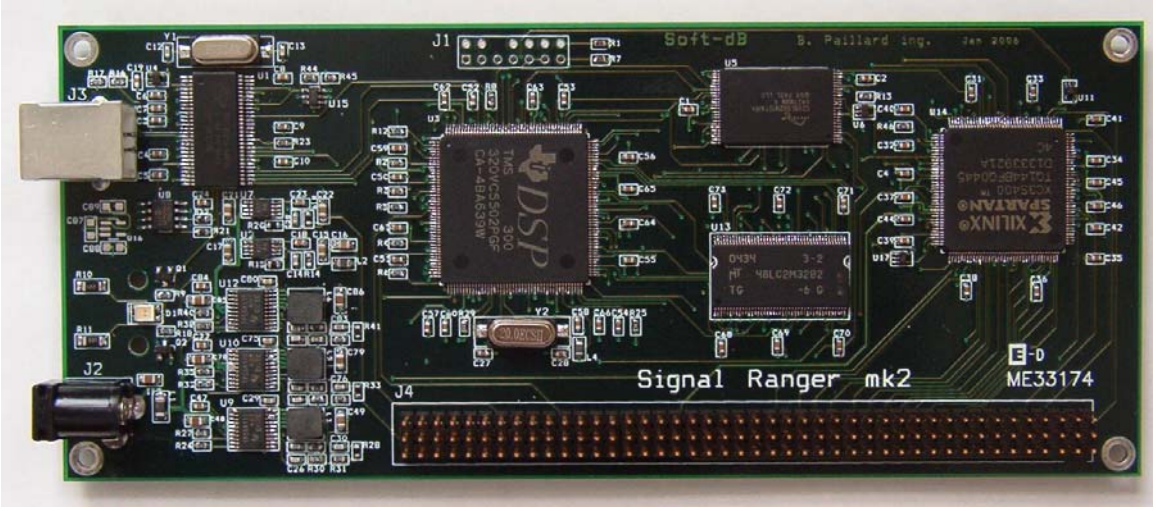
Signal Ranger_mk2 DSP Board

User's Manual



Bruno Paillard

Rev 03 February 28, 2006

Main Features

Signal_Ranger_mk2 is a fixed point DSP board featuring a 300MHz TMS320C5502 DSP, a 400 kgates SPARTAN 3 FPGA and a high-speed USB 2 interface, providing fast communications to the board. The Windows driver allows the connection of any number of boards to a PC.
The DSP board may be used while connected to a PC, providing a means of exchanging data and commands between the PC and DSP in real-time. It may also be used in stand-alone mode, executing embedded DSP code.
Given its very flexible resources (DSP+FPGA) and the fact that it can work as a stand-alone board, the *Signal_Ranger_mk2* board may be used in many applications. With the addition of analog daughter boards, *Signal_Ranger_mk2* covers the following applications with ease:

- Multi-channel speech and audio acquisition and processing.
- Multi-channel control.
- Instrumentation and measurement.
- Vibro-acoustic analysis.
- Acoustic Array processing/Beamforming
- DSP software development.

Architecture And Boot Modes

The *Signal Ranger_mk2* board includes a 1M word Flash ROM, from which the DSP may boot. Furthermore, the Flash may also contain the configuration code of the FPGA, allowing the DSP to initialize the FPGA with its logic as part of the initial boot process.
There are two ways the DSP can boot:

- **Stand-Alone Boot:** At Power-Up, the USB controller in stand-alone configuration takes control of the DSP and FPGA. It loads a communication kernel into DSP RAM and executes it. This kernel then looks in Flash memory for an FPGA logic description file. If it finds it, it loads the FPGA. It then looks further in Flash memory for DSP code. If it finds it, it loads it into RAM and executes it. By pre-programming the Flash memory with FPGA logic and/or DSP code, the board can work in stand-alone mode, executing an embedded DSP application directly from power-up.
- **PC Boot:** After the board has been connected to a PC, the PC can force the DSP to reboot. In this mode, the PC can force the reloading of new FPGA logic and DSP code. This mode may be used to “take control” of the DSP at any time. In particular, it may be used to reprogram the Flash memory in a completely transparent manner, without using any jumpers.

Even when the DSP board has booted in stand-alone mode, a PC can be connected at any time to read/write DSP memory without interrupting the already executing DSP code. These functions may be used to provide real-time visibility into, or send commands to the executing embedded DSP code.

Technical Data:

- **USB**

USB 2.0 PC connection. Average data throughput: 18Mb/s (reads), 22Mb/s (writes). Stand-alone USB controller requires no management from the DSP software.

- **DSP**

TMS320C5502 16-bits fixed point DSP, running at 300 MHz, with 32Kwords of on-chip RAM.

- **FPGA**

XC3S400 FPGA. 400 000 gates. 56 kbits distributed RAM, 288 kbits block RAM, 16 dedicated 18x18 multipliers, 4 DCMs. Provides 63 user-configurable I/Os.

- **Power supply**

Signal Ranger_mk2 is Self-Powered using an external 5V (+-5%) power pack. It can work without any connection to a PC.

- **Memory**
  - 64 kbytes on-chip (DSP) double-access RAM, mapped in data and program spaces.
  - 4 Mbytes external 75MHz Synchronous Dynamic RAM, mapped in data and program space.
  - 2 Mbytes external Flash Rom, mapped in data and program space.

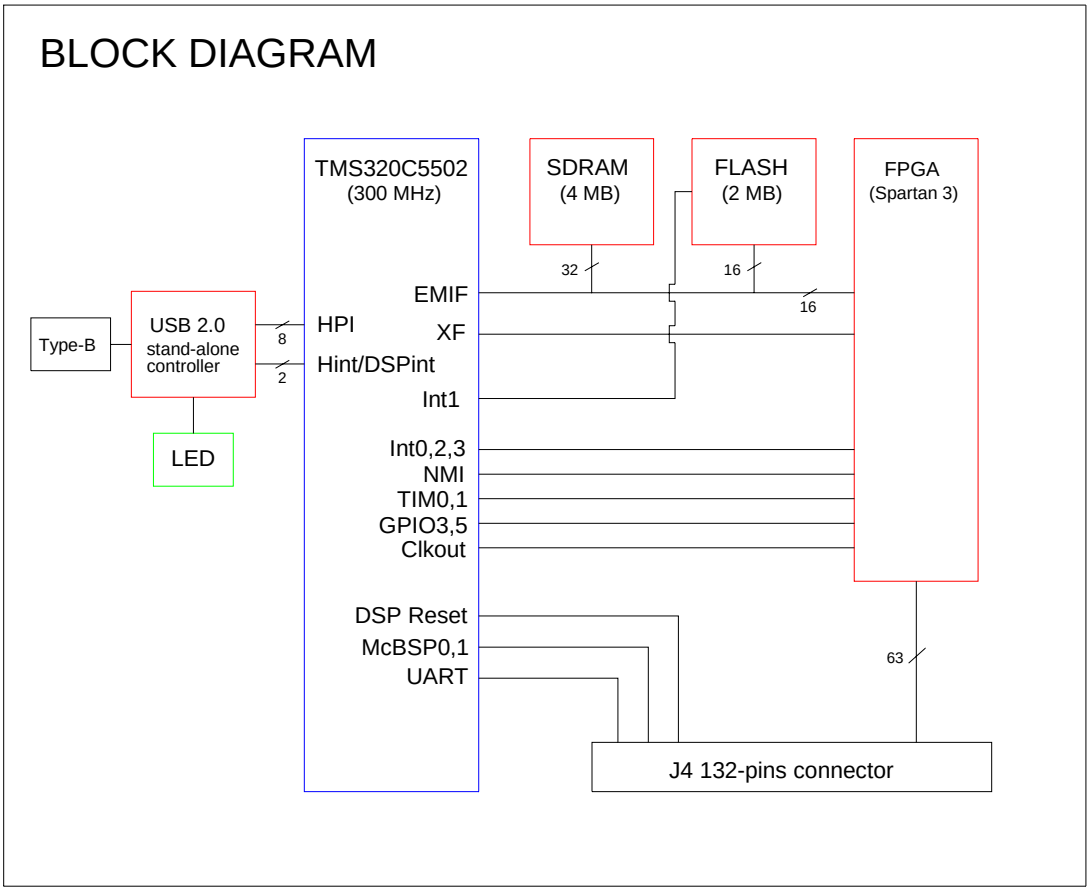


Figure 1: Block Diagram of the Signal Ranger mk2 DSP board

Software:

- **Driver for Win2k and WinXP:**
  This driver allows the connection of any number of boards to the PC.
- **Full-featured symbolic debugger:**
  The debugger includes features such as real-time graphical data plotting, symbolic read/write access to variables, dynamic execution, Flash programming… etc. At its core, the mini-

debugger uses the same interface libraries that a developer uses to design a stand-alone DSP application. This insures a seamless transition from the development/debugging environment to the deployed application.

- **LabView interface:**
  This library of LabView VIs allows the development of LabView code to interface with the DSP board. It includes VIs to download DSP code (COFF loader), launch DSP functions, and read/write DSP memory while the DSP code is executing.
- **C/C++ interface:**
  This DLL allows the development of PC code written in C/C++ to interface with the DSP board. They include functions to download DSP code (COFF loader), launch DSP functions, and read/write DSP memory while the DSP code is executing.
- **SelfTest application:**
  This application tests all the hardware on the DSP board.
- **Code examples:**
  Several demo LabView applications demonstrate the development of DSP code in C and in assembly. They also show how to interface this code to a PC application written in LabView. One demo Visual Studio application demonstrates the development of a PC application written in C/C++.
- **Flash driver and example code:**
  This driver includes all the code to configure and use the on-board 2 Mbytes Flash ROM from within user DSP code.
- **Factory-Default FPGA Configuration:**
  The board is provided with a factory default FPGA configuration, which provides 63 configurable digital I/Os.

Installation

Signal Ranger_mk2 works with Windows 2000 and Windows XP. None of the *Signal Ranger_mk2* software, including the driver, is compatible with the previous *Signal Ranger* software. The new software must be installed.

Note: Do not connect the SignalRanger_mk2 DSP board into the PC's USB port until the software has been installed. The driver installation process, which occurs as soon as the board is connected to the PC, requires that the driver file be present on the PC.

Software Installation

Simply execute Setup.exe, this will install two sets of files:

- The *SignalRanger_mk2* software, including the required libraries, documentation, and demo applications will be installed first. The installer creates a directory named *SignalRanger_mk2* in *C:\Program Files*, and stores all the required files into it. It also creates shortcuts in the Windows START menu.
- The LabVIEW 7.1 run-time engine is installed afterwards. This run-time engine is required to execute the compiled versions of the demo applications. It is also required to use the C/C++ software interface. It is not required to use the LabVIEW software interface. The LabVIEW 7.1 run-time engine is installed in the *C:\Program Files\National Instruments\Shared\LabVIEW Run-Time\7.1* directory.

After this installation has been performed, it is necessary to install the USB driver for the board. For this, refer to the hardware installation section below.

What Is Installed Where?

The installer creates the *C:\Program Files\SignalRanger_mk2* directory. This directory contains all the software tools:

- A directory named *Documentation* containing all the required documentation, including this document.
- A directory named *Driver* containing the driver for the board.
- A directory named *Examples* containing:
  - A directory named *C_Examples* containing a zip file. When deflated, this zip file contains the C/C++ interface demo application, including PC Visual .net project and DSP code.
  - A directory named *LabVIEW_Examples_DSPCode* containing a zip file. When deflated, this zip file contains the DSP code of the LabVIEW examples as well as documentation. The LabVIEW code of these examples is in the main install directory, in the DemoLabview.llb library.
- All the LabVIEW libraries that are described in this document.
- The *SRanger2.dll* DLL, which is the lowest level of interface, and is required by all other levels of interface.
- All the DSP executable files (".out") required by the interfaces. This includes both kernels, as well as the Flash and FPGA support code.
- The *SR2_SelfTest.rbt* file, which contains the factory-default FPGA logic.
- The *Revision_History.txt* file, which details the revision history.

Hardware Installation

Note: Only power the board using the provided power-supply, or using a 5V (+-5%) power supply. When using a custom power supply, make sure that the positive side of the supply is in the center of the plug. Failure to use a proper power supply may damage the board.

- Power-up the board from the 5V adapter first.
- The LED should light-up red for 1/2s, to indicate that the board is properly powered, then orange to indicate that the DSP section is functional.
- Connect the *SignalRanger_mk2* board into the USB port of the PC.
- After a few seconds, Windows should detect the new board and present a standard driver installation wizard.
- Make the proper selections to specify the location of the driver.
- When asked to, navigate to the directory *C:\Program Files\SignalRanger_mk2\Driver*, and select the file *SRm2.inf*.
- Windows should install the driver.
- At this time the LED turns green to indicate that the PC is communicating with the board.
- After this first installation, the PC should always recognize the board automatically a few seconds after it is connected. It may take more time if the board is connected into a different USB root or hub on the PC. In that case it is possible that the PC indicate that a new device has been found. However it should be able to find the driver automatically.
- At any time after the board has been connected to the PC, and the PC has recognized it, the LED should be green. The LED must be green before attempting to run any PC application that communicates with the board.

What To Do In Case The Driver Installation Fails

To do a manual driver installation, follow these steps:

- Power-up the board and connect it to the PC.
- Go to the START menu, under *Settings\Control Panel\System*.
- Select the *Hardware* tab.
- Press the *Device Manager* button.
- Go to the *Universal Serial Bus Controllers* item and expand it.

- There should be an *Unknown Device* item in the tree.
- Right-click on the *Unknown Device* icon.
- Press on *Update Driver.*
- Then follow the driver installation procedure described above.

Led Indicator

The LED of the Signal Ranger_mk2 board has the following behaviour:

- It lights up red when the 5V power is first applied to the board. This indicates that the board is properly powered.
- It turns orange on its own 1/2s after the board has been powered-up. This indicates that the board went through its proper reset and initialization sequence. If boot DSP code was previously programmed in the Flash ROM, this code is running when the LED is orange.
- It turns green whenever the board has been connected to the PC, and the PC has loaded its driver. The green LED indicates that the PC is able to communicate with the board. The LED must be green before attempting to execute any PC application that communicates with the board.
- The LED turns orange whenever the USB connection is interrupted. This is the case when the PC is turned off or goes into stand-by, or if the USB cable is pulled from the board or the PC. However, the LED turning orange does not mean that the DSP code has stopped running.
- The LED also may turn orange temporarily when the board is being initialized, or the FPGA is reloaded by a PC application.
- The LED colour may be changed at any time by a user application.

Note: This LED behaviour is different from the LED behaviour of the Signal Ranger SP2 board. In the latter case, the LED turning orange indicated PC enumeration, while the LED turning green indicated DSP initialization. The meaning of the two colours has been reversed in the Signal Ranger_mk2 board

Testing The Board

At any point after the board has been powered-up and connected to a PC (after the LED has turned green), the *SR2_Self_Test* application may be run.

The user interface of the SelfTest application is given below:

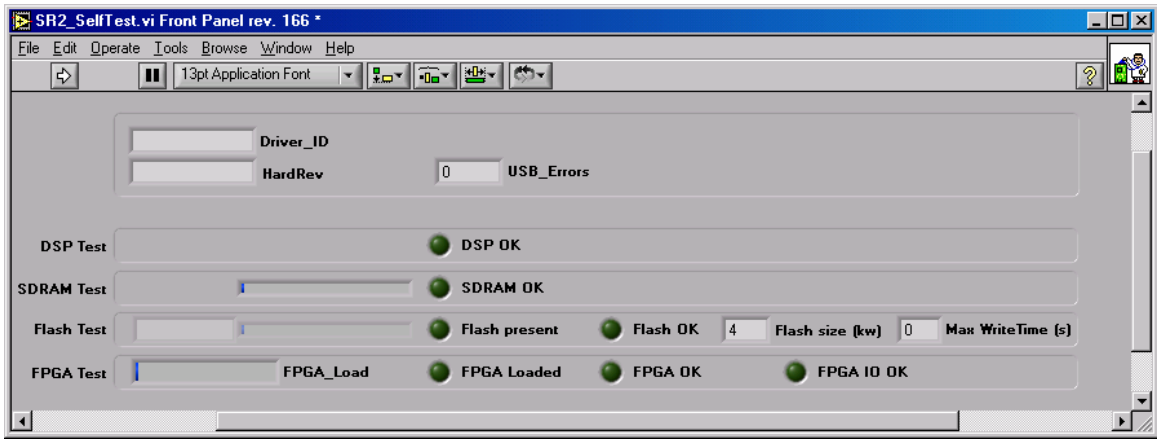


Figure 2 SR2_Self_Test application

To run the application again after it has stopped, simply click on the arrow at the top left of the window.

The application initializes the board, then loads the kernel on the DSP, and proceeds to test the DSP, external RAM, Flash and FPGA. Each test takes a few seconds to complete.

Note: The Flash test erases the contents of the Flash. A dialog is presented before this operation so that the user may cancel it to preserve the Flash contents.

Note: The FPGA IO test configures all the IOs as outputs to test them. This may damage an IO, or external logic, if any logic is connected to the FPGA (though the expansion connector J4) during the test. A dialog is presented before this operation so that the user may cancel it to avoid damaging connected IOs.

