain control of the DSP through the USB connection. This connection may be used to reset the DSP, force the reload and execution of a new user application, or simply to communicate with the user application that has booted from the Flash ROM, without resetting or stopping it in any way.

**Note:** *Because this board enumerates as a self-powered device, it is important to note that it may not be powered from the USB cable. A separate power supply adapter is provided for this purpose.*

The SP2 version of Signal Ranger must be powered with a 5V supply using a power connector on the SR\_FlashBoot board.

# Installation

For installation instructions go to the first section of the “SignalRanger\_UserManual.pdf” or “SignalRanger\_Manuel.pdf” documents.

All the software of the SP2 version of the Signal Ranger board is now included in revision 1.5 of the Signal Ranger software. If you have a previous version of the software, simply uninstall it and reinstall version 1.5. To obtain version 1.5, contact Soft-dB at: [www.softdb.com](http://www.softdb.com).

# Behaviour of the SP2 version

The Signal Ranger SP2 board enumerates directly as a “Signal Ranger SP (Self-Powered)”. Therefore it uses the same Windows driver as the SP version. The behavior of the SP2 configuration is slightly different from the fully reenumerated SP version:

The SP2 version enumerates as a self-powered device. The board must be externally powered for the PC to be able to detect and enumerate it. From the moment it is powered, the USB controller powers the DSP section and downloads the special “SRKernel\_FB” kernel. Therefore the LED turns green even with no PC connection.

From this point on, the USB controller and DSP section will stay alive even if the PC is disconnected from the board or put in stand-by or hibernation. The DSP section is never powered down or reset, unless specifically requested by the user application, using the corresponding USB vendor request. Note that the Mini-debugger application can be used to reset and take control of the DSP.

A disconnection-reconnection of the USB cable is slightly different from a stand by-resume event. In the case of a disconnection, the PC dumps the driver that allows control and communications with the board when it detects the disconnection. It loads a new instance of the driver when it detects the reconnection. In the case of a stand by-resume event, the same driver usually stays

loaded in PC memory and functional. In both cases the DSP stays powered and is not reset through the event.

When the PC resumes from stand-by or hibernation, or the USB cable is reconnected, the PC is normally able to regain the communication with the board. However, the behavior of the PC when resuming from stand-by or hibernation is often inconsistent from one PC/BIOS/Windows version to another. Some systems loose control of the driver when they are awoken from stand-by or hibernation. In this case it may be necessary to disconnect and reconnect the USB cable to regain communication. This forces the PC to dump the present instance of the driver and load a fresh one. These are PC-specific issues that are out of our control.

## Booting from the Flash

When the board is powered-up, the USB controller completely boots from the 24LC64 without any USB connection. It then resets the DSP and loads a special FlashBoot kernel into its RAM. The “SRKernel\_FB” (FlashBoot) kernel is similar to the usual “SRKernel”, except that it first checks if a properly programmed FLASH ROM is present in the program space at address 4000<sub>H</sub>. If no ROM is present, or if the ROM that is present is not programmed, the kernel waits for USB communication from the PC as the regular kernel does. All the functions of the regular “SRKernel” kernel are present at the same addresses in the “SRKernel\_FB” kernel. This way, all PC communications (code download, execution, memory reads and writes) are supported without any code modification on the PC side.

At all times the PC can reset the DSP through the USB controller and force the reload of the regular “SRKernel” kernel. Since the regular kernel does not check for the FlashBoot board, this process bypasses the flash boot and forces the DSP to boot from code downloaded from the PC. This process is used to erase or reprogram the boot flash without the need for a jumper or any physical change on the board.

The only time Signal Ranger can boot from the FLASH ROM is when it is first powered-up. It cannot boot from the FLASH ROM when it is reset from the PC because the PC downloads the regular “SRKernel” kernel then.

The PC can communicate with the DSP code that is running from the FLASH ROM, which is useful to debug it. However to do this the PC must not reset the DSP. For this reason, the new version of the mini-debugger does not reset the DSP and load the kernel when it is launched, as it used to. The RESET button of the mini-debugger is still there though, and may be used to force a DSP reset and kernel reload manually.