Indicators:

- **Driver_ID**: A character string of the form *SRm2_x*, where 0<*x*<n-1 refers to the actual board being accessed.

Indexes x are given by the PC to *Signal_Ranger_mk2* boards, at connection time, in the order they are connected to the USB chain. For instance:

- SRm2_0 will be given to the first board connected
- SRm2_1 will be given to the second one...etc.

- **HardRev**: The firmware revision number of the on-board USB controller.
- **USB_Errors**: Indicates the number of USB errors detected during the test. Note that the presence of some USB errors does not preclude the operation of the *Signal_Ranger_mk2* board. The USB protocol is very robust to errors. However, this is usually an indication of a poor USB cable or USB connection. When the error rate is too large it may cause a USB communication break-down.
- **DSP OK**: Lights up green if the DSP test is successful. Lights-up red if it is not.
- **SDRAM OK**: Lights up green if the SDRAM test is successful. Lights-up red if it is not.
- **Flash Present**: Lights-up green if the Flash is detected. Lights-up red if it is not.
- **Flash OK**: Lights up green if the Flash test is successful. Lights-up red if it is not. Does not light-up if the test is skipped.
- **Flash Size**: Should indicate 1024 kwords if the Flash is properly detected.
- **Max Write Time**: Indicates the time it takes to program one 16-bit word into the Flash. It should be around 60μs if the Flash is in good shape.
- **FPGA Loaded**: Lights up green if the FPGA is loaded successfully. Lights-up red if it is not.
- **FPGA OK**: Lights up green if the FPGA is able to communicate with the DSP. Lights-up red if it is not. Does not light-up if the test is skipped.
- **FPGA IO OK**: Lights up green if the FPGA IO test is successful. Lights-up red if it is not. Does not light-up if the test is skipped.

Hardware Description

Connector Map

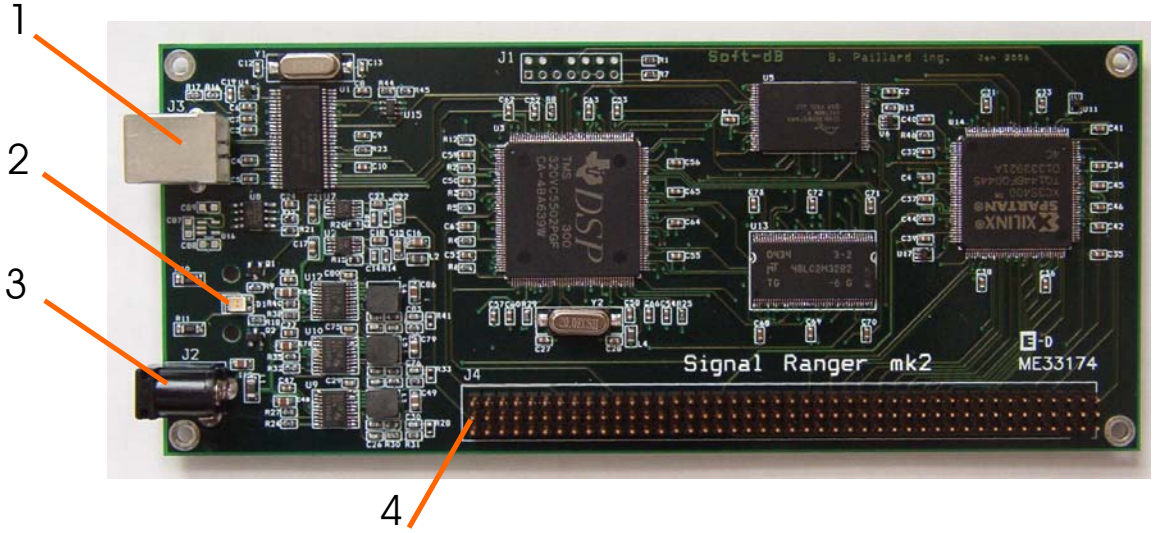
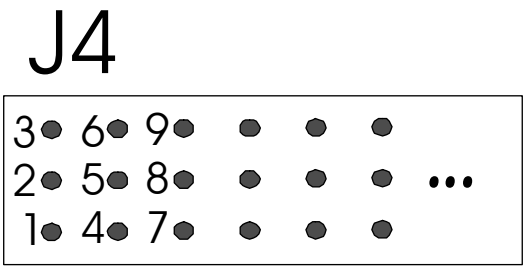


Figure 3

Legend:

1	USB port
2	Bi-color LED
3	5V Power Supply
4	Expansion connector J4

Expansion Connector J4



No	Function	No	Function	No	Function
1	+5V	2	Gnd	3	+2.5V
4	+1.8V	5	Gnd	6	+1.2V
7	+1.26V	8	Gnd	9	+3.3V
10	-3.3V	11	Gnd	12	DSP_Reset
13	CLKR0	14	Gnd	15	DR0
16	FSR0	17	Gnd	18	CLKX0

19	DX0	20	Gnd	21	FSX0
22	CLKR1	23	Gnd	24	DR1
25	FSR1	26	Gnd	27	CLKX1
28	DX1	29	Gnd	30	FSX1
31	UART_Rx	32	Gnd	33	UART_Tx
34	NC	35	Gnd	36	NC
37	FPGA_17	38	Gnd	39	FPGA_15
40	FPGA_14	41	Gnd	42	FPGA_13
43	FPGA_12	44	Gnd	45	FPGA_11
46	FPGA_10	47	Gnd	48	FPGA_8
49	FPGA_7	50	Gnd	51	FPGA_6
52	FPGA_5	53	Gnd	54	FPGA_4
55	FPGA_2	56	Gnd	57	FPGA_1
58	FPGA_141	59	Gnd	60	FPGA_140
61	FPGA_137	62	Gnd	63	FPGA_135
64	FPGA_68	65	Gnd	66	FPGA_69
67	FPGA_70	68	Gnd	69	FPGA_73
70	FPGA_74	71	Gnd	72	FPGA_76
73	FPGA_77	74	Gnd	75	FPGA_78
76	FPGA_79	77	Gnd	78	FPGA_80
79	FPGA_82	80	Gnd	81	FPGA_83
82	FPGA_84	83	Gnd	84	FPGA_85
85	FPGA_86	86	Gnd	87	FPGA_87
88	FPGA_89	89	Gnd	90	FPGA_90
91	FPGA_92	92	Gnd	93	FPGA_93
94	FPGA_95	95	Gnd	96	FPGA_96
97	FPGA_97	98	Gnd	99	FPGA_98
100	FPGA_99	101	Gnd	102	FPGA_100
103	FPGA_102	104	Gnd	105	FPGA_103
106	FPGA_104	107	Gnd	108	FPGA_105
109	FPGA_107	110	Gnd	111	FPGA_108
112	FPGA_112	113	Gnd	114	FPGA_113
115	FPGA_116	116	Gnd	117	FPGA_118
118	FPGA_119	119	Gnd	120	FPGA_122
121	FPGA_123	122	Gnd	123	FPGA_124
124	FPGA_125	125	Gnd	126	FPGA_129
127	FPGA_130	128	Gnd	129	FPGA_131
130	FPGA_132	131	Gnd	132	FPGA_HS_EN

Power Supply Pins

+5V

This is the same +5V line that is brought to the power connector J2. The maximum current that may be drawn from this pin is 500mA. It may be further limited by the capacity of the power-supply that is used.

+2.5V

This is the FPGA's Vccaux supply. A maximum of 200mA may be drawn from this pin.

Note: This value takes into account the FPGA's Vccaux quiescent current. This is valid when the FPGA is not loaded or is loaded with the factory-default logic.

+1.8V

This supply is not used on-board. It is intended for external devices, such as ADC's/DACs. A maximum of 250mA may be drawn from this pin.

+1.2V

This is the FPGA's Vccint supply. A maximum of 400mA may be drawn from this supply.

Note: This value takes into account the FPGA's Vccint quiescent current. This is valid when the FPGA is not loaded or is loaded with the factory-default logic.

+1.26V

This is the DSP's CVdd supply. A maximum of 250 mA may be drawn from this supply.

+3.3V

This supply is used by the DSP, FPGA, Flash, SDRAM and USB controller. A maximum of 100mA may be drawn from this supply.

Note: This value takes into account the FPGA's Vcco quiescent current. This is valid when the FPGA is not loaded or is loaded with the factory-default logic. It does not take into account current drawn from the FPGA's I/O pins.

-3.3V

This supply is not used on-board. It is intended for analog devices, such as ADC's/DACs. A maximum of 60mA may be drawn from this pin.

DSP Pins

DSP_Reset

This is the DSP's reset pin. It is activated (low) at power-up, and under control of the USB controller. It may be used to reset external logic whenever the DSP is reset.

McBSP_0, McBSP_1

All the DSP's McBSP_0 and McBSP_1 signals are provided on J4. These may be used to interface external devices, such as AICs, to the DSP.

UART

The DSP's UART Tx and Rx pins are provided on J4.

FPGA Pins

FPGA_*i*

All of the FPGA's free I/Os and clock pins are provided on J4.

FPGA_*HS_EN*

This is the FPGA's HSWAP_EN pin. If pulled-up or left floating, all the FPGA I/Os are floating during configuration. If it is pulled low, the FPGA I/Os are pulled-up during configuration. The state of the FPGA I/Os after configuration is defined by the logic loaded into the FPGA, and the state of HSWAP_EN has no bearing on it.

System Frequencies

The DSP crystal has a frequency of 20MHz. Immediately after reset, the various system frequencies are as follows:

- CPU Core Clock: 20MHz
- SYSCLK1 (Fast Peripherals): 5MHz
- SYSCLK2 (Slow Peripherals): 5MHz
- SYSCLK3 (EMIF): 5MHz

However, the above configuration is short-lived. Just after reset, the kernel is loaded into DSP memory and run. The kernel configures the clock generator as follows:

- CPU Core Clock: 300 MHz
- SYSCLK1 (Fast Peripherals): 150 MHz
- SYSCLK2 (Slow Peripherals): 75 MHz
- SYSCLK3 (EMIF): 75 MHz

Peripheral Interfaces

Memory Map

The following figure describes the DSP memory map. Addresses are in bytes.

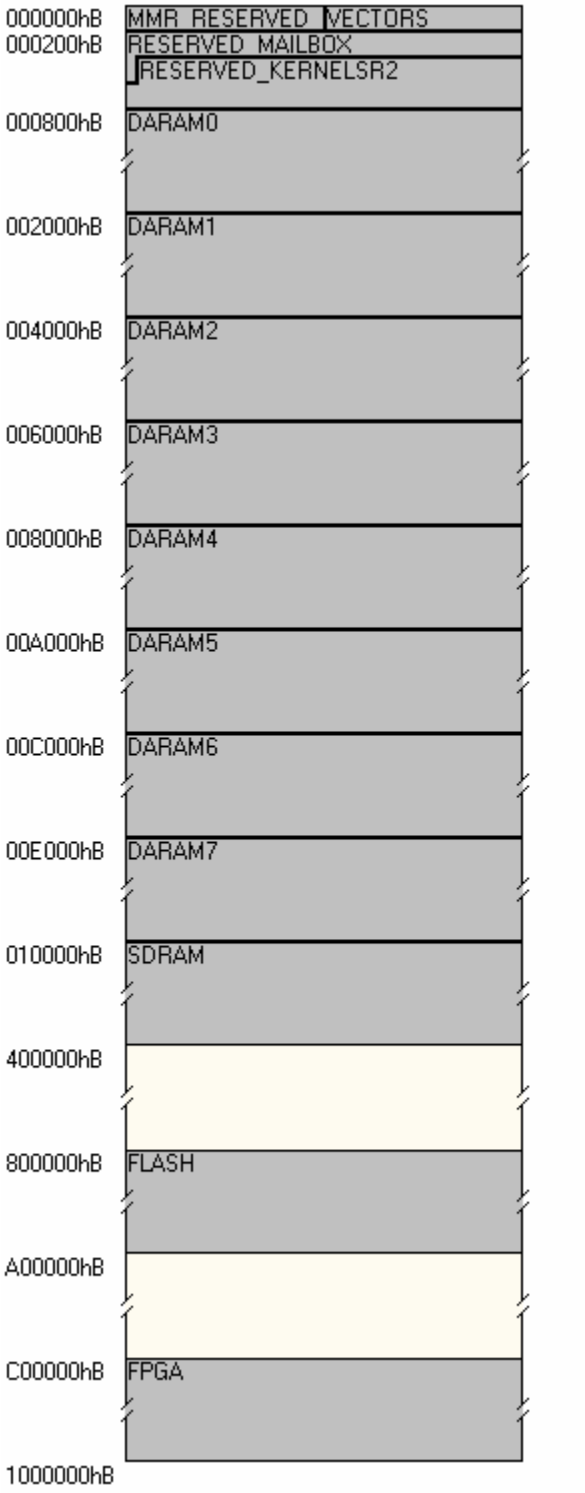


Figure 4 Memory Map

EMIF Configuration

The EMIF is properly configured by the kernel to access the Flash, SDRAM and FPGA.

With the DSP running at 300MHz, the EMIF is configured by the DSP kernel to run at 75MHz (1/4 CPU clock).
The EMIF registers are configured as follows:

- EMIF Global Control Register 1 (0x0800) Value: 0x07FC
  - NOHOLD = 1
  - EK1HZ = 1
  - EK1EN = 1

- EMIF Global Control Register 2 (0x0801) Value: 0x000A
  - EK2RATE = 2
  - EK2HZ = 1
  - EK2EN = 0 (disabled)

- EMIF CE0 SPACE Control Register 1 (0x0804) Value: 0xFF33

(all fields are defaults, except MTYPE)

  - TA = 3
  - READ STROBE = 63
  - MTYPE = 3
  - WRITE HOLD MSB = 0
  - READ HOLD = 3

- EMIF CE0 SPACE Control Register 2 (0x0805) Value: 0xFFFF

(all fields are default)

  - WRITE SETUP = 15
  - WRITE STROBE = 63
  - WRITE HOLD = 3
  - READ SETUP = 15

- EMIF CE2 SPACE Control Register 1 (0x0808) Value: 0xC811
  - TA = 3
  - READ STROBE = 8
  - MTYPE = 1
  - WRITE HOLD MSB = 0
  - READ HOLD = 1

- EMIF CE2 SPACE Control Register 2 (0x0809) Value: 0x11E2
  - WRITE SETUP = 1
  - WRITE STROBE = 7
  - WRITE HOLD = 2
  - READ SETUP = 2

- EMIF CE3 SPACE Control Register 1 (0x080A) Value: 0xC422
  - TA = 3
  - READ STROBE = 4
  - MTYPE = 1
  - WRITE HOLD MSB = 0
  - READ HOLD = 2

- EMIF CE3 SPACE Control Register 2 (0x080B) Value: 0x2122
  - WRITE SETUP = 2
  - WRITE STROBE = 4
  - WRITE HOLD = 2

- READ SETUP = 2

- EMIF SDRAM Control Register 1 (0x080C) Value: 0x5000
  - TRC = 5 (6 cycles @75MHz for 70ns)
  - SLFRFR = 0 (SDRAM in normal operation)

- EMIF SDRAM Control Register 2 (0x080D) Value: 0x4611 (0x4711 for init)
  - SDWDTH[4:0] = 3 (4 banks, 11 row-bits, 8 col-bits)
  - RFEN = 1 (Refresh Enable)
  - INIT = 0 (No Init)
  - TRCD = 1 (2 cycles @75MHz for 20ns)
  - TRP = 1 (2 cycles @75MHz for 20ns)

- EMIF SDRAM Refresh Control Register (0x080E) Value: 0x0494
  - Period = 1172 (15.625µs/row @ 75MHz)

- EMIF SDRAM Refresh Control Register 2 (0x080F) Value: 0x0000
(do not initialize, leave default value)
  - Extra Refreshes = 0 (1 refresh)

- EMIF SDRAM Extension Register (0x0810) Value: 0x4527
  - R2WDQM(L) = 0 (3 cycles recommended value (CL = 3))
  - RD2WR = 4 (5 cycles recommended value (CL = 3))
  - RD2DEAC = 1 (2 cycles recommended value (CL = 3))
  - RD2RD = 0 (1 cycle recommended value (CL = 3))
  - THZP = 2 (3 cycles @75MHz)
  - TWR = 1 (2 cycles @75MHz for 20.33ns)
  - TRRD = 0 (2 cycles @75MHz for 14ns)
  - TRAS = 3 (4 cycles @75MHz for 42ns)
  - TCL = 1 (3 cycles)

- EMIF SDRAM Extension Register 2 (0x0811) Value: 0x0005
  - WR2RD = 0 (1 cycle recommended value (CL = 3))
  - WR2DEAC = 1 (2 cycles recommended value (CL = 3))
  - WR2WR = 0 (1 cycle recommended value (CL = 3))
  - R2WDQM(H) = 1 (3 cycles recommended value (CL = 3))

SDRAM

This is a 2Mx32 (8 Mbytes) device. However, the EMIF interface only allows access to the first half (1Mx32) of the device. Together with the on-chip RAM, the SDRAM device covers the whole of the CE0 space.