Communications are enabled through the “SRKernel\_FB” kernel in exactly the same way as they are with the regular “SR\_Kernel” kernel. This way, DSP and PC code that has been developed and tested through the use of the mini-debugger, or a custom PC interface will run in exactly the same way when the DSP boots from the FLASH ROM.

The load data programmed in the Boot table in FLASH ROM allows the DSP to load any number of sections in any space (Data, Program or IO). The download is only limited by the effective size of the FLASH ROM which is 48Kwords deep (4000<sub>H</sub> – FFFF<sub>H</sub>). The upper space of the FLASH ROM, above 4000<sub>H</sub> is not useable since the on-chip DSP RAM is mapped in program space at these addresses.

After loading all the sections, the DSP branches to the address that is specified as the last word programmed in FLASH ROM. As is the case when the DSP boots from the PC, the DSP code that is launched may be a complete program that never returns, or it can be a simple function that returns after completing its task. In this case the DSP returns to the kernel where it waits for further PC communications.

## Boot table

To allow the DSP to boot from FLASH, the following data structure must be present at address 4000<sub>H</sub> in the program space.

The “SRKernel\_FB” kernel first checks to see if the magic number 3009<sub>H</sub> is present at address 4000<sub>H</sub>. If it is, it then interprets the remainder of the table as described below. No checks are performed to insure that the data in the table makes sense. In all likelihood, if the magic number is correct but the rest of the data has not been programmed or is inconsistent, the DSP will crash during the boot procedure.

Address	Data	Description
4000 <sub>H</sub>	3009 <sub>H</sub>	Magic Number
4001 <sub>H</sub>	Nb_Sections	Total Number of sections to load into RAM
4002	Section Space 1 <sup>st</sup> Section	Memory space where to load the first section
4003 <sub>H</sub>	Section Length 1 <sup>st</sup> Section	Number of words that follow
4004 <sub>H</sub>	Address 1 <sup>st</sup> Section	Load address for the following section
4005 <sub>H</sub>	1 <sup>st</sup> Word 1 <sup>st</sup> Section	Data words
4006 <sub>H</sub>	2 <sup>nd</sup> Word 1 <sup>st</sup> Section	...
4007 <sub>H</sub>	...	...
...	...	...
...	Last Word 1 <sup>st</sup> Section	...
...	Section Space 2 <sup>nd</sup> Section	Memory space where to load the second section
...	Section Length 2 <sup>nd</sup> Section	Number of words that follow
...	Address 2 <sup>nd</sup> Section	Load address for the following section
...	1 <sup>st</sup> Word 2 <sup>nd</sup> Section	Data word
...	2 <sup>nd</sup> Word 2 <sup>nd</sup> Section	Data word
...	...	...
...	Entry Point	After loading the various sections, the DSP transfers execution to this address.

**Table 1**

## Limitations on the code loaded in the boot Flash

Due to the fact that the boot Flash occupies the program space above 4000<sub>H</sub> and is limited to 48K in length, the following restrictions apply to the executable code loaded into the boot Flash:

1. The total length of the loaded code and initialized variables must be less than 48K words. In fact the boot table includes additional data such as section space, section length, load addresses... etc., which further limit the total length of the loaded code slightly.
2. No section should be loaded in program space above address 4000<sub>H</sub> because this is where the boot Flash is located.
3. Sections may be loaded anywhere in data space, below 4000<sub>H</sub> in program space, and anywhere in IO space. Loading sections in IO space may be done to initialize peripherals that may be located there, prior to code execution. Sections above 4000<sub>H</sub> in data space are loaded in external RAM.

Most of these restrictions also applied to the code that was written to be loaded from the PC in the standard version of the board. The only exception is restriction No 1, which limits the total size of the loadable code and initialized data to 48Kwords. Previously the total size of the code and initialized data was only limited to 64Kwords which represented the total RAM available on board).

## Details of the Flash interface with the DSP

The Flash circuit is interfaced with the DSP in its program space. The Flash ROM circuit that is used is an AM29LV200BB-120EC from AMD. This is a 128Kwords deep device, but only the first 64K are used (the A16 address bit of the Flash ROM is grounded). Also, since the 16Kwords of on-chip DSP RAM are mapped in the first 16K of the program space (addresses 0000<sub>H</sub> to 3FFF<sub>H</sub>) only the remaining 48Kwords of Flash are actually accessible. The access time is 120ns, the Flash ROM requires 14 wait-states to be accessed. The typical programming time is 11 $\mu$ s per word.

The schematic of the interface is detailed in the SR\_FlashBootSchem.pdf document.

Since only the bottom 48Kwords of the Flash are visible, the Flash is actually seen as two sectors. The first sector extends from 4000<sub>H</sub> to 7FFF<sub>H</sub>. The second sector extends from 8000<sub>H</sub> to FFFF<sub>H</sub>.

The INT1 interrupt is connected to the (inverted) Ry/By pin of the Flash ROM. This way the DSP receives an INT1 interrupt whenever the current program or erase operation is completed. Note that after receiving the interrupt the INT1 line stays low until the next program or erase operation is started.

The reset line of the Flash ROM is connected to the reset line of the DSP. This way the Flash ROM is properly reset at power-up, or at any time the DSP is reset under control of the PC.

## Accessing the Flash for general-purpose use.

The Flash ROM can be used by the user DSP code to store user data or parameters. This use is simplified by the fact that a DSP driver is provided. User DSP code can be linked with the functions of this driver to read, write or erase the Flash ROM. On the PC-side, a Flash management LabVIEW library is also provided. Functions from this library are used in the mini-debugger application, which provides a good starting example for the developer.

When using the Flash for general purpose, the user should be careful not to erase or reprogram the boot sector (the first sector starting at address 4000<sub>H</sub>) if one exists. If the DSP code must absolutely make modifications in the first sector of the Flash ROM any boot code that appears there should be temporarily stored and reprogrammed with the user data.

The Flash program and erase cycles normally include write cycles at addresses 0555<sub>H</sub> and 02AA<sub>H</sub>. Write cycles at these addresses won't reach the Flash ROM because they are mapped into the DSP's on-chip ROM. Consequently they will not even appear on the bus. To properly program or erase the Flash, addresses 8555<sub>H</sub> and 82AA<sub>H</sub> must be substituted to addresses

0555<sub>H</sub> and 02AA<sub>H</sub>. These addresses work as well since the Bit No15 of the Flash ROM is not decoded internally for write and erase cycles.

## DSP Flash Driver

A DSP driver is provided to help the user's DSP code development. This driver takes the form of a library (**FB\_Driver.lib**). It is used in the **FB\_Write.out** DSP code that forms the basis of the LabVIEW **FlashWrite.lib** library. This library is itself used in the Mini-Debugger application. All this code is open at all levels, and forms a good starting point for the user who wants to use the Flash.

The source code for the driver can be found in C:\Program Files\SignalRanger\Sources\FlashDriver.exe.exe. Running this executable will create the whole project directory for Code Composer Studio.

The source code for the **FB\_Write.out** executable that supports the operation of the **FlashWrite.lib** LabVIEW library that is used to program the Flash in the mini-debugger can be found in C:\Program Files\SignalRanger\Sources\FlashWrite\_DSP.exe. Running this executable will create the whole project directory for Code Composer Studio.

The driver is composed of C-callable functions, as well as appropriate data structures.

The functions allow read, erasure and sequential write accesses to the FLASH. It uses a write FIFO buffer, so that the write functions do not have to wait for each write operation to be completed.

Read functions are performed asynchronously and are very fast. Writes are sequential and are performed under INT1 interrupt. The typical write time is 11  $\mu$ s per word.

## Overview

Read, write and erase addresses are 16 bits.

Reads are very simple. They are performed asynchronously using the **\_FB\_Read** function. This function returns the content of any 16-bit address.