Memory map

The SDRAM is interfaced on CE0, at byte-addresses 10000_H to $3FFFFF_H$ (word addresses 8000_H to $1FFFFF_H$). It follows the on-chip DSP DARAM.

Speed

The SDRAM device is clocked by ECLKOUT1. It can be clocked at a frequency up to 100MHz. However, when the DSP is running at 300MHz, then ECLKOUT1 must be set to $1/4^{th}$ the CPU clock frequency (75MHz). This is the configuration that is provided as a default.

If the CPU is set to run at 200MHz, then ECLKOUT1 may be set to ½ the CPU clock frequency. In this condition the SDRAM would be clocked at 100MHz. This alternate configuration would give a slightly faster SDRAM access time, at the cost of a slower DSP. This alternate configuration is possible. However it is not the configuration that is provided as a default.

Flash

This is a 4 Mbytes device (2Mx16). However, the EMIF interface only allows access to half (1Mx16) of the device.

Memory map

The device is interfaced on CE2, at byte addresses 800000_H to $9FFFFF_H$ (word addresses 400000_H to $4FFFFF_H$).

Sectors

The FLASH is segmented into 32 sectors of 32kWords each.

Interrupt

The (inverted) RY/BY output of the device is connected to the DSP's INT1 input. This way, programming and erasure operations can be managed using INT1 interrupts. The provided Flash driver uses INT1 to perform writes.

Access efficiency

Because this device is configured in the EMIF as a 16-bit wide memory, the EMIF always reads 2 words at a time regardless of whether one word or two words are specified by the DSP instruction. When a CPU instruction specifies a single 16-bit word read, the EMIF reads two consecutive words, and discards the one that is not specified in the instruction. This process costs a read cycle every time a 16-bit word is read.
This does not happen when the EMIF writes to the Flash memory.

Incremental programming

Contrary to previous generations of Flash devices that have been used in *Signal Ranger* boards, the Flash device used in *Signal_Ranger_mk2* cannot be incrementally programmed. This means that a word location that has been previously programmed MUST be erased before reprogramming. This is true even if the reprogramming operation is only intended to turn some of the remaining "1s" into "0s".

FPGA

Memory map

The device is interfaced on CE3, at byte addresses $C00000_H$ to $FFFFFF_H$ (word addresses 600000_H to $7FFFFF_H$).

Physical interface

The details of the interface between the FPGA and the DSP are as follows:

- **Data lines:** The 16 lower bits of the EMIF's data bus are connected to the FPGA. Only the 8 lower bits are used during configuration.
  - **Data lines (low):** The EMIF data bus lines D7 to D0 are connected to the FPGA pins 46, 47, 50, 51, 59, 60, 63 and 65 respectively. These are used for FPGA configuration and may be used after configuration for FPGA read/write.

    - **Data lines (high):** The EMIF data bus lines D15 to D8 are connected to the FPGA pins 28, 30, 31, 32, 33, 35, 36, 44 respectively. These should be used for FPGA read/write after configuration.
- **Address lines:** The EMIF lines A6 to A2 are connected to the FPGA pins 23, 24, 25, 26, and 27 respectively. These may be used after configuration to distinguish between a maximum of 32 logical addresses during FPGA read/write.
- **RDWR_B** The FPGA's RDWR_B input is tied to ground. Therefore it is not possible to read-back configuration data. This pin should not be used after configuration because it is tied to ground.
- **PROG_B** The FPGA's PROG_B input is connected to the DSP's XF output and is used to initiate FPGA configuration.
- **DONE** The FPGA's DONE pin is pulled-up to 3.3V using a 10k resistor, and connected to the DSP's GPIO3 pin. GPIO3 should be configured as an input and is required to monitor the FPGA's configuration.
- **INIT_B** The FPGA's INIT_B pin is pulled-up to 3.3V using a 10k resistor, and is connected to the DSP's INT3. It is required to monitor the FPGA's configuration process and may be used to trigger the INT3 interrupt after configuration. The INIT_B pin is also connected to the DSP's GPIO5 pin. GPIO5 should be configured as an input and may be used as an alternate to INT3 to monitor the FPGA's configuration process
- **AWE** The EMIF line AWE is connected to the FPGA's CCLK through a 3.3V-tolerant buffer. This signal is used to configure the FPGA. There is a maximum 4.4ns buffer delay between the EMIF AWE line and the FPGA CCLK input. AWE is also connected to the FPGA's pin 56. This is a GCLK input. It may be used after configuration for an FPGA write cycle.
- **ARE** The EMIF line ARE is connected to the FPGA pin 55. This is a GCLK input. It may be used after configuration for an FPGA read cycle.
- **AOE** The EMIF line AOE is connected to the FPGA pin 53. This is a GCLK input. It may be used after configuration for an FPGA read cycle.
- **CE3** The EMIF line CE3 is connected to the FPGA CS_B. This pin is used to select the FPGA during configuration. It may be used after configuration for an FPGA read or write cycle.
- **INT2** The DSP INT2 input is connected to the FPGA pin 20. It may be used after configuration to trigger the INT2 interrupt. To use this as an external interrupt input, a simple follower may be implemented inside the FPGA to connect a line from any general-purpose IO on expansion connector J4 to the FPGA pin 20.
- **INT0** The DSP INT0 input is connected to the FPGA pin 18. It may be used after configuration to trigger the INT0 interrupt. To use this as an external interrupt input, a simple follower may be implemented inside the FPGA to connect a line from any general-purpose IO on expansion connector J4 to the FPGA pin 18.
- **NMI** The DSP NMI input is connected to the FPGA pin 21. It may be used after configuration to trigger the NMI interrupt. To use this as an external interrupt input, a simple follower may be implemented inside the FPGA to connect a line from any general-purpose IO on expansion connector J4 to the FPGA pin 21.
- **DSP CLKOUT** The CLKOUT DSP clock output is connected to the FPGA pin 52. This is a GCLK input and may be used to clock the FPGA design. By default it runs at 150MHz after the kernel has been loaded.
- **Clock TIM0** The TIM0 DSP clock IO is connected to the FPGA pin 128. This is a GCLK input and may be used to clock the FPGA design.
- **Clock TIM1** The TIM1 DSP clock IO is connected to the FPGA pin 127. This is a GCLK input and may be used to clock the FPGA design.

EMIF Configuration

The EMIF is configured on CE3 (the FPGA address range) as a 32-bit device. The reason for this is that if any other interface width is used (16 or 8 bits), the EMIF automatically makes several

accesses during a read cycle to the device (2 accesses for a 16-bit interface and 4 accesses for an 8-bit interface – read accesses are always performed for a total of 32 bits). Although the read behaviour of the EMIF is generally only an annoyance when reading memory with a width other than 32 bits (read cycles are wasted and unneeded data is discarded). The extra read cycles might cause problems when reading specialized logic. For instance if the logic implemented on the FPGA is a FIFO, the extra read cycles would advance the FIFO. More information on the EMIF may be found in Texas Instrument's document SPRU621.
Even though it is interfaced to the EMIF as a 32-bit interface, only the 16 lower bits of the EMIF's data bus are used to transport data to and from the FPGA. This provides more I/Os to be used in the FPGA's user logic.
Because of this particular arrangement, care must be exercised when transferring data to and from the FPGA, and in particular when configuring the FPGA. More details are provided below.

FPGA Configuration

During configuration the EMIF address bits are ignored by the FPGA. A write to any address within the CE3 space should be valid to send configuration data into the FPGA. However, because the EMIF is a 32-bit interface, the address used for the access has an impact on whether the high 16 bits or the low 16 bits of the data bus is used for the transfer. To make sure that the data is transported on the low part of the data bus, which is the one interfaced to the FPGA, make sure that an odd word-address (a byte-address that has its bit No 1 set to 1) is used to access the FPGA, and make sure that a 16-bit access is used (not a 32-bit access). For instance a series of word writes at byte-address $C00002_H$ may be used to configure the FPGA.

To initiate the configuration the DSP must send a low pulse at least 300ns-long on XF. It must then wait for INIT_B (GPIO5) to go high or wait for at least 5ms after the rising edge on XF.

When configuring the FPGA only the lower 8 bits of data are used.

Suggested Setup After Configuration

After configuration it is suggested that the EMIF interface to the FPGA be kept configured as a 32-bit asynchronous interface, only using data bits 0 to 15 (the EMIF's data bits 16 to 31 are not physically connected to the FPGA).
Only the 5 lower address bits of the EMIF are connected to general purpose IOs of the FPGA. These addresses may be used to select one among a maximum of 32 registers that may be defined in the FPGA logic. The following table provides the address that must be used from the DSP's point of view, as a function of the 5 address lines. Just as for the configuration, the table assumes that a word access is performed by the DSP.

FPGA Register Address	DSP Byte-Address	DSP Word-Address
00_H	$C00002_H$	600001_H
01_H	$C00006_H$	600003_H
02_H	$C0000A_H$	600005_H
03_H	$C0000E_H$	600007_H
04_H	$C00012_H$	600009_H
05_H	$C00016_H$	$60000B_H$
06_H	$C0001A_H$	$60000D_H$
07_H	$C0001E_H$	$60000F_H$
08_H	$C00022_H$	600011_H
09_H	$C00026_H$	600013_H
$0A_H$	$C0002A_H$	600015_H
$0B_H$	$C0002E_H$	600017_H
$0C_H$	$C00032_H$	600019_H
$0D_H$	$C00036_H$	$60001B_H$

$0E_H$	$C0003A_H$	$60001D_H$
$0F_H$	$C0003E_H$	$60001F_H$
10_H	$C00042_H$	600021_H
11_H	$C00046_H$	600023_H
12_H	$C0004A_H$	600025_H
13_H	$C0004E_H$	600027_H
14_H	$C00052_H$	600029_H
15_H	$C00056_H$	$60002B_H$
16_H	$C0005A_H$	$60002D_H$
17_H	$C0005E_H$	$60002F_H$
18_H	$C00062_H$	600031_H
19_H	$C00066_H$	600033_H
$1A_H$	$C0006A_H$	600035_H
$1B_H$	$C0006E_H$	600037_H
$1C_H$	$C00072_H$	600039_H
$1D_H$	$C00076_H$	$60003B_H$
$1E_H$	$C0007A_H$	$60003D_H$
$1F_H$	$C0007E_H$	$60003F_H$

FPGA pinout constraints

Since some of the FPGA pins that can become user I/Os after configuration are physically tied to DSP outputs, Vcc or Gnd, it is of critical importance that the FPGA designer properly check the FPGA pinout after configuration. An improper FPGA pinout, leading for instance to an FPGA output driving into a ground or an EMIF output, may lead to hardware damage. In particular, the following FPGA pins should be checked:

- FPGA pins 28, 30, 31, 32, 33, 35, 36, 44, 46, 47, 50, 51, 59, 60, 63 and 65 are connected to the EMIF's data bus (D0 to D15). After configuration these FPGA pins should only be used to exchange data with the DSP (in accordance to proper EMIF's read/write cycle), configured as inputs or left floating.
- FPGA pins 23, 24, 25, 26, and 27 are connected to the EMIF's address bus (a0 to A4). After configuration these pins should only be configured as inputs or left floating.
- FPGA pin 41 (RDWR_B) is tied to ground. After configuration this pin should only be configured as an input or left floating.
- FPGA pins 40, 53, 55, and 56 are connected to EMIF control outputs. After configuration these pins should only be configured as inputs or left floating.
- FPGA pins 52, 127 and 128 are connected to DSP clock outputs. After configuration these pins should only be configured as inputs or left floating.

Peripheral Access Benchmarks

Note: The benchmarks below assume that the EMIF is ready to execute the read or write cycle. In particular no SDRAM refresh cycle occurs during the read or write access.

SDRAM

16-bit Write

147 ns / 11 EMIF cycles

16-bit Read

240 ns / 18 EMIF cycles

32-bit Write

147 ns / 11 EMIF cycles

32-bit Read

240 ns / 18 EMIF cycles

Flash

16-bit Write

147 ns / 11 EMIF cycles

- 1 cycle Write setup
- 7 cycles Write strobe
- 2 cycles Write hold
- 1 cycle Write hold period (added so that total cycle number is >= 11)

Note: This is not the programming time, but rather the time of the write cycle, which is part of the programming operation.

16-bit Read

440 ns / 33 EMIF cycles

- 2 cycles Read setup
- 8 cycles Read strobe
- 1 cycle Read hold
- 2 cycles Read setup
- 8 cycles Read strobe
- 1 cycle Read hold
- 11 cycles Read hold period (added to last Read hold so that the sum is equal to 12 cycles)

Note: Because the Flash is configured as a 16-bit peripheral, a read operation in Flash actually triggers two consecutive read cycles from the EMIF. The first one is at the required address. The second one is at the following address. The data from the second read is discarded by the EMIF. This does not occur with writes.

FPGA

16-bit Write

147 ns / 11 EMIF cycles

- 2 cycles Write setup
- 4 cycles Write strobe
- 2 cycles Write hold
- 3 cycles Write hold period (added so that total cycle number is >= 11)

16-bit Read

240 ns / 18 EMIF cycles

- 2 cycles Read setup
- 4 cycles Read strobe
- 2 cycles Read hold
- 10 cycles Read hold period (added to last Read hold so that the sum is equal to 12 cycles)

Note: Because the FPGA is configured as a 32-bit peripheral, the EMIF only performs one read cycle (rather than two consecutive read cycles) for each read operation at a specified address.

Factory-Default FPGA Logic

The Flash is loaded at the factory with a default FPGA Logic file. This file is loaded into the FPGA at power-up by the Power-Up kernel. The file is called *SR2_SelfTest.rbt*, and may be found in the *C:\Program Files\SignalRanger_mk2* directory. In case the Flash is re-written with a different FPGA configuration, the factory-default FPGA logic may be programmed into the Flash again using the mini-debugger.

This factory-default logic implements three 16-bit General Purpose I/O Registers, and one 15-bit General Purpose I/O Register.

The Logic only uses 3% of the FPGA's logic resources, and does not consume significant power beyond the FPGA's quiescent current values (see Xilinx documentation).

The I/Os for these registers are mapped to the following pins of the J4 Expansion Connector:

No	Function	No	Function	No	Function
37	PortC_0	38	Gnd	39	PortD_1
40	PortC_1	41	Gnd	42	PortD_2
43	PortC_2	44	Gnd	45	PortD_3
46	PortC_3	47	Gnd	48	PortD_4
49	PortC_4	50	Gnd	51	PortD_5
52	PortC_5	53	Gnd	54	PortD_6
55	PortC_6	56	Gnd	57	PortD_7
58	PortC_7	59	Gnd	60	PortD_8
61	PortC_8	62	Gnd	63	PortD_9
64	PortC_9	65	Gnd	66	PortD_10
67	PortC_10	68	Gnd	69	PortD_11
70	PortC_11	71	Gnd	72	PortD_12
73	PortC_12	74	Gnd	75	PortD_13
76	PortC_13	77	Gnd	78	PortD_14
79	PortC_14	80	Gnd	81	PortD_15
82	PortC_15	83	Gnd	84	PortB_0
85	PortA_0	86	Gnd	87	PortB_1
88	PortA_1	89	Gnd	90	PortB_2
91	PortA_2	92	Gnd	93	PortB_3
94	PortA_3	95	Gnd	96	PortB_4
97	PortA_4	98	Gnd	99	PortB_5
100	PortA_5	101	Gnd	102	PortB_6
103	PortA_6	104	Gnd	105	PortB_7
106	PortA_7	107	Gnd	108	PortB_8
109	PortA_8	110	Gnd	111	PortB_9
112	PortA_9	113	Gnd	114	PortB_10
115	PortA_10	116	Gnd	117	PortB_11
118	PortA_11	119	Gnd	120	PortB_12
121	PortA_12	122	Gnd	123	PortB_13
124	PortA_13	125	Gnd	126	PortB_14
127	PortA_14	128	Gnd	129	PortB_15
130	PortA_15	131	Gnd	132	FPGA_HS_EN

Each of the four ports (PortA, PortB, PortC, PortD) has two registers:

- **Port*X*_Dir** is a direction register. It sets the corresponding pin as an input or an output. By writing a bit pattern to the *PortX_Dir* register, any individual port pin may be configured as an input (0) or an output (1).
- **Port*X*_Dat** is a data register. It may be read to determine the state of an input pin, or written to set the state of an output pin.

Hardware Details

All pins are configured as inputs at power-up.

All inputs and outputs are programmed to conform to the LVCMOS 3.3V standard.

When a port pin is configured as an input, a weak keeper is instantiated for the input. This way, the pin will keep its last set state if it is left floating.

Pins configured as outputs have a +- 12mA current capacity.

If FPGA_HS_EN is pulled-low during board power-up, then pull-ups will appear on all FPGA pins during configuration. If FPGA_HS_EN is pulled-up or left floating during board power-up, then the pull-ups will not be present during configuration. In any case, the pull-ups disappear after the FPGA is configured, and only the weak keepers are left on the pins.

Note: The FPGA pins configured as inputs should not be driven by a voltage greater than 3.8V when the FPGA is powered, or greater than 0.5V when the FPGA is not powered. Alternately, the current through any input should be no more than 10mA. Refer to relevant Xilinx documentation for details.
For applications where FPGA IOs may be driven by voltages higher than 3.3V, or may be driven when the Signal Ranger mk2 board is not powered, we recommend the use of a bus switch such as the SN74CB3T16211.