Writes are performed sequentially using the **\_FB\_Write** function. Writes are performed at addresses defined in the **FB\_WriteAddress** register. This register is not user-accessible. It must be initialized before the first write of a sequence, and is automatically incremented after each write. The **FB\_WriteAddress** register can be initialized using the **\_FB\_SetAddress** function, or the **FB\_WritePrepare** function.

A write operation can turn ones into zeros, but cannot turn zeros back into ones. Normally, a sector of Flash should be erased before any write is attempted in this sector. Several functions are provided for that purpose:

The **\_FB\_SectorErase** function erases the sector containing the specified address.

The **\_FB\_EraseAll** function erases both sectors of the Flash circuit.

The **\_FB\_WritePrepare** function pre-erases all the sectors starting at the specified address, and containing at least the specified sequential number of words. Because erasure is performed sector by sector, this function may erase more words that are actually specified to the function.

The function then initializes the **FB\_WriteAddress** register to the beginning address specified, so that the next write is performed at the beginning of the specified memory segment.

The **\_FB\_Write** function does not wait for the write to be completed. It just places the word to be written into the **FB\_WriteFIFO** buffer and returns. The writes are actually performed under interrupt control, without intervention from the user code.

The fill state of the write FIFO, as well as the state of write and erase errors can be monitored using the `_FB_FIFOState` function.

## Setup of the driver

All the functions of the driver that are defined below are contained in the `FB_Driver.lib` library. The user code must be linked with this library to function properly (the library must be added to the project source files).

Since the writes are performed under INT1 interrupts, the INT1 interrupt vector must be initialized to the “`_FBINT`” label. This label is the entry point of the INT1 interrupt routine. The INT1 interrupt routine is defined in the `FB_Driver.lib` library.

## Data structures

### `_FB_WriteFIFO`

`_FB_WriteFIFO` is a 32-words buffer accessed with FIFO access logic. The FIFO itself is not user-accessible. It may only be written by the `_FB_Write` function. It is only emptied by the INT1 interrupt function.

### `_FB_WriteAddress`

`_FB_WriteAddress` is a 16-bit unsigned word that always contains the address of the next word to be written. `_FB_WriteAddress` can be initialized by the `_FB_SetAddress` function or the `_FB_WritePrepare` function. After each write completes, the `_FB_WriteAddress` register is automatically incremented. This increment happens in the INT1 interrupt routine. Therefore to the user code the value always indicates the address for the next write, never the value of the write that is in progress.

The current value of the `_FB_WriteAddress` register can be read with the `_FB_FIFOState` function.

### `_FB_WriteEraseError`

`_FB_WriteEraseError` is an integer that contains various error status bits. It is returned by several functions, including `_FB_SetAddress`, `_FB_FIFOState`, `_FB_SectorErase`, `_FB_EraseAll`, `_FB_WritePrepare` and `_FB_Write`.

Once an error bit is set to one, indicating an error, it stays one until the error word is cleared using `_FB_ErrorClear`. Execution of `_FB_Init` also clears the error word.

Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
											MI	CE	SE	WP	WE

**WE**    *Write Error*                    An error occurred during a write. Either a write was attempted at an address that was not previously erased, or the circuit is not functioning properly.

**WP**    *Write in progress*            When it is one, this bit indicates that writes are in progress. This bit is only cleared to 0 when the write FIFO is empty and the last write operation is completed. It is set to one as soon as a new word is written into the write FIFO.

**SE**    *Sector Erase Error*            This bit indicates that an error occurred during the requested sector erase operation. The most likely cause is that the circuit is not functioning properly.

**CE**    *Chip Erase Error*                This bit indicates that an error occurred during the requested chip erase operation. The most likely cause is that the circuit is not functioning properly.

**MI** *Mission Impossible* This bit indicates that the requested operation is not possible. This bit is set to one in the following conditions:

- When a **\_FB\_SectorErase** function is called to erase a sector outside the address range of the board. Note that addresses between 0000<sub>H</sub> and 3FFF<sub>H</sub> are considered outside the address range of the board.
- When a **\_FB\_SetAddress** function is called to initialize the **FB\_WriteAddress** pointer outside the address range of the board.
- When a **\_FB\_SectorErase** function is called to erase a sector outside the address range of the board.
- When a **\_FB\_WritePrepare** function is called to prepare a segment that extends outside the address range of the board.
- When a **\_FB\_Write** extends outside the address range of the board.