Register Map

Register	Byte-Address	Word-Address
Port_A_Dat	$C00002_H$	600001_H
Port_A_Dir	$C00006_H$	600003_H
Port_B_Dat	$C0000A_H$	600005_H
Port_B_Dir	$C0000E_H$	600007_H
Port_C_Dat	$C00012_H$	600009_H
Port_C_Dir	$C00016_H$	$60000B_H$
Port_D_Dat	$C0001A_H$	$60000D_H$
Port_D_Dir	$C0001E_H$	$60000F_H$

Software Interfaces

In the following sections the LabVIEW interface is used as an example. The same concepts that are discussed are equally applicable to the C/C++ interface. Simply replace the term “VI” in the text by “function”. Every specific VI discussed below has a corresponding function in the C/C++ interface.

How DSP Boards Are Managed

Every time a DSP board is connected to the PC, the PC loads its driver, and creates a data structure representing the board in the system. This data structure is accessed by a symbolic name of the form "base*name_i*", where *basename* is a symbolic name that represents the type of board (for instance SignalRanger_mk2 boards have the name *SRm2_*), and *i* is a zero-based number that is assigned to the board at connection time. For instance if 3 SignalRanger_mk2 boards are connected in sequence on the PC, their complete symbolic names will be *SRm2_0*, *SRm2_1*and *SRm2_2*.

For each board that is opened (using the *SR2_Base_Open_Next_Avail_Board* VI in the LabVIEW interface), the interface software creates and keeps an entry in a *Global Board Information Structure* that contains information about open boards. This structure contains the following information:

- A handle on the board driver that is required to access the board
- A board ID structure that contains the following:
  - USB Vendor ID Number
  - USB Product ID Number
  - Hardware Revision Number
- A Symbol Table for the symbols of the kernel that is presently loaded on the DSP. This table is created/updated, whenever a kernel is loaded or reloaded.
- A Symbol Table for the symbols of the user code that is presently loaded on the DSP. This table is created/updated whenever new DSP code is loaded on the DSP.
- A High-Level USB retries structure. Whenever a USB transaction fails (because of noise on the USB line or other abnormal conditions), the interface retries the transaction 5 times, before returning an error. The structure is updated every time a transaction is retried. It contains the complete call-chain of Vis that generated the faulty transaction. Note that this error control structure is implemented at high-level, "on top of" the intrinsic USB error control. The USB protocol itself will retry up to 3 times in case of USB errors, before failing the transaction. High-Level retries occur only after a USB transaction is failed at low-level.

All the Vis of the interface must be passed a **BoardRef** number. This number points to the entry corresponding to the selected board in the *Global Board Information Structure*. It is created by the *SR2_Base_Open_Next_Avail_Board* VI.

When a board is closed, its entry in the *Global Board Information Structure* is deleted.

Note: The handle that the driver provides to access the board is exclusive. This means that only one application, or process, at a time can open and manage a board. A consequence of this is that a board cannot be opened twice. A board that has already been opened using the SR2_Base_Open_Next_Avail_Board VI cannot be opened again until it is properly closed using the SR2_Base_Close_BoardNb VI. This is especially a concern when the application managing the board is closed under abnormal conditions. If the application is closed without properly closing the board, the next execution of the application will fail to find and open the board, simply because the corresponding driver instance is still open.
One procedure to get out of this lock-up is to completely close the application, disconnect the board, then reconnect the board and re-open and run the application. Closing the application is required in addition to disconnecting the board so that Windows may unload the driver.

Kernel vs Non-Kernel Interface Vis

Some of the memory transfer functions are supported by two seemingly equivalent groups of Vis : Those that use a DSP kernel, and those that do not. There are major functional differences between the two:

- The most basic difference is that Vis that execute transfers using the kernel require the kernel to be loaded in DSP memory and running properly to perform the transfer. Therefore they may fail if the DSP has crashed. On the other hand, Vis that do not use the kernel only rely on the hardware of the HPI to execute the transfer. They will not fail even if the DSP has crashed. This is an important consideration when debugging code.
- Vis that use the kernel have access to any address range in any memory space. While Vis that do not use the kernel are limited in scope. They can only transfer to and from the memory that is directly accessible by the hardware of the HPI (essentially the DSP's on-chip DARAM).
- Vis that use the kernel use bulk pipes to achieve the transfer. Vis that do not use the control pipe 0.
    - One consequence of using bulk pipes for transfers using the kernel is that the transfer bandwidth is much faster. Transfers using control pipe zero have only a limited bandwidth (about 500kb/s for USB 1.1).
    - Another consequence of using bulk pipes for transfers using the kernel is that the bandwidth is shared with other USB devices in the same chain and is not guaranteed. There is a possibility in principle that another device in the USB chain could take so much bandwidth that it would slow down DSP board transfers considerably. On the other hand, transfers that do not use the kernel benefit from the guaranteed bandwidth of control pipe zero (as low as it may be).

In short, transfers that use the kernel are higher-performance, and have a much wider scope. On the other hand, transfers that do not use the kernel are much more reliable, especially during debugging, or in circumstances where the DSP code may crash.

Error Control

There are four levels of error control, and four basic error types:

Low-Level USB Errors

The USB protocol itself provides a large amount of error control. The Host Controller Driver on the PC retries packets that contain errors, because of noise, faulty cable or other abnormal conditions up to three times per transaction before failing them. This level of error control is generally hidden from the user, and the developer. The USB controller on the DSP board however has an error counter that can be accessed by PC code to monitor the reliability of the USB connection. This circular 4-bit error counter is incremented every time a USB error is detected. Interface Vis are provided to read and reset this counter. This counter indicates the errors occurring at the level of the USB protocol (low-level).

USB Retries

The software interface described below has its own layer of USB error control that operates at a higher level. Whenever the Host Controller Driver on the PC decides to fail a transaction (after 3 retries), the interface software retries it up to 5 times. Every time it does, it logs an entry in the *Retry* member of the *Global Board Information Structure* (see above), indicating the number of retries, and the call-chain that led to the error. If after 5 times, the transaction is still not successful, the VI that led to the error abandons and returns an error. The *Retry* member of the *Global Board Information Structure* may be read at any time in order to determine if errors are occurring on a regular basis.

High-Level Errors

When a USB transaction is failed at the level of the interface (after 5 high-level retries (i.e. 15 low-level retries), then the transaction is failed at high-level, and the VI that created the conditions returns an appropriate error code in its *Error-Out* structure. This error code may be processed by the *SR2_Base_Error_Message* VI to obtain a text description of the error, as well as the call chain that led to the error. Note that errors other than USB failed transactions are also reported in

the *Error-Out* structure. For instance if critical files are missing (such as a kernel), this will also be reported at this level.

Application-Specific DSP Errors And Completion Codes

The DSP kernel manages a field in its mailbox that may be used to return a user-specific error or completion code whenever a user DSP function is executed. This error or completion code is completely under the control of the developer. It may be used or not. This code is returned as an *ErrorCode* indicator by all the Vis that access the DSP via the kernel.

Note: For this ErrorCode to be returned by the DSP, there must be no USB communication error.

USB Lock-Up

Contrary to the situation for the previous Signal Ranger boards, there are situations in *Signal_Ranger_mk2* that can lead to a lock-up of the USB channel:

- The TMS320VC5502 has extensive clock and peripheral functionality under software control. So much so that it is possible to stop the clock on the DSP by software. Stopping the CPU clock will freeze the USB communication channel.
- Also, it is possible to disable the HPI completely by software. This situation will also freeze the communication channel.
- The DARAM can only support 2 accesses per cycle, for which the CPU always has priority against DMA and HPI accesses. Some rare sequences of instructions, when executed in a tight loop require those two DARAM accesses per cycle, thereby completely denying access of the DARAM block from which the code executes to the DMA and HPI. This condition occurs only when the code is executing from the same DARAM block where the mail-box resides (the first block).

Under either of these conditions the USB controller is unable to transfer data to and from the HPI. More precisely, the USB controller waits indefinitely for the HPI to signal that it is ready to transfer data, which never happens. The USB communication and the PC application controlling the DSP board appear to freeze. No error message is passed to the user.

The easiest actions that can take the board out of this lock-up condition are:

- Disconnect and reconnect the USB connector or
- Cycle the board's power-supply.

Symbolic Access

The DSP access Vis, including the memory read and write Vis, and the DSP function execution Vis include full symbolic access. This means that memory reads and writes, as well as function execution can be directed to a symbol defined when the DSP code is created, rather than to an absolute address. This is a powerful feature, since it allows the PC code to be insensitive to a rebuild of the DSP code. Indeed, after a DSP code is rebuilt, the addresses of the accessed symbols may have changed, but the symbols stay the same, and therefore the PC-side of the code does not have to be rebuilt.

The symbolic access works by comparing the ASCII string of the specified symbol to a symbol table that is loaded in the *Global Board Information structure*, when the DSP code is loaded into memory. When a match is found, the symbol is replaced by its value and its memory space, as they are found in the symbol table. If no match is found, then an error is generated. To disable symbolic access, simply leave the ASCII string of the symbol empty. In this case, the access is made using the absolute address and memory space that are specified explicitly.

The symbol table corresponding to the DSP code is parsed and reloaded in the *Global Board Information structure* from the DSP's ".out" executable file every time a new DSP code is reloaded into DSP memory. A new symbol table can also be reloaded from DSP's ".out" file,

without actually modifying the DSP code. This is useful to gain symbolic access on a DSP code that is loaded and executed from Flash memory at power-up.

The following guidelines should be observed for the symbolic access to work properly:

- Only the global symbols found in the DSP's ".out" file are retained in the symbol table. This means that to allow symbolic access, variables declared in C should be declared as global (outside all blocks and functions, and without the *static* keyword). Variables and labels declared in assembly (function entry points for instance) should be declared with the directive *.global*.
- When a variable or a function entry point is declared in C, the C compiler appends an underscore at the beginning of the name. This underscore should not be omitted when specifying a symbolic name for an access.

In this new version of the software interfaces, the symbolic access feature has been expanded to allow access to structure members, with unlimited nesting capability. However, to benefit from this new feature, the following guidelines should also be observed:

- The symbolic name accessed should be constructed using symbolic member names, separated by ".", just as it is done in C:
  "*Global_Struct_Name.Member_Name_1.Member_Name_2…"* where:
  - *Global_Struct_Name* is the global name given to the instance of the primary structure to be accessed
  - *Member_Name_1* is the name of the member of the structure that is to be accessed
  - *Member_Name_2* is the name of the member of a structure nested within the primary structure, to be accessed.
- When structure members are declared in C, the C compiler appends an underscore at the beginning of the name. This underscore should not be omitted when specifying a symbolic member name for an access.
- The DSP code should be compiled with the option ***Full Symbolic Debugging***. Otherwise the structure information is not present in the ".out" executable file.

For instance, if structures are declared in the following manner:

```
struct      SignatureIndex      {       unsigned int        i_sample    [N_SigNibbles];
                                        unsigned int        i_channel   [N_SigNibbles];
                                        unsigned int        i_mask      [N_SigNibbles];
                                        };

#define     SigBufSize                  1000*6

struct      SigBuffer           {       unsigned int                    Error;
                                        unsigned int                    Fill;
                                        unsigned int                    Size;
                                        unsigned int                    InPoint;
                                        unsigned int                    OutPoint;
                                        unsigned int                    SigWord[SigBufSize];
                                        struct SignatureIndex           Index;
                                };

struct SigBuffer                        SigBuf;
```

Then, to access the *i_Channel* member of the *SignatureIndex* structure that is nested within the *SigBuffer* structure named *SigBuf*, the following symbol string should be used:

_SigBuf._SignatureIndex._i_Channel

Note the « _ » prefixing the structure name, as well as every member name.
Note : Symbolic access to a particular element of the i_Channel array is not allowed. However, all memory access Vis allow an offset to be specified. This byte-address offset is simply added to the base address computed from the symbolic name.

All memory access Vis allow an offset to be specified for the access. This byte-address offset is simply added to the base address specified explicitly (if symbolic access is disabled) or resolved from the specified symbolic name. This offset can be used to access a particular element of an array. However, care should be taken to the fact that this offset is always expressed as a number of byte-addresses, rather than a number of elements of the type specified for the access. This means that for accesses of 16-bit types the offset should be specified as a multiple of 2. For 32-bit types, the offset should be specified as a multiple of 4. This type-independent behaviour is required to allow the most flexibility when accessing arrays of undefined type (for instance arrays of structures).

Address Specification

Contrary to the previous Signal Ranger platform, the new *Signal_Ranger_mk2* platform (TMS320VC5502) and its EMIF support different addressing standards including byte, 16-bit-word, and 32-bit-word.
For simplicity and coherence, and because all the addresses that are described in a DSP's ".out" file are byte-addresses, all the interface software works with the **byte-addressing standard**. This means that:

- All the addresses provided by the ".out" file are byte-addresses. In particular, the addresses provided by the *SR2_Base_Resolve_UserSymbol* VI are byte-addresses.
- All symbolic addresses provided to all the access Vis (*SR2_Base_K_Exec, SR2_Base_Bulk_Move_Offset, SR2_Base_User_Move_Offset, SR2_Base_HPI_Move_Offset, SR2_Flash_FlashMove*) represent and point to byte-addresses
- The explicit addresses provided externally to all of the access Vis (*SR2_Base_K_Exec, SR2_Base_Bulk_Move_Offset, SR2_Base_User_Move_Offset, SR2_Base_HPI_Move_Offset, SR2_Flash_FlashMove*) must be byte-addresses.

However, it is important to note that:

- Because of hardware constraints in the HPI, all PC accesses (reads and writes) are performed in 16-bit words (2 bytes at a time). This means that access functions that transfer (read or write) individual bytes always read or write an even number of bytes. Byte arrays are padded if necessary.
- User code download is also subject to this constraint since it is performed using the same functions that read and write data. This should not cause a problem when developing in C because the compiler always creates sections that are aligned on an even byte-address. When developing in assembly, the ".even" directive should be used systematically at the beginning of any code section, to insure that code sections are aligned on even byte-addresses.
- Because of the same hardware constraints, read and write accesses are always required to occur on even byte-addresses. If an odd byte-address is passed to a transfer VI, it is truncated so that the effective transfer address is the previous even byte-address. However, this should never cause a problem because data is always aligned to an even byte-address in DSP memory by the linker.
- Entry points are not subject to this constraint, so the execution address passed to the *SR2_Base_K_Exec* VI may be an odd byte-address. Similarly, the *Branch Address* passed to the *SR2_Base_User_Move_Offset* may be an odd byte-address.
- Most interface Vis that perform transfers (*SR2_Base_Bulk_Move_Offset, SR2_Base_User_Move_Offset, SR2_Base_HPI_Move_Offset*) allow the addition of an offset to the transfer address. Just like the address, this offset is also a byte-address offset. Just like the address, the offset must also be even.
- Even though transfer addresses are always byte-addresses, the number of elements to transfer to and from the PC is always specified in number of elements **of the type being transferred**.

LabView Interface

The provided LabView interface is organized as several VI libraries. All the libraries with names ending in "_U" contain support Vis and it is not expected that the developer will have to use individual Vis in these libraries.

Altogether, the LabView interface allows the developer to leverage *Signal_Ranger_mk2's* real time processing and digital I/O capabilities, with the ease of use, high-level processing power and graphical user interface of LabView.

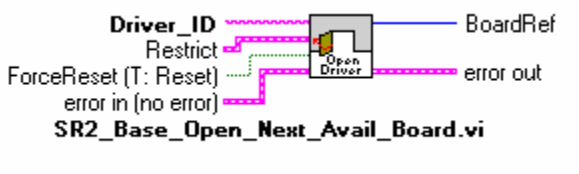
Core Interface Vis

These Vis are found in the *Sranger2.llb* library.

SR2_Base_Open_Next_Avail_Board

This Vi performs the following operations:

- Tries to find a DSP board with the selected driver ID that is connected, but presently free (not opened), on the PC.
- If it finds one, creates an entry in the *Global Board Information Structure*.
- Waits for the Power-Up kernel to be loaded on the board.
- If "ForceReset" is true, forces DSP reset, then reloads the Host-Download kernel.
- Places the symbol table of the present kernel in the "Global Board Info" structure.



Controls:

- **Driver ID**: This is a character string representing the base name for the selected board. For instance for *Signal Ranger_mk2* boards, this string must be set to *SRm2_*.
- **Restrict**: This is a structure used to restrict the access by Vendor ID, Product ID or Hardware Revision Number. If this control is wired, and if any of the members of the Restrict structure are changed from their default, the access is restricted to the boards having the selected parameters. Each member operates independently of the others. For instance if HardRev is set, but idVendor and idProduct are left to its default 0 value, then only DSP boards with the selected hardware revision will be opened, regardless of their Vendor or Product Ids.
- **ForceReset**: If true, the DSP is reset and the host-download kernel is loaded. All previously running DSP code is aborted.
- **Error In** LabView instrument-style error cluster. Contains error number and description. Leave it unwired if this is the first interface VI in the sequence.

Indicators:

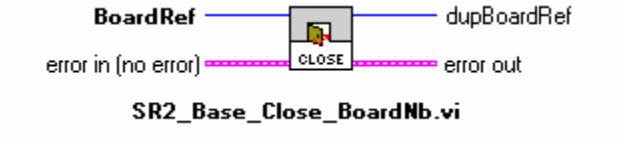
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the in *Global Board Information Structure*. All other interface Vis use this number to access the proper board.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

Note: The handle that the driver provides to access the board is exclusive. This means that only one application, or process, at a time can open and manage a board. A consequence of this is that a board cannot be opened twice. A board that has already been opened using the SR2_Base_Open_Next_Avail_Board VI cannot be opened again until it is properly closed using the SR2_Base_Close_BoardNb VI. This is especially a concern when the application managing

the board is closed under abnormal conditions. If the application is closed without properly closing the board. The next execution of the application will fail to find and open the board, simply because the corresponding driver instance is still open.

SR2_Base_Close_BoardNb

This Vi Closes the instance of the driver used to access the board, and deletes the corresponding entry in the *Global Board Information Structure*. Use it after the last access to the board has been made, to release Windows resources that are not used anymore.



Controls:

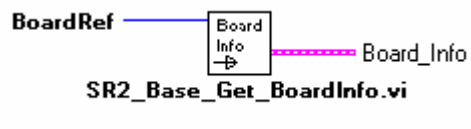
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Get_BoardInfo

This Vi returns the board entry from the *Global Board Information Structure*.



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi

Indicators:

- **BoardInfo**: Contains DSP board information. This information is described in section "How DSP Boards are Managed"

SR2_Base_Complete_DSP_Reset

This VI performs the following operations:

- Temporarily flashes the LED orange
- Resets the DSP
- Reinitializes HPIC
- Loads the Host-Download kernel

These operations are required to completely take control of a DSP that is executing other code or has crashed. The complete operation takes 500ms.

Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

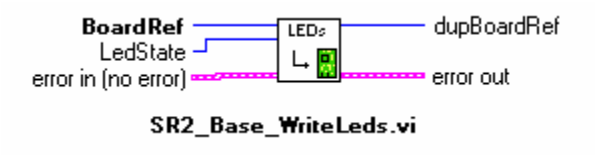
Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_WriteLeds

This Vi allows the selective activation of each element of the bi-color Led.

- Off
- Red
- Green
- Orange



Controls:

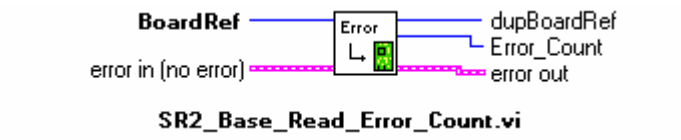
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **LedState**: This enum control specifies the state of the LEDs (Red, Green, Orange or Off).
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Read_Error_Count

The hardware of the USB controller contains an error counter. This 4-bit circular counter is incremented each time the controller detects a USB error (because of noise or other reason). The contents of this counter may be read periodically to monitor the health of the USB connection. Note that a USB error usually does not lead to a failed transaction. The USB protocol will retry packets that contain errors up to three times in a single transaction.



Controls:

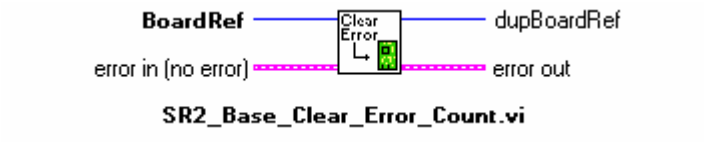
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Error_Count**: This is the value contained in the counter (between 0 and 15).
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Clear_Error_Count

This VI is provided to clear the 4-bit USB error counter.



Controls:

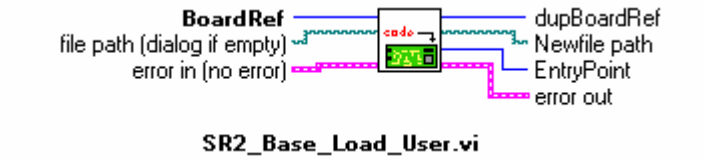
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Load_User

This VI loads a user DSP code into DSP memory. If *file path* is empty, a dialog box is used. The kernel has to be loaded prior to the execution of this VI. The DSP is reset prior to the load. After loading the code, the symbol table is updated in the *Global Board Information Structure*.
The VI checks if the type of COFF file is right for the target DSP. If not an error is generated.



Controls:

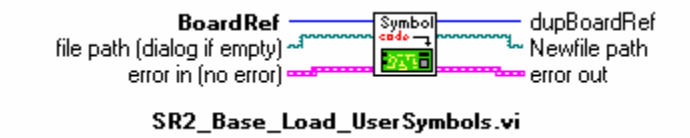
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **File path**: This is the file path leading to the COFF (.out) file of the DSP user code. A dialog box is presented if the path is empty.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **NewFile path**: This is the file path where the COFF file was found.
- **EntryPoint**: This is the address in DSP memory where execution should begin.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Load_UserSymbols

This VI loads the symbol table in the *Global Board Information Structure*. If *file path* is empty, a dialog box is used. Usually, this VI is used to gain symbolic access when code is already loaded and running on the DSP (for instance code that was loaded at power-up). It is not necessary to load symbols after executing SR2_Base_Load_User, or SR2_Base_LoadExec_User, since both Vis update the symbol table automatically.
The VI checks if the type of COFF file is right for the target DSP. If not an error is generated.



Controls:

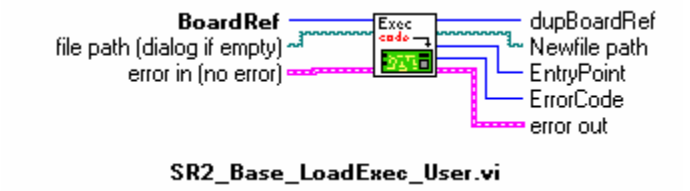
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **File path**: This is the file path leading to the COFF (.out) file of the DSP user code. A dialog box is presented if the path is empty.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **NewFile path**: This is the file path where the COFF file was found.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_LoadExec_User

This VI loads a user DSP code into DSP memory and runs it from the address of the entry point found in the COFF file. If *file path* is empty, a dialog box is used. The kernel has to be loaded prior to the execution of this VI. The DSP is reset prior to beginning the load. After loading the code, the symbol table is updated in the *Global Board Information Structure*.
The VI checks if the type of COFF file is right for the target DSP. If not an error is generated.
After completing the branch, the USB controller and the Vi waits for an acknowledge from the DSP, to complete its execution. If this signal does not occur within 5s, the Vi will abort and return an error. Normally the DSP code that is launched by this Vi should acknowledge the branch by asserting the *HINT* signal (see section about the DSP Communication Kernel).



Controls:

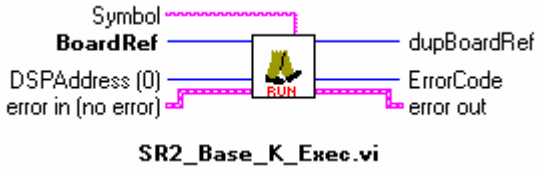
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **File path**: This is the file path leading to the COFF (.out) file of the DSP user code. A dialog box is presented if the path is empty.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **NewFile path**: This is the file path where the COFF file was found.
- **EntryPoint**: This is the address in DSP memory where execution should begin.
- **ErrorCode**: This is the error code, or completion code, returned by the user DSP function that is executed (function residing at the entry point). The kernel documentation mentions that the DSP function that is called (the entry function in this case) should contain an acknowledge to signal that the branch has been taken. Just prior to sending the acknowledge, the user DSP code has the opportunity to modify the ErrorCode field in the mailbox. This error code is sent back to the PC by the USB controller after it has received the acknowledge from the DSP. This error code is completely user-application-specific. It does not need to be managed by the interface. If the DSP code does not modify the Error Code, zero is returned.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_K_Exec

This Vi forces execution of the DSP code to branch to a specified address, passed in argument. If *Symbol* is wired and not empty, the Vi searches in the symbol table for the address corresponding to the symbolic label. If the symbol is not found, an error is generated. If *Symbol* is not wired, or is an empty string, the value passed in *DSPAddress* is used as the entry point.
The kernel must be loaded and executing for this Vi to be functional.
After completing the branch, the USB controller and the Vi waits for an acknowledge from the DSP, to complete its execution. If this signal does not occur within 5s, the Vi will abort and return an error. Normally the DSP code that is launched by this Vi should acknowledge the branch by asserting the *HINT* signal (see section about the DSP Communication Kernel).



SR2_Base_K_Exec.vi

Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **DSPAddress**: Physical branch address. It is used if for the branch if *Symbol* or *SymbolTable* are empty or left unwired.
- **Symbol**: Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* is used instead.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **ErrorCode**: This is the error code, or completion code, returned by the user DSP function that is executed. The kernel documentation mentions that the DSP function that is called should contain an acknowledge to signal that the branch has been taken. Just prior to sending the acknowledge, the user DSP code has the opportunity to modify the ErrorCode field in the mailbox. This error code is sent back to the PC by the USB controller after it has received the acknowledge from the DSP. This error code is completely user-application-specific. It does not need to be managed by the interface. If the DSP code does not modify the Error Code, zero is returned.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Bulk_Move_Offset

This VI reads or writes an unlimited number of data words to/from the program, data, or I/O space of the DSP, using the kernel. This transfer uses bulk pipes. The bandwidth is usually high (up to 22Mb/s for USB 2).
The VI is polymorphic, and allows transfers of the following types:

- Signed 8-bit bytes (I8), or arrays of this type.
- Unsigned 8-bit bytes (U8), or arrays of this type.
- Signed 16-bit words (I16), or arrays of this type.
- Unsigned 16-bit words (U16), or arrays of this type.
- Signed 32-bit words (I32), or arrays of this type.
- Unsigned 32-bit words (U32), or arrays of this type.
- 32-bit floating-point numbers (float), or arrays of this type.
- Strings

These represent all the basic data types used by the C compiler for the DSP.
To transfer any other type (structures for instance), the simplest method is to use a “cast” to allow this type to be represented as an array of U16 on the DSP side (cast the required type to an array of U16 to write it to the DSP, read an array of U16 and cast it back to the required type for a read).

The DSP address and memory space of the transfer are specified as follows:

- If Symbol is wired, and the symbol is represented in the symbol table, then the transfer occurs at the address and memory space corresponding to Symbol. Note that Symbol must represent a valid address. Also, the DSP COFF file must be linked with the usual page number convention:
  - Program space = page number 0
  - Data space = page number 1
  - IO space = page number 2

  All other page numbers are accessed as data space.
- If *Symbol* is unwired, then *DSPAddress* is used as the byte-address for the transfer, and *MemSpace* is used as the memory space. Note that *DSPAddress* is required to be even, to point to a valid 16-bit word (see section about Address Specification).
- The value of *Offset* is added to *DSPAddress*. This functionality is useful to access individual members of structures or arrays on the DSP. Note that the value of *Offset* is always counted in bytes (just as *DSPAddress* it is a byte-address offset, required to be even). This is required to access an individual member of an heterogeneous structure.
- In case of a write, if the accessed data type is an 8-bit type (I8, U8, or string), then an additional byte is appended if the number of bytes to transfer is odd. This is required because native transfers are 16-bit wide. The extra byte is set to FF_H.

- Even if the accessed data type is an 8-bit type (I8, U8, or string), the access is required to occur on an even byte address. This is because native transfers using the HPI are 16-bit wide.

Note: The kernel must be loaded and executing for this Vi to be functional.

Note: Since the VI is polymorphic, to read a specific type requires that this type be wired at the DataIn input. This simply forces the type for the read operation.

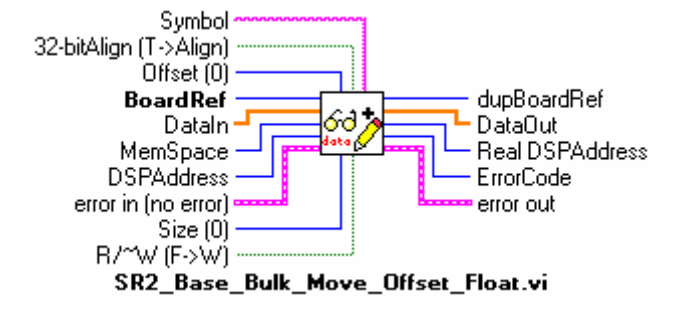
Note: When reading or writing scalars, Size must be left unwired.

The DSP's internal representation uses 16-bit words. When reading or writing 8-bit data, the bytes represent the high and low parts of 16-bit memory registers. They are presented MSB first and LSB next.
When reading, or writing 32 bit data words (I32, U32 or Float), the PC performs 2 separate accesses (at 2 successive memory addresses) for every transferred 32-bit word. In principle, the potential exists for the DSP or the PC to access one word in the middle of the exchange, thereby corrupting the data.
For instance, during a read, the PC could upload a floating-point value just after the DSP has updated one 16-bit word constituting the float, but before it has updated the other one. Obviously the value read by the PC would be completely erroneous.
Symmetrically, during a write, the PC could modify both 16-bit words constituting a float in DSP memory, just after the DSP has read the first one, but before it has read the second one. In this situation the DSP is working with an "old" version of half the float, and a new version of the other half.
These problems can be avoided if the following precautions are observed:

- When the PC accesses a group of values, it does so in blocks of up to 32 16-bit words at a time (up to 256 words if the board has enumerated on a USB_2-capable hub or root). Each of these block accesses is atomic. The DSP is uninterruptible and cannot do any operation in the middle of a block of the PC transfer. Therefore the DSP cannot "interfere" in the middle of any single 32 or 256 block access by the PC.
- This alone does not guarantee the integrity of the transferred values, because the PC can still transfer a complete block of data in the middle of another concurrent DSP operation on this same data. To avoid this situation, it is sufficient to also make atomic any DSP operation on 32-bit data that could be modified by the PC. This can easily be done by disabling DSPInt interrupts for the length of the operation for instance, then the PC accesses are atomic on both sides, and data can safely be transferred 32 bits at a time.

Note: Data is transferred between the PC and DSP in atomic blocks of 32 (Full-Speed USB connection) or 256 (High-Speed USB connection) words. To achieve this, the kernel function that performs the block transfer disables interrupts during the transfer. The time it takes to transfer 32 or 256 words of data to the DSP's on-chip RAM is usually very short (3.3ns/word). However, it may take much longer (up to 240ns/word) if the transfer is performed to or from SDRAM. In this case, the transfer time may be so long that it interferes with the service of other interrupts in the system. This can be a factor especially when the USB connection is high-speed, because in this case the block size is large (256 words). If this is the case, and transfer atomicity is not required, then a custom DSP function should be used to handle the transfer, instead of than the standard kernel function that is instantiated by SR2_Base_Bulk_Move_Offset. Such a custom transfer function can be called by the SR2_Base_User_Move_Offset. An example of the use of such a custom transfer function is provided in the examples directory. Another way to handle this case would be to split the transfer at high-level so that only small blocks are transferred at a time.

Note: Data is transferred between the PC and DSP in atomic blocks of 32 (Full-Speed USB connection) or 256 (High-Speed USB connection) words, only if the transfer does not cross a 64k words boundary. If the transfer does cross a boundary, the transfer is split at high-level so that both halves of the transfer occur in the same 64k page. In this condition the transfer loses its "atomicity".



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **DataIn**: data words to be written to DSP memory. DataIn must be wired, even for a read, to specify the data type to be transferred.
- **MemSpace**: Memory space for the exchange (data, program or IO). MemSpace is only used if Symbol is empty or left unwired.
- **DSPAddress**: Physical base DSP address for the exchange. DSPAddress is only used if Symbol is empty or left unwired.
- **32-bitAlign**: This control is provided for compatibility with other boards. On Signal Ranger_mk2 the control has no effect as long as the byte-address provided is even (as it must be).
- **Size**: Only used for reads of array types, represents the size (in number of items of the requested data type) of the array to be read from DSP memory. For Writes, the whole contents of *DataIn* are written to DSP memory, regardless of Size. When Size is wired, the data can only be transferred as arrays, not scalars.
- **Symbol**: Character string of the symbol to be accessed. If Symbol is empty or unwired, *DSPAddress* and *MemSpace* are used.
- **Offset**: Represents the offset of the data to be accessed, from the base address indicated by *Symbol* or *DSPAddress*. The actual access address is calculated as the sum of the base address and the offset. *Offset* is useful to access individual members of a structure, or an array.
- **R/~W**: Boolean indicating the direction of transfer (true->read, false->write).
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **DataOut**: Data read from DSP memory.
- **Real DSPAddress**: Actual address where the transfer took place. This address takes into account the resolution of *Symbol* (if used), and the effect of *Offset*.
- **ErrorCode**: This is the error code returned by the kernel function that is executed. Kernel reads always return a completion code of 1, kernel writes always return a completion code of 2.

- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_User_Move_Offset

This VI is similar to SR2_Base_Bulk_Move_Offset, except that it allows a user-defined DSP function to replace the intrinsic kernel function that SR2_Base_Bulk_Move_Offset uses.
The operation of the USB controller and the kernel allows a user-defined DSP function to override the intrinsic kernel functions (see kernel documentation below). For this, the user-defined DSP function must perform the same actions with the mailbox as the intrinsic kernel function would (kernel read or kernel write). This may be useful to define new transfer functions with application-specific functionality. For example, a function to read or write a FIFO could be defined this way. In addition to the data transfer functionality, a FIFO read or write function would also include the required pointer management that is not present in intrinsic kernel functions.

Accordingly, SR2_Base_User_Move_Offset includes two controls to define the entry point of the function that should be used to replace the intrinsic kernel function.
When the user-defined function is called, the mailbox contains:

- The BranchAddress field is set to the entry point of the function. This field is not normally used directly by the DSP function. It is set to its entry point, leading to its execution.
- The TransferAddress field. It is set to the address corresponding to Symbol, if Symbol is used, or to the user-defined 32-bit number DSPAddress if Symbol is not wired or is empty. Note that this field is not required to contain a valid address. For instance in the case of a FIFO management function, it could be a FIFO number instead.
- The NbWords field. This 16-bit number represents the number of words to transfer. It is always between 1 and 32 (Full-Speed USB connection), or between 1 and 256 (High-Speed USB connection). This represents the size of the data field of the mail-box. Tranfers of larger data blocks are segmented into transfers of up to 32 or 256 words.
- The ErrorCode field. It is set to 1 for reads, and to 2 for writes.
- In the case of a write transfer, from 1 to 32 words, as indicated by NbWords, (1 to 256 words for a High-Speed USB connection) have been written to the data field of the mailbox by the USB controller prior to the DSP function call.

The user-defined function should perform its function. If it is a read, it should read the required number of words from the data field of the mailbox. Then it should update the mailbox with the following:

- If it is a write it should provide the required number of words to the Data field of the mailbox.
- Then it may update the ErrorCode field of the mailbox with a completion code that is appropriate for the situation (this is the error code that is returned as the ErrorCode indicator of the VI).

After this, the user-defined function should simply send an acknowledge.

A transfer of a number of words greater than 32 (greater than 256 for a High-Speed USB connection) is segmented into as many 32-word (256-word) transfers as required. The user-defined function is called at every new segment. If the total number of words to transfer is not a multiple of 32 (256), the last segment contains the remainder.

The *TransferAddress* field of the mailbox is only initialized at the first segment. The user-defined function may choose to increment it to point to successive addresses (this is what the intrinsic kernel functions do), or may choose to leave it untouched (this would be appropriate if the field contains a FIFO number for instance). The way this field is managed is completely application-specific.

Note: If TransferAddress is used to transport information other than a real transfer address, the following restrictions apply:

- *The total size of the transfer must be smaller or equal to 32768 words. This is because transfers are segmented into 32768-word transfers at a higher level. The information in*

TransferAddress is only preserved during the first of these higher-level segments. At the next one, TransferAddress is updated as if it were an address to point to the next block.

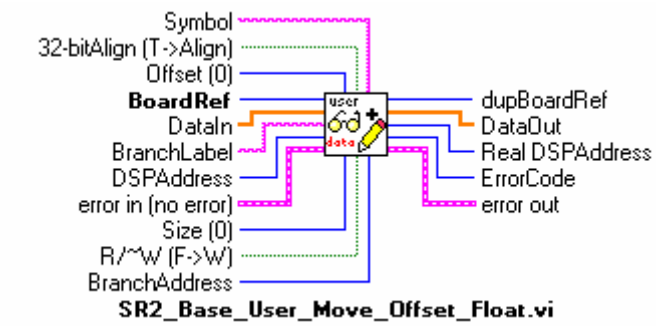
- *The transfer must not cross a 64 kWord boundary. Transfers that cross a 64 kWord boundary are split into two consecutive transfers. The information in TransferAddress is only preserved during the first of these higher-level segments. At the next one, TransferAddress is updated as if it were an address to point to the next block*
- *TransferAddress must be even. It is considered to be a byte transfer address, consequently its bit 0 is masked at high-level.*

The NbWords field is initialized with the size of the segment transferred, at each segment.
The ErrorCode field is is only initialized at the first segment with the value 1 (read) or 2 (write).
The value that is returned to the ErrorCode indicator of the VI is the value that may be updated by the user-defined function before acknowledging the LAST segment. If the user-defined function does not update the Error Code, the same value (1 for reads, and 2 for writes) is returned back to the PC.

Note: The kernel AND the DSP code of the user-defined function must be loaded and executing for this Vi to be functional.

Note: The MemSpace control of the VI is not used. The user-defined function performs the type of access that is coded within the function regardless of the value of MemSpace.

Note: The value of the R/~W indicator is reflected by the contents of the ErrorCode field of the mailbox at the entry of the user-defined function. For reads, the value is 1, for writes the value is 2.



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **DataIn**: data words to be written to DSP memory. DataIn must be wired, even for a read, to specify the data type to be transferred.
- **MemSpace**: Not used
- **DSPAddress**: Physical base DSP address for the exchange. DSPAddress is only used if Symbol is empty or left unwired. DSPAddress is written to the TransferAddress field of the mailbox before the first call of the user-defined DSP function (the call corresponding to the first segment). DSPAddress is not required to represent a valid address. It may be used to transmit an application-specific code (a FIFO number for instance).
- **32-bitAlign**: This control is provided for compatibility with other boards. On Signal Ranger_mk2 the control has no effect as long as the byte-address provided is even (as it must be).

- **Size**: Only used for reads of array types, represents the size (in number of items of the requested data type) of the array to be read from DSP memory. For Writes, the whole contents of *DataIn* are sent to DSP memory, regardless of Size. When Size is wired, the data can only be transferred as arrays, not scalars.
- **Symbol**: Character string of the symbol to be accessed. If Symbol is empty or unwired, *DSPAddress* and *MemSpace* are used.
- **Offset**: Represents the offset of the data to be accessed, from the base address indicated by *Symbol* or *DSPAddress*. The actual access address is calculated as the sum of the base address, and the offset. *Offset* is useful to access individual members of a structure, or an array. If DSPAddress is used to transport application-specific data, Offset should not be connected.
- **R/~W**: Boolean indicating the direction of transfer (true->read, false->write).
- **BranchLabel**: Character string corresponding to the label of the user-defined function. If *BranchLabel* is empty or unwired, *BranchAddress* is used instead.
- **BranchAddress**: Physical base DSP address for the user-defined function.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

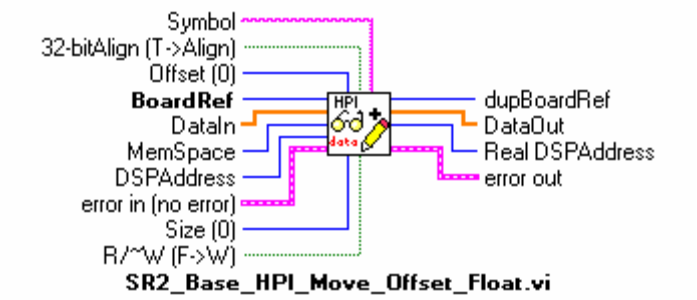
Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **DataOut**: Data read from DSP memory.
- **Real DSPAddress**: Actual address where the transfer took place. This address takes into account the resolution of *Symbol* (if used), and the effect of *Offset*.
- **ErrorCode**: The value that is returned through the ErrorCode indicator is the value that is updated by the user-defined DSP function before acknowledging the LAST segment transferred. If the user function does not update the Error Code, the function returns 1 for reads, and 2 for writes.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_HPI_Move_Offset

This VI is similar to SR2_Base_Bulk_Move_Offset, except that is relies on the hardware of the HPI, rather than the kernel to perform the transfer.
Transfers are limited to the on-chip RAM that is directly accessible via the HPI (byte-addresses $00C0_H$ to $FFFF_H$). The MemSpace control is not used. This VI will perform transfers into and out of the data and program space. This VI will transfer to any address accessible via the HPI, regardless of memory space. The on-chip I/O space is not accessible. If an attempt is made to write data outside the allowed range, the data is not written. If an attempt is made to read data from outside the allowed range erroneous data is returned.

Note: The kernel does not need to be loaded or functional for this VI to execute properly. This VI will complete the transfer even if the DSP has crashed, making it a good debugging tool. Transfers with the HPI use the control pipe 0 instead of the fast bulk pipes used by SR2_Base_Bulk_Move_Offset. The bandwidth for such transfers is typically low (500kb/s for USB 1.1). However it is guaranteed.



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **DataIn**: Data words to be written to DSP memory. DataIn must be wired, even for a read, to specify the data type to be transferred.
- **MemSpace**: Memory space for the exchange (data, program or IO). MemSpace is only used if Symbol is empty or left unwired.

Note: the memory space selection is not used. The actual transfer uses the hardware of the HPI, regardless of the memory space selection.

- **DSPAddress**: Physical base DSP address for the exchange. DSPAddress is only used if Symbol is empty or left unwired.
- **32-bitAlign**: This control is provided for compatibility with other boards. On Signal Ranger_mk2 the control has no effect as long as the byte-address provided is even (as it must be).
- **Size**: Only used for reads of array types, represents the size (in number of items of the requested data type) of the array to be read from DSP memory. For Writes, the whole contents of *DataIn* are written to DSP memory, regardless of Size. When Size is wired, the data can only be transferred as arrays, not scalars.
- **Symbol**: Character string of the symbol to be accessed. If Symbol is empty or unwired, *DSPAddress* is used.
- **Offset**: Represents the offset of the data to be accessed, from the base address indicated by *Symbol* or *DSPAddress*. The actual access address is calculated as the sum of the base address, and the offset. *Offset* is useful to access individual members of a structure, or an array.
- **R/~W**: Boolean indicating the direction of transfer (true->read, false->write).
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

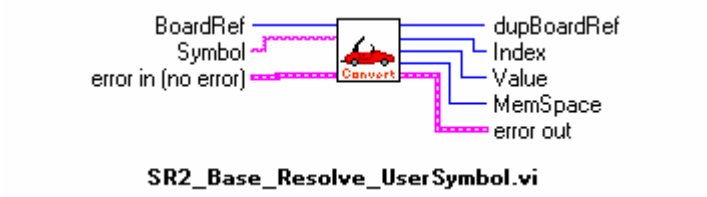
Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **DataOut**: Data read from DSP memory.
- **Real DSPAddress**: Actual address where the transfer took place. This address takes into account the resolution of *Symbol* (if used), and the effect of *Offset*.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Resolve_UserSymbol

This Vi may be used to provide the address corresponding to a particular symbol in the symbol table of the presently loaded DSP code. Used in conjunction with the data transfer Vis (SR2_Base_Bulk_Move_Offset, SR2_Base_User_Move_Offset and SR2_Base_HPI_Move_Offset) it allows more flexibility for choosing the actual transfer address.

For instance the address may be transformed in an application-specific manner prior to the transfer.



Controls:

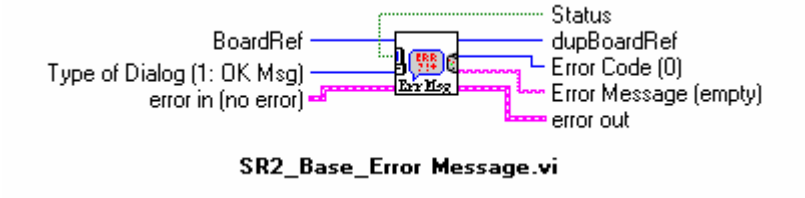
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Symbol**: Character string of the symbol to be accessed.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Index**: This is the index of the symbol in the symbol table.
- **Value**: This is the resolution value of the symbol.
- **MemSpace**: This is the memory space for the symbol. Note that this value actually reflects the “page number” that was used at link time for this symbol. For this number to indicate a valid memory space, the symbol must be linked using the usual convention:
  - 0 -> Program Space
  - 1 -> Data Space
  - 2 -> IO Space
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Base_Error Message

This Vi may be used to provide a text description of an error generated by a VI of the interface. It is similar to a traditional LabVIEW Error Message VI, except that it contains all the error codes that may be generated by the interface, in addition to normal LabVIEW error codes.



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Type of Dialog**: Selects one of 3 standard behaviours for error management.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Error Code**: Provides the Error Code.

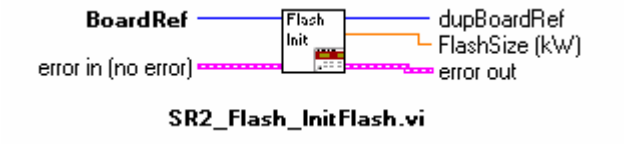
- **Error Message**: Provides a text explanation of the error if one exists.
- **Status**: Boolean, indicates if there is an error (True) or not (False).
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

Flash Support Vis

These Vis are found in the *SR2_Flash.llb* library.
These Vis are provided to support Flash-programming operations. The Vis also cover Flash-reading operations for symmetry. However, the SR2_Base_Bulk_Move_Offset VI can perfectly be used to read the Flash, and does not require the presence of Flash support code.

SR2_Flash_InitFlash

This Vi downloads and runs the Flash support DSP code. The DSP is reset as part of the download process. All DSP code is aborted. The Flash support code must be running in addition to the kernel to support Flash programming Vis. The VI also detects the Flash, and if it finds one it returns its size in kWords. If no Flash is detected, the size indicator is set to 0.



Controls:
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

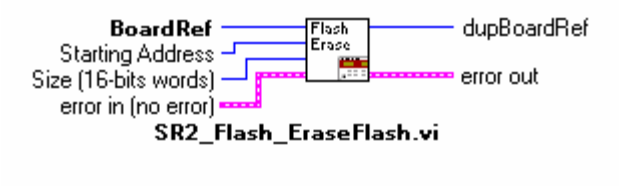
Indicators:
- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **FlashSize**: This indicator returns the size of the Flash detected. If no Flash is detected, it returns zero.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Flash_EraseFlash

This Vi erases the required number of 16-bit words from the Flash, starting at the selected address. The erasure proceeds in sectors therefore more words may be erased that are actually selected. For instance, if the starting address is not the first word of a sector, words before the starting address will be erased, up to the beginning of the section. Similarly, if the last word selected for erasure is not the last word of a section, additional words will be erased, up to the end of the last selected sector. The erasure is such that the selected words, including the starting address, are always erased.

Note: The sector size is 32 kwords.

Note: Erasure should only be attempted in the sections of the memory map that contain Flash. Erasure attempts outside the Flash will fail.



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **Starting Address**: Address of the first word to be erased.
- **Size**: Number of words to be erased.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

SR2_Flash_FlashMove

This VI reads or writes an unlimited number of data words to/from the Flash memory. Note that if only Flash memory reads are required the VI SR2_Base_Bulk_Move_Offset should be used instead (specifying program memory), since it does not require the presence of the Flash support code.

The VI is polymorphic, and allows transfers of the following types:

- Signed 8-bit bytes (I8), or arrays of this type.
- Unsigned 8-bit bytes (U8), or arrays of this type.
- Signed 16-bit words (I16), or arrays of this type.
- Unsigned 16-bit words (U16), or arrays of this type.
- Signed 32-bit words (I32), or arrays of this type.
- Unsigned 32-bit words (U32), or arrays of this type.
- 32-bit floating-point numbers (float), or arrays of this type.
- Strings

These represent all the basic data types used by the C compiler for the DSP.
To transfer any other type (structures for instance), the simplest method is to use a “cast” to allow this type to be represented as an array of U16 on the DSP side (cast the required type to an array of U16 to write it to the DSP, read an array of U16 and cast it back to the required type for a read).

An attempt to write outside of the Flash memory will result in failure.

The writing process can change ones into zeros, but not change zeros back into ones. If a write operation is attempted that should result in a zero turning back into a one, then it results in failure. Normally an erasure should be performed prior to the write, so that all the bits of the selected write zone are turned back into ones.

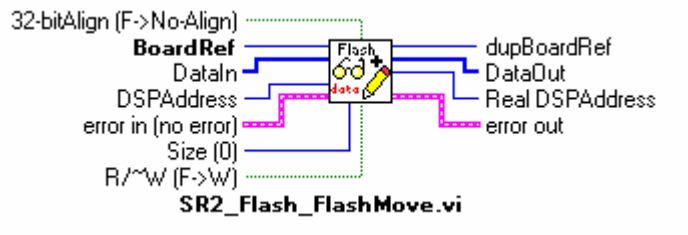
Note: Contrary to previous generations of Flash devices that have been used in Signal Ranger boards, the Flash device used in Signal_Ranger_mk2 cannot be incrementally programmed. This means that a word location that has been previously programmed MUST be erased before reprogramming. This is true even if the reprogramming operation is only intended to turn some of the remaining “1s” into “0s”.

In case of a write, if the accessed data type is an 8-bit type (I8, U8, or string), then an additional byte is appended if the number of bytes to transfer is odd. This is required because native transfers are 16-bit wide. The extra byte is set to FF_H.

Since the VI is polymorphic, to read a specific type requires that this type be wired to the DataIn input. This simply forces the type for the read operation.

Note: When reading or writing scalars, Size must be left unwired.

The DSP's internal representation uses 16-bit words. When reading or writing 8-bit data, the bytes represent the high and low parts of 16-bit memory registers. They are presented MSB first and LSB next.