## User Functions

### unsigned short **\_FB\_Init()**

Initializes the driver and resets the Flash circuit. It detects the Flash ROM and returns the memory size in words, or zero if the circuit is not present.

This function must be called at least once before any other function of the driver is called. This function may be called to reinitialize the driver.

### unsigned short **\_FB\_SetAddress(unsigned short FB\_WAddress)**

This function waits for all pending writes to complete. Then it resets the **FB\_WriteAddress** register to the 16-bit value passed in argument.

If an address set is attempted outside the possible address range for the board, the **MI** bit is set in the **FB\_WriteEraseError** status word.

The function returns the current **FB\_WriteEraseError** status.

### unsigned short **\_FB\_FIFOState(unsigned short FB\_FIFOCount, unsigned short FB\_WAddress)**

This function returns the number of words still in the write FIFO in the **FB\_FIFOCount** argument, and the present value of the **FB\_WriteAddress** register in the **FB\_WAddress** argument.

The function returns the current **FB\_WriteEraseError** status.

Note: A return value of zero for **FB\_FIFOCount** does not mean that all writes are completed. The last write may still be in progress. To verify that all writes have indeed been completed, the **WP** bit in the **FB\_WriteEraseError** status register should be checked.

### Void **\_FB\_ErrorClear()**

The function clears the current **FB\_WriteEraseError** status register.

### short **\_FB\_Read(unsigned short FB\_RAddress)**

The function returns the word read from the **\_FB\_RAddress** address. Note that no check is performed to insure that the read occurs in the section occupied by the Flash ROM in the program space memory map (4000<sub>H</sub> to FFFF<sub>H</sub>). If a read is attempted between 0000<sub>H</sub> and 3FFF<sub>H</sub>, the function simply returns the contents of the on-chip RAM which is mapped at these addresses in the program space.

### unsigned short \_FB\_SectorErase(unsigned short FB\_WAddress)

The function waits for all pending writes to complete. Then it checks the sector containing the **FB\_Waddress** address to see if it is already erased. If it is already erased it returns. If not it erases the sector, and then it verifies the erasure.

If a sector erase is attempted outside the possible address range for the board, the **MI** bit is set in the **FB\_WriteEraseError** status register.

The function returns the current **FB\_WriteEraseError** status.

The function does not return until the erase is completed. It typically takes 0.7s per sector, but may take up to 15s per sector to complete.

### unsigned short \_FB\_EraseAll()

The function waits for all pending writes to complete. Then it erases all the sectors of all the chips present on the board one by one and verifies the erasure. The erasure proceeds sector by sector, using the **\_FB\_SectorErase** function.

The function returns the current **FB\_WriteEraseError** status.

The function does not return until the erase is completed. It typically takes 1.4s to erase the two accessible sectors.

### unsigned short \_FB\_WritePrepare(unsigned short FB\_WAddress, unsigned short FB\_WSize)

The function pre-erases all the sectors of the Flash circuit, required to write a segment **FB\_WSize** long, from the **FB\_WAddress** address. It then initializes the **FB\_WriteAddress** register to the value of **FB\_WAddress**, so that the next call to **\_FB\_Write** will effectively write at the beginning of the prepared segment.

Because erasure is performed sector by sector only, this function may erase more words that are actually specified. This is the case if **FB\_WAddress** is not an address corresponding to the beginning of a sector, or if **FB\_WAddress + FB\_WSize - 1** is not an address corresponding to the end of a sector.

The function waits for all pending writes to complete before starting the erasure.

If during the preparation a sector erase is attempted outside the possible address range for the board, the **MI** bit is set in the **FB\_WriteEraseError** status word.

The function returns the current **FB\_WriteEraseError** status.