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **DataIn**: Data words to be written to Flash memory. DataIn must be wired, even for a read, to specify the data type to be transferred.
- **DSPAddress**: Physical base DSP address for the transfer.
- **32-bitAlign**: This control is provided for compatibility with other boards. On Signal Ranger_mk2 the control has no effect as long as the byte-address provided is even (as it must be).
- **Size**: Only used for reads of array types, represents the size (in number of items of the requested data type) of the array to be read from DSP memory. For Writes, the whole contents of *DataIn* are written to DSP memory, regardless of Size. When Size is wired, the data can only be transferred as arrays, not scalars.
- **R/~W**: Boolean indicating the direction of transfer (true->read, false->write).
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

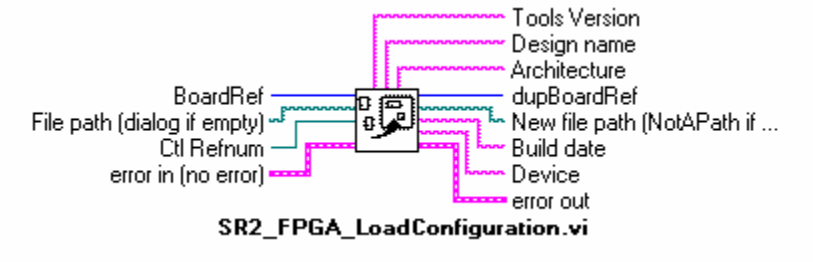
- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **DataOut**: Data read from Flash memory.
- **Real DSPAddress**: Actual address where the transfer took place. This address takes into account the effect of *Offset*.
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

FPGA Support Vis

These Vis are found in the *SR2_FPGA.llb* library.

SR2_FPGA_LoadConfiguration

This Vi downloads an "*.rbt" logic configuration file into the FPGA. The DSP is reset prior to the download. All DSP code is aborted. The .rbt file must be valid, and must be correct for the specified FPGA. Loading an invalid rbt file into an FPGA may damage the part.



Controls:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi
- **File path**: This is the file path leading to the ".rbt" file describing the FPGA logic. A dialog box is presented if the path is empty.
- **Ctl Refnum**: This is a refnum on the progress bar that is updated by the VI. This way, a progress bar may be displayed on the front-panel of the calling VI.
- **Error in**: LabView instrument-style error cluster. Contains error number and description of the previously running Vi.

Indicators:

- **DupBoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_Base_Open_Next_Avail_Board.vi Use this output to propagate the reference number to other Vis.
- **New file path**: This is the file path where the ".rbt" file describing the FPGA logic was found.
- **Tools Version**: ASCII chain indicating the version of the tools that were used to generate the ".rbt" file.
- **Design Name**: ASCII chain indicating the name of the logic design.
- **Architecture**: ASCII chain indicating the type of device targeted by the ".rbt" file
- **Device**: ASCII chain indicating the device number targeted by the ".rbt" file
- **Build Date**: ASCII chain indicating the build date for the ".rbt" file
- **Error out**: LabView instrument-style error cluster. Contains error number and description.

Error Codes

The following table lists the error codes that may be returned by the various Vis of the LabVIEW interface.

Note: The list is not complete. Only the most usual codes are listed.

Error Nb	Cause
00000002h	Memory is full
00000003h	Wrong memory zone accessed
00000004h	End-of-file encountered
00000005h	File already open
00000006h	File I/O error
00000007h	File not found
00000008h	File permission error
00000009h	Disk full

0000000Ah	Duplicate path
0000000Bh	Too many files open
BFFC0804h	No Board Detected
BFFC0805h	Open board generate error
BFFC0806h	Board info generate error
BFFC0807h	Close board generate error
BFFC0808h	DSP File not accessible
BFFC0809h	Reset DSP generate error
BFFC0812h	This DSP kernel is not for c5000
BFFC080Ah	Unreset_DSP genrate error
BFFC080Bh	DSP file not found
BFFC080Ch	DSP file not valid
BFFC080Dh	DSP file type not supported
BFFC080Eh	DSP kernel does not compare
BFFC080Fh	Write_hpi generate error
BFFC0810h	Read_hpi generate error
BFFC0811h	This DSP file is not for c5000
BFFC0813h	Write and read test error
BFFC0814h	DSP did not return… may have crashed
BFFC0815h	Symbol not found in the table
BFFC0816h	DSPInt generate error
BFFC0817h	DSP Timeout
BFFC0818h	Kernel not loaded
BFFC0820h	Close driver error
3FFC0104h	Error query not supported
BFFC0821h	Open driver error
BFFC0822h	Data move with kernel generates error
BFFC0823h	Exec low level generates error
BFFC0824h	Get pipe info generates error
BFFC0825h	Reset DSP generates error
BFFC0826h	HPI move generates error
BFFC0827h	DSP int generates error
BFFC0828h	Get configuration descriptor error
BFFC0829h	Write Led generates error
BFFC082Ah	Get device descriptor error
BFFC082Bh	Read and write HPI control error
BFFC082Ch	Change USB Timeout mode error
BFFC082Dh	DSP code section does not check
BFFC082Eh	FPGA file not valid
BFFC082Fh	FPGA configuration aborted before the end
BFFC0830h	Driver access error
BFFC0831h	DSP family not recognized or board not open
BFFC0832h	FPGA file not for target device family
BFFC0833h	Section too large to fit in Flash
BFFC0834h	DSP File may not be linked
BFFC0835h	Section not aligned on an even byte address
BFFC0836h	Erase Flash Error
BFFC0837h	Write Flash Error

Example Code

Examples are provided to accelerate user development. These examples include the PC side, and the DSP side of the code. On the DSP side, they include code written in assembly, as well as code written in C. The LabVIEW side of these examples are contained in the LabVIEWDemo.llb library in the main install directory. The DSP-side of these examples are zipped and stored in the following directory:
C:\Program Files\ SignalRanger_mk2\Examples\LabVIEW_Examples_DSPCode\ LabviewInterfaceDemo.zip.
Simply unzip *each file* to deflate the corresponding example as well as documentation.

C/C++ Interface.

This interface has been designed with C/C++ development in mind, and has only been tested on the ".net" version of Microsoft's Visual Studio. However, it may be possible to use it with other development environments allowing the use of DLLs.
The C/C++ interface is provided in the form of a DLL called *SRm2_HL.dll*. Compared to the LabVIEW interface, this interface provides a slightly more limited functionality. However, it covers the essentials.

The aspects of the C/C++ interface that have been limited in comparison to the LabVIEW interface are as follows:

- The LabVIEW interface provides polymorphic Vis to exchange data between the DSP and the host PC. The C/C++ DLL only provides one function to exchange arrays of I16 (short) type. This is not limiting because at low-level the transfers are performed using arrays of I16 data. To exchange other types, the developer simply needs to cast these arrays of I16 data into other types.
- The VI SR2_Base_Get_BoardInfo in the LabVIEW interface does not have an equivalent in the C/C++ interface.
- The VI SR2_Base_ErrorMessage in the LabVIEW interface does not have an equivalent in the C/C++ interface.

To work at run-time, this DLL requires that the following files, be in the same directory as the user-mode application that uses it:

- SRm2_HL.dll - The main DLL supporting the API
- SRanger2.dll - A low-level support DLL
- SR2Kernel_HostDownload.out - The "Host-Download" DSP kernel code
- SR2Kernel_PowerUp.out - The "Power-Up" DSP kernel code
- SR2_Flash_Support.out - The Flash Support DSP code
- SR2_FPGA_Support.out - The FPGA support DSP code

Furthermore, the LabVIEW 7.1 run-time engine must be installed on the computer that needs to use the DLL. The LabVIEW 7.1 run-time engine is installed automatically by the software installation described at the beginning of this document. However, if the user wants to deploy an application using the C/C++ interface, which is required to run on computers other than those on which it was developped, the LabVIEW 7.1 run-time engine should be installed separately on those computers. A run-time engine installer is available for free from the National Instruments web site www.ni.com.

An example is provided, which covers the development of code in Visual C/C++. This example is discussed at the end of this chapter.

Execution Timing And Thread Management

Two functions of the *SRm2_HL* DLL accessing **the same** *Signal_Ranger_mk2* board cannot execute concurrently. The previous function must have completed before a new one can be called. Care should be taken in multi-threaded environments to ensure that all functions of the interface accessing the same board do not run at the same time (in different threads). The simplest method is to ensure that functions accessing the same board are executed in the same thread. However, functions of the interface accessing different boards can be called concurrently. All the functions of the interface perform an access to a *Signal_Ranger_mk2* board via its USB 2.0 connection. The timing parameters (access time and throughput) depend a lot on the type of connection (Full-Speed or High-Speed), and on the type of USB Host-Controller software on the PC. They also depend to some extent on the USB traffic with other peripherals that may be connected to the same USB root (Printer, Modem, Hard-Disk, Web-Cam…etc.), as well as on the speed of the PC. Note that some of these other peripherals may use a large amount of the

available USB bandwidth, and may lower the effective data transfer performance between the PC and the *Signal_Ranger_mk2* DSP board. USB bandwidth is shared among all the USB peripherals connected to the same USB controller.
The *Signal_Ranger_mk2* board has a USB 2.0 port. It connects at High-Speed (480mb/s) on any USB 2.0-capable root or hub. It connects at Full-Speed (12Mb/s) on any root or hub that is only USB 1.1-capable.
All the functions of the *SRm2_HL* DLL are blocking. This means that they do not return until the requested action has been performed on the board.
None of the functions of the DLL can be blocked indefinitely. If the function cannot perform the requested operation within a few seconds, a time-out occurs and the function returns with a non-zero error code.

Further details about transfer speed benchmarks can be found in the *Under the Hood – USB Benchmarks* section.

Calling Conventions

The functions are called using the C calling conventions, rather than the standard Windows API (Pascal) conventions.
All the functions return a *USB Error Code* in the form of a 32-bit signed integer (long). This error code is zero if no error occurred. If it is non-zero, its value indicates the type of error. The *Error Codes* section lists the error codes. They are the same for the C/C++ interface as for the LabVIEW interface.
Whenever a function must return a number, array or string, the corresponding space (of sufficient size) must be allocated by the caller, and a reference to this space must be passed to the function.
Whenever a function must return an element of variable size (an array or a string), the size of the element that has been allocated by the caller is also passed to the function. This size argument is the argument immediately following the pointer to the array or string in the calling line.

Building A Project Using Visual C++ ".Net"

To build a project using Visual C++ ".net" the following guidelines should be followed. An example is provided to accelerate the learning curve.

- If the project is linked statically to the *SRm2_HL.lib* library, it must be loaded using the DELAYLOAD function of Visual C++. To use DELAYLOAD, add *delayimp.lib* to the project (in Visual C++ ".net" it can be found in *Program Files\Microsoft Visual Studio .NET 2003\Vc7\lib*\); in *Project Properties*, under *Linker\Command Line\Additional Options*, add the command */DELAYLOAD:SRm2_HL.dll*.
- Alternately, the DLL may be loaded dynamically using *LoadLibrary* and DLL functions must be called using *GetProcAddress*. Do not link statically with the *SRm2_HL.lib* library without using the DELAYLOAD function.
- Add *#include "SRm2_HL.h"* in the main.
- If using the DELAYLOAD function to link statically to the *SRm2_HL.lib* library, add *SRm2_HL.lib* to the project.
- The following files must be placed in the folder containing the project sources:
  - extcode.h
  - fundtypes.h
  - platdefines.h
  - SRm2_HL.lib

Notes: All these files are found with the example that is provided. See example section below.

Exported Interface Functions

SR2_DLL_Open_Next_Avail_Board

```
long __cdecl SR2_DLL_Open_Next_Avail_Board(char Driver_ID[],
      unsigned short ForceReset, unsigned short idVendor_Restrict,
      unsigned short idProduct_Restrict, char HardRev_Restrict[],
      long *BoardRef);
```

Description:
This function performs the following operations:

- Tries to find a DSP board with the selected driver ID that is connected, but presently free, on the PC.
- If it finds one, creates an entry in the *Global Board Information Structure*.
- Waits for the Power-Up kernel to be loaded on the board.
- If "ForceReset" is true, forces DSP reset, then reloads the Host-Download kernel.
- Places the symbol table of the present kernel in the "Global Board Info" structure.

Inputs:

- **Driver ID**: This is a character string representing the base name for the selected board. For instance for *Signal Ranger_mk2* boards, this string must be set to *SRm2_*.
- **ForceReset**: Set to 1 to reset the DSP and load the Host-Download kernel. In this case all previously running code is aborted. Set to 0 to leave the DSP undisturbed. In this case, the code that may be already running (code launched by the Power-Up kernel for instance) continues running.
- **idVendor_Restrict**: This argument is used to restrict the access by Vendor ID. If this control non-zero, the access is restricted to the boards having the selected Vendor ID. Set to zero to avoid restriction.
- **idProduct_Restrict**: This argument is used to restrict the access by Product ID. If this control non-zero, the access is restricted to the boards having the selected Product ID. Set to zero to avoid restriction.
- **HardRev_Restrict**: This argument is used to restrict the access by the firmware revision of the USB controller. If this string is not empty, the access is restricted to the boards having the selected string for a hardware revision. Leave the string empty to avoid restriction.

Outputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the in *Global Board Information Structure*. All other interface functions use this number to access the proper board.

Note: The handle that the driver provides to access the board is exclusive. This means that only one application, or process, at a time can open and manage a board. A consequence of this is that a board cannot be opened twice. A board that has already been opened using the SR2_DLL_Open_Next_Avail_Board function cannot be opened again until it is properly closed using the SR2_DLL_Close_BoardNb function. This is especially a concern when the application managing the board is closed under abnormal conditions. If the application is closed without properly closing the board. The next execution of the application will fail to find and open the board, simply because the corresponding driver instance is still open.

SR2_DLL_Close_BoardNb

```
long __cdecl SR2_DLL_Close_BoardNb(long BoardRef);
```

Description:
This function Closes the instance of the driver used to access the board, and deletes the corresponding entry in the *Global Board Information Structure*. Use it after the last access to the board has been made, to release Windows resources that are not used anymore.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board

Outputs:
N/A

SR2_DLL_Complete_DSP_Reset

```
long __cdecl SR2_DLL_Complete_DSP_Reset(long BoardRef);
```

Description:
This function performs the following operations:

- Temporarily flashes the LED orange
- Resets the DSP
- Reinitializes HPIC
- Loads the Host-Download kernel

These operations are required to completely take control of a DSP that is executing other code or has crashed. The complete operation takes 500ms.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board

Outputs:
N/A

SR2_DLL_WriteLeds

```
long __cdecl SR2_DLL_WriteLeds(long BoardRef, unsigned short LedState);
```

Description:
This function allows the selective activation of each element of the bi-color Led.

- Off
- Red
- Green
- Orange

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.
- **LedState**: This enum control specifies the state of the LEDs (0->Off, 1->Red, 2->Green, 3->Orange).

Outputs:
N/A

SR2_DLL_Read_Error_Count

```
long __cdecl SR2_DLL_Read_Error_Count(long BoardRef,
     unsigned char *Error_Count);
```

Description:
The hardware of the USB controller contains an error counter. This 4-bit circular counter is incremented each time the controller detects a USB error (because of noise or other reason).

The contents of this counter may be read periodically to monitor the health of the USB connection. Note that a USB error usually does not lead to a failed transaction. The USB protocol will retry packets that contain errors up to three times in a single transaction.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.

Outputs:

- **Error_Count**: This is the value contained in the counter (between 0 and 15).

SR2_DLL_Clear_Error_Count

```
long __cdecl SR2_DLL_Clear_Error_Count(long BoardRef);
```

Description:
This function is provided to clear the 4-bit USB error counter.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.

Outputs:
N/A

SR2_DLL_Load_User

```
long __cdecl SR2_DLL_Load_User(long BoardRef, char file_path[],
      unsigned long *EntryPoint);
```

Description:
This function loads a user DSP code into DSP memory. If *file_path* is empty, a dialog box is used. The kernel has to be loaded prior to the execution of this function. The DSP is reset prior to the load. After loading the code, the symbol table is updated in the *Global Board Information Structure*.
The function checks if the type of COFF file is right for the target DSP. If not an error is generated.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.
- **File path**: This is the file path leading to the COFF (.out) file of the DSP user code. A dialog box is presented if the path is empty.

Outputs:

- **EntryPoint**: This is the address in DSP memory where execution should begin.

SR2_DLL_Load_UserSymbols

```
long __cdecl SR2_DLL_Load_UserSymbols(long BoardRef, char file_path[]);
```

Description:
This function loads the symbol table in the *Global Board Information Structure*. If *file_path* is empty, a dialog box is used. Usually, this function is used to gain symbolic access when code is already loaded and running on the DSP (for instance code that was loaded at power-up). It is not necessary to load symbols after executing SR2_DLL_Load_User, or SR2_DLL_LoadExec_User, since both functions update the symbol table automatically.

The function checks if the type of COFF file is right for the target DSP. If not an error is generated.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.
- **File path**: This is the file path leading to the COFF (.out) file of the DSP user code. A dialog box is presented if the path is empty.