The function does not return until the erasure is completed. The time is dependant on the length of the segment to be prepared. It typically takes 0.7s per sector to erase.

### unsigned short FB\_Write(short Data)

The function places the value of Data in the write FIFO. It normally returns without waiting for the write to be completed. The writes are performed under INT1 interrupts. However, if the FIFO is full when the function is called, the function does wait for a slot to be available in the FIFO, before placing the next value in the FIFO and returning.

It typically takes 11µs per word to program, so if the function is called while the FIFO is full, it may not return before 11 µs have elapsed.

The requested write begins as soon as the previous writes in the FIFO are completed. The data is written at the current value of **FB\_WriteAddress**.

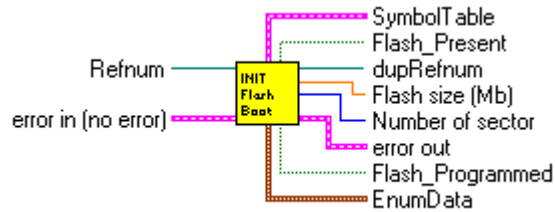
If a write is attempted outside the possible address range for the board, the **MI** bit is set in the **FB\_WriteEraseError** status word.

The function returns the current **FB\_WriteEraseError** status. However, this error word does not reflect the status of the requested write, because the function does not wait for this write to actually begin.

# LabVIEW Flash programming interface

The Flash programming interface takes the form of two LabVIEW libraries named **FB\_Write.Ilb** and **\_FB\_Write\_U.Ilb**. The first one contains three high-level VIs that can be used by the developer and are documented below. The second one contains lower-level VIs that are used by the VIs in the **FB\_Write.Ilb** library. This library requires that the **FB\_Write.out** DSP code be loaded in DSP memory to enable its operation. The VI **FB\_Init.vi** loads this DSP driver in memory and initializes it. The section below describes the VIs in **FB\_Write.Ilb**:

## FB\_Init.vi



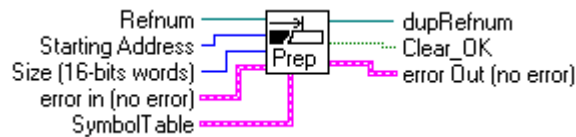
Checks to see if the Flash ROM is present and if the magic number is correct. If the Flash is present, the function then loads the **FB\_Write.out** DSP Flash driver and executes it. If the Flash is not present, the function returns without loading the Flash driver. The VI also returns a Boolean indicating if the magic number at address 4000<sub>H</sub> is programmed with value 3009<sub>H</sub>, a 32-bit integer showing the number of sectors, and a float indicating the size in Mwords.

This VI should be executed prior to executing any of the other VIs in the library.

Note: the board is reset prior to loading the Flash driver.

Note: Loading the Flash driver may corrupt code/data already residing in memory.

## FB\_Write\_Prep.vi



This VI clears as many sectors as needed to be able to write the specified number of words into Flash, starting at the specified address. The clear is performed on a sector-by-sector basis. It is not possible to clear a fractional amount of a sector.

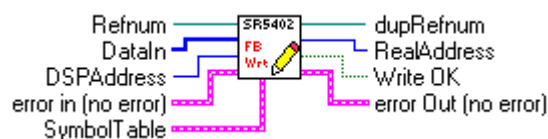
The VI requires **FB\_Init** to be executed successfully before attempting its execution.

Note: executing this VI with a size of zero will clear the sector in which the starting address resides.

This VI has a timeout of 20 seconds. If the clear does not complete within 20 seconds, the VI returns with a false in its "Clear\_OK" output boolean.

The symbol table input must be passed from the **FB\_Init.vi** VI.

## FB\_Write\_Complete.vi



This polymorphic VI writes an unlimited number of data words to the Flash. The contents of the Flash are read back after the write to verify the write. DSPAddress represents the address in Flash to which the transfer should occur (read or write). The address should be above 4000<sub>H</sub>, and the length of the array should be such that no write is attempted past address FFFF<sub>H</sub>. Otherwise the write is not executed.

DataIn can be a scalar or an array of the following types:

- Float (32-bit floating point value)
- U32 (32-bit unsigned integer)
- I32 (32-bit signed integer)
- U16 (16-bit unsigned integer)
- I16 (16-bit signed integer)
- U8 (8-bit unsigned integer)
- I8 (8-bit signed integer)

When 32-bit types are written, the address is aligned to the next even address. Care must be taken because this may cause the write address to be shifted by one position. The output named RealAddress represents the effective write address after the eventual alignment.

When 8-bit types are written, they are written two at a time (the word size for this Flash is 16-bits). The symbol table input must be passed from the FB\_Init.vi VI.

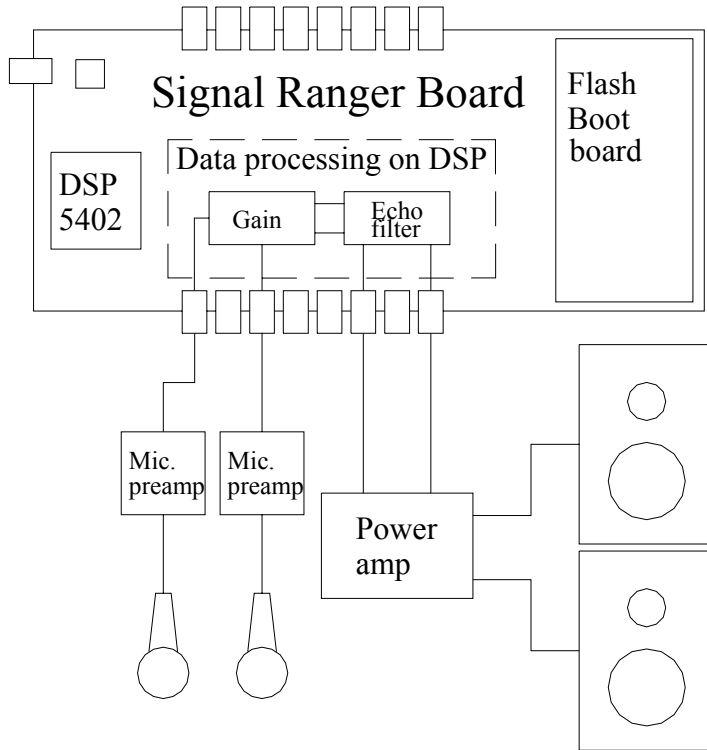
The VI returns with a false in its "WriteOK" boolean if the write fails at some point. This would most likely be the case if a write is attempted without previously clearing the Flash using FB\_Write\_Prepare.vi.

The VI requires FB\_Init to be executed successfully before attempting its execution.

## Self-Boot code example

The application FB\_Demo is a DSP application example that can be loaded into the Flash ROM of the FlashBoot expansion board. With this code loaded into Flash, the demo application will run at power-up on the "SP2" version of Signal Ranger. The DSP code of this example can be found in the folder C:\Program Files\SignalRanger\Sources. Before testing this demo, the file FB\_Demo\_DSP.out must be written into the boot sector of the flash memory using the MiniDebugger interface.

The signal processing implemented on the DSP board is a simple gain and "reverb" filter, as described in figure 1. This processing is performed on two channels. The result of this processing can be heard using the illustrated setup.



**Figure 1**

Even though the application will run without a PC connected, a PC can be connected with the USB cable to change the gain and to observe signal and filter coefficients in real time. These buffers are located in data space at the following addresses, and can easily be displayed (and modified) in real time using the functions of the mini-debugger. In particular the signal can be displayed continuously, using a graph. Gain and filter impulse responses can be changed dynamically while the application is running.

Address	Name	Size	Description
0x300	Gain	1 word	Gain (between 0 and 1) in Q15 representation (0 : 0 and 1 : 32767)
0x2000	Filter	2048 words	2048 coefficients of the filter
0x1500	Buffer0	2048 words	Memory buffer of the input 0 signal
0x700	Buffer1	2048 words	Memory buffer of the input 1 signal

**Table 2**