Outputs:
N/A

SR2_DLL_LoadExec_User

```
long __cdecl SR2_DLL_LoadExec_User(long BoardRef, char file_path[],
      unsigned long *EntryPoint, unsigned short *ErrorCode);
```

Description:
This function loads a user DSP code into DSP memory and runs it from the address of the entry point found in the COFF file. If *file_path* is empty, a dialog box is used. The kernel has to be loaded prior to the execution of this function. The DSP is reset prior to beginning the load. After loading the code, the symbol table is updated in the *Global Board Information Structure*.
The function checks if the type of COFF file is right for the target DSP. If not an error is generated.
After completing the branch, the USB controller and the function wait for an acknowledge from the DSP, to complete its execution. If this signal does not occur within 5s, the function will abort and return an error. Normally the DSP code that is launched by this function should acknowledge the branch by asserting the *HINT* signal (see section about the DSP Communication Kernel).

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.
- **File path**: This is the file path leading to the COFF (.out) file of the DSP user code. A dialog box is presented if the path is empty.

Outputs:

- **EntryPoint**: This is the address in DSP memory where execution should begin.
- **ErrorCode**: This is the error code, or completion code, returned by the user DSP function that is executed (function residing at the entry point). The kernel documentation mentions that the DSP function that is called (the entry function in this case) should contain an acknowledge to signal that the branch has been taken. Just prior to sending the acknowledge, the user DSP code has the opportunity to modify the ErrorCode field in the mailbox. This error code is sent back to the PC by the USB controller after it has received the acknowledge from the DSP. This error code is completely user-application-specific. It does not need to be managed by the interface. If the DSP code does not modify the Error Code, zero is returned.

SR2_DLL_K_Exec

```
long __cdecl SR2_DLL_K_Exec(long BoardRef, char Symbol[],
      unsigned long DSPAddress, unsigned short *ErrorCode);
```

Description:
This function forces execution of the DSP code to branch to a specified address, passed in argument. If *Symbol* is not empty, the function searches in the symbol table for the address corresponding to the symbolic label. If the symbol is not found, an error is generated. If *Symbol* is an empty string, the value passed in *DSPAddress* is used as the entry point.

The kernel must be loaded and executing for this function to work.
After completing the branch, the USB controller and the function wait for an acknowledge from the DSP, to complete its execution. If this signal does not occur within 5s, the function will abort and return an error. Normally the DSP code that is launched by this function should acknowledge the branch by asserting the *HINT* signal (see section about the DSP Communication Kernel).

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.
- **Symbol**: Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* is used instead.
- **DSPAddress**: Physical branch address. It is used if for the branch if *Symbol* is empty.

Outputs:

- **ErrorCode**: This is the error code, or completion code, returned by the user DSP function that is executed (function residing at the entry point). The kernel documentation mentions that the DSP function that is called (the entry function in this case) should contain an acknowledge to signal that the branch has been taken. Just prior to sending the acknowledge, the user DSP code has the opportunity to modify the ErrorCode field in the mailbox. This error code is sent back to the PC by the USB controller after it has received the acknowledge from the DSP. This error code is completely user-application-specific. It does not need to be managed by the interface. If the DSP code does not modify the Error Code, zero is returned.

SR2_DLL_Bulk_Move_Offset_I16

```
long __cdecl SR2_DLL_Bulk_Move_Offset_I16(long BoardRef,
      unsigned short ReadWrite, char Symbol[], unsigned long DSPAddress,
      unsigned short MemSpace, unsigned long Offset, short Data[],
      long Size, unsigned short *ErrorCode);
```

Description:
This function reads or writes an unlimited number of data words to/from the program, data, or I/O space of the DSP, using the kernel. This transfer uses bulk pipes. The bandwidth is usually high (up to 22Mb/s for USB 2).

The DSP address and memory space of the transfer are specified as follows:

- If *Symbol* is wired, and the symbol is represented in the symbol table, then the transfer occurs at the address and memory space corresponding to *Symbol*. Note that *Symbol* must represent a valid address. Also, the DSP COFF file must be linked with the usual page number convention:
  - Program space = page number 0
  - Data space = page number 1
  - IO space = page number 2

  All other page numbers are accessed as data space.
- If *Symbol* is unwired, then *DSPAddress* is used as the byte-address for the transfer, and *MemSpace* is used as the memory space. Note that *DSPAddress* is required to be even, to point to a valid 16-bit word (see section about Address Specification).
- The value of *Offset* is added to *DSPAddress*. This functionality is useful to access individual members of structures or arrays on the DSP. Note that the value of *Offset* is always counted in bytes (just as *DSPAddress* it is a byte-address offset, required to be even). This is required to access an individual member of an heterogeneous structure.

Note: The kernel must be loaded and executing for this function to work.

Note: Data is transferred between the PC and DSP in atomic blocks of 32 (Full-Speed USB connection) or 256 (High-Speed USB connection) words. To achieve this, the kernel function that performs the block transfer disables interrupts during the transfer. The time it takes to transfer 32 or 256 words of data to the DSP's on-chip RAM is usually very short. However, it may take much longer (up to 240ns/word) if the transfer is performed to or from SDRAM. In this case, the transfer time may be so long that it interferes with the service of other interrupts in the system. This can be a factor especially when the USB connection is high-speed, because in this case the block size is large (256 words). If this is the case, and transfer atomicity is not required, then a custom DSP function should be used to handle the transfer, instead of than the standard kernel function that is instantiated by SR2_DLL_Bulk_Move_Offset_I16. Such a custom transfer function can be called by the SR2_DLL_User_Move_Offset_I16. Another way to handle this case would be to split the transfer at high-level so that only small blocks are transferred at a time.

Note: Data is transferred between the PC and DSP in atomic blocks of 32 (Full-Speed USB connection) or 256 (High-Speed USB connection) words. However, this is only the case if the transfer does not cross a 64k words boundary. If the transfer does cross a boundary, the transfer is split at high-level so that both halves of the transfer occur in the same 64k page. In this condition the transfer loses its "atomicity".

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board
- **ReadWrite**: Indicates the direction of transfer (1->read, 0->write).
- **Symbol**: Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* and *MemSpace* are used instead.
- **DSPAddress**: Physical base DSP address for the exchange. *DSPAddress* is only used if *Symbol* is empty.
- **MemSpace**: Memory space for the exchange (0-> program, 1->data or 2->IO). *MemSpace* is only used if *Symbol* is empty.
- **Offset**: Represents the offset of the data to be accessed, from the base address indicated by *Symbol* or *DSPAddress*. The actual access address is calculated as the sum of the base address and the offset. *Offset* is useful to access individual members of a structure, or an array.
- **Size**: Represents the size of the allocated *Data* array, as well as the number of elements to read or write.

Outputs:

- **ErrorCode**: This is the error code returned by the kernel function that is executed. Kernel reads always return a completion code of 1; kernel writes always return a completion code of 2. This error code is different from the *USB_Error_Code* that is returned by the function.

In/Outs:

- **Data**: Data words to be read from or written to DSP memory. For a read, the caller must allocate the *Data* array prior to the call.

SR2_DLL_User_Move_Offset_I16

```
long __cdecl SR2_DLL_User_Move_Offset_I16(long BoardRef,
            unsigned short ReadWrite, char Symbol[],
            unsigned long DSPAddress, unsigned long Offset,
            char BranchLabel[], unsigned long BranchAddress,
            short Data[], long Size, unsigned short *ErrorCode);
```

Description:

This function is similar to SR2_DLL_Bulk_Move_Offset, except that it allows a user-defined DSP function to replace the intrinsic kernel function that SR2_DLL_Bulk_Move_Offset uses.
The operation of the USB controller and the kernel allows a user-defined DSP function to override the intrinsic kernel functions (see kernel documentation below). For this, the user-defined DSP function must perform the same actions with the mailbox as the intrinsic kernel function would (kernel read or kernel write). This may be useful to define new transfer functions with application-specific functionality. For example, a function to read or write a FIFO could be defined this way. In addition to the data transfer functionality, a FIFO read or write function would also include the required pointer management that is not present in intrinsic kernel functions.

Accordingly, SR2_DLL_User_Move_Offset includes two controls to define the entry point of the function that should be used instead of the intrinsic kernel function.
When the user-defined function is called, the mailbox contains:

- The BranchAddress field is set to the entry point of the function. This field is not normally used directly by the DSP function. It is set to its entry point, leading to its execution.
- The TransferAddress field. It is set to the address corresponding to *Symbol*, if *Symbol* is used, or to the user-defined 32-bit number *DSPAddress* if *Symbol* is empty. Note that this field is not required to contain a valid address. For instance in the case of a FIFO management function, it could be a FIFO number instead.
- The NbWords field. This 16-bit number represents the number of words to transfer. It is always between 1 and 32 (Full-Speed USB connection), or between 1 and 256 (High-Speed USB connection). This represents the size of the data field of the mail-box.
- The ErrorCode field. It is set to 1 for reads, and to 2 for writes.
- In the case of a write transfer, from 1 to 32 words, as indicated by NbWords, (1 to 256 words for a High-Speed USB connection) have been written to the data field of the mailbox by the USB controller prior to the DSP function call.

The user-defined function should perform its function. If it is a read, it should read the required number of words from the data field of the mailbox. Then it should update the mailbox with the following:

- If it is a write it should provide the required number of words to the Data field of the mailbox.
- Then it may update the ErrorCode field of the mailbox with a completion code that is appropriate for the situation (this is the error code that is returned as the ErrorCode indicator of the VI).

After this, the user-defined function should simply send an acknowledge.

A transfer of a number of words greater than 32 (greater than 256 for a High-Speed USB connection) is segmented by the interface into as many 32-word (256-word) transfers as required. The user-defined function is called at every new segment. If the total number of words to transfer is not a multiple of 32 (256), the last segment contains the remainder.

The TransferAddress field of the mailbox is only initialized at the first segment. The user-defined function may choose to increment it to point to successive addresses (this is what the intrinsic kernel functions do), or may choose to leave it untouched (this would be appropriate if the field contains a FIFO number for instance). The way this field is managed is completely application-specific.

Note: If TransferAddress is used to transport information other than a real transfer address, the following restrictions apply:

- *The total size of the transfer must be smaller or equal to 32768 words. This is because transfers are segmented into 32768-word transfers at a higher level. The information in TransferAddress is only preserved during the first of these higher-level segments. At the next one, TransferAddress is updated as if it were an address to point to the next block.*
- *The transfer must not cross a 64 kWord boundary. Transfers that cross a 64 kWord boundary are split into two consecutive transfers. The information in TransferAddress is only preserved*

during the first of these higher-level segments. At the next one, TransferAddress is updated as if it were an address to point to the next block
- *TransferAddress must be even. It is considered to be a byte-address, consequently its bit 0 is masked at high-level.*

The NbWords field is initialized with the size of the segment transferred, at each segment.
The ErrorCode field is is only initialized at the first segment with the value 1 (read) or 2 (write).
The value that is returned to the ErrorCode indicator of the VI is the value that may be updated by the user-defined function before acknowledging the LAST segment. If the user-defined function does not update the Error Code, the same value (1 for reads, and 2 for writes) is returned back to the PC.

Note: The kernel AND the DSP code of the user-defined function must be loaded and executing for this Vi to be functional.

Note: The value of the R/~W indicator is reflected by the contents of the ErrorCode field of the mailbox at the entry of the user-defined function. For reads, the value is 1, for writes the value is 2.

Inputs:
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board
- **ReadWrite**: Indicates the direction of transfer (1->read, 0->write).
- **Symbol**: Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* is used instead.
- **DSPAddress**: Physical base DSP address for the exchange. *DSPAddress* is only used if *Symbol* is empty.
- **Offset**: Represents the offset of the data to be accessed, from the base address indicated by *Symbol* or *DSPAddress*. The actual access address is calculated as the sum of the base address and the offset. *Offset* is useful to access individual members of a structure, or an array.
- **BranchLabel**: Character string corresponding to the label of the user-defined function. If *BranchLabel* is empty, *BranchAddress* is used instead.
- **BranchAddress**: Physical base DSP address for the user-defined function.
- **Size**: Represents the size of the allocated *Data* array, as well as the number of elements to read or write.

Outputs:
- **ErrorCode**: This is the error code returned by the user function that is executed. This error code is different from the *USB_Error_Code* that is returned by the function.

In/Outs:
- **Data**: Data words to be read from or written to DSP memory. For a read, the caller must allocate the *Data* array prior to the call.

SR2_DLL_HPI_Move_Offset_I16

```
long __cdecl SR2_DLL_HPI_Move_Offset_I16(long BoardRef,
            unsigned short ReadWrite, char Symbol[],
            unsigned long DSPAddress, unsigned short MemSpace, unsigned
            long Offset, short Data[], long Size);
```

Description:

This function is similar to SR2_DLL_Bulk_Move_Offset, except that is relies on the hardware of the HPI, rather than the kernel to perform the transfer.
Transfers are limited to the on-chip RAM that is directly accessible via the HPI (byte-addresses $00C0_H$ to $FFFF_H$). The MemSpace control is not used. This VI will perform transfers into and out of the data and program space. This VI will transfer to any address accessible via the HPI, regardless of memory space. The on-chip I/O space is not accessible. If an attempt is made to write data outside the allowed range, the data is not written. If an attempt is made to read data from outside the allowed range erroneous data is returned.

Note: The kernel does not need to be loaded or functional for this function to execute properly. This function will complete the transfer even if the DSP has crashed, making it a good debugging tool.
Transfers with the HPI use the control pipe 0 instead of the fast bulk pipes used by SR2_DLL_Bulk_Move_Offset. The bandwidth for such transfers is typically low (500kb/s for USB 1.1). However it is guaranteed.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board
- **ReadWrite**: Indicates the direction of transfer (1->read, 0->write).
- **Symbol**: Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* and *MemSpace* are used instead.
- **DSPAddress**: Physical base DSP address for the exchange. *DSPAddress* is only used if *Symbol* is empty.
- **MemSpace**: Memory space for the exchange (0-> program, 1->data or 2->IO). *MemSpace* is only used if *Symbol* is empty.
- **Offset**: Represents the offset of the data to be accessed, from the base address indicated by *Symbol* or *DSPAddress*. The actual access address is calculated as the sum of the base address and the offset. *Offset* is useful to access individual members of a structure, or an array.
- **Size**: Represents the size of the allocated *Data* array, as well as the number of elements to read or write.

Outputs:
N/A

In/Outs:

- **Data**: Data words to be read from or written to DSP memory. For a read, the caller must allocate the *Data* array prior to the call.

SR2_DLL_Resolve_UserSymbol

```
long __cdecl SR2_DLL_Resolve_UserSymbol(long BoardRef, char Symbol[],
                 unsigned long *Value, unsigned short *MemSpace);
```

Description:
This function may be used to provide the address corresponding to a particular symbol in the symbol table of the presently loaded DSP code. Used in conjunction with the data transfer functions (SR2_DLL_Bulk_Move_Offset, SR2_DLL_User_Move_Offset and SR2_DLL_HPI_Move_Offset) it allows more flexibility for choosing the actual transfer address. For instance the address may be transformed in an application-specific manner prior to the transfer.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board
- **Symbol**: Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* and *MemSpace* are used instead.
- **MemSpace**: Memory space for the exchange (0-> program, 1->data or 2->IO). *MemSpace* is only used if *Symbol* is empty.

Outputs:
- **Value**: This is the resolution value of the symbol.
- **MemSpace**: This is the memory space for the symbol. Note that this value actually reflects the "page number" that was used at link time for this symbol. For this number to indicate a valid memory space, the symbol must be linked using the usual convention:
  - 0 -> Program Space
  - 1 -> Data Space
  - 2 -> IO Space

SR2_DLL_InitFlash

```
long __cdecl SR2_DLL_InitFlash(long BoardRef, double *FlashSize);
```

Description:
This function downloads and runs the Flash support DSP code. The DSP is reset as part of the download process. All DSP code is aborted. The Flash support code must be running in addition to the kernel to support Flash programming functions. The function also detects the Flash, and if it finds one it returns its size in kWords. If no Flash is detected, the size indicator is set to 0.

Inputs:
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board

Outputs:
- **FlashSize**: This indicator returns the size of the Flash detected in kWords. If no Flash is detected, it returns zero.

SR2_DLL_EraseFlash

```
long __cdecl SR2_DLL_EraseFlash(long BoardRef,
      unsigned long StartingAddress, unsigned long Size);
```

Description:
This function erases the required number of 16-bit words from the Flash, starting at the selected address. The erasure proceeds in sectors therefore more words may be erased that are actually selected. For instance, if the starting address is not the first word of a sector, words before the starting address will be erased, up to the beginning of the sector. Similarly, if the last word selected for erasure is not the last word of a sector, additional words will be erased, up to the end of the last selected sector. The erasure is such that the selected words, including the starting address, are always erased.

Note: The sector size is 32 kwords.

Note: Erasure should only be attempted in the sections of the memory map that contain Flash. Erasure attempts outside the Flash will fail.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.
- **Starting Address**: Address of the first word to be erased.
- **Size**: Number of words to be erased.

Outputs:
- **N/A**

SR2_DLL_FlashMove_I16

```
long __cdecl SR2_DLL_FlashMove_I16(long BoardRef,
      unsigned short ReadWrite, unsigned long DSPAddress,
      short DataIn[], long Size);
```

Description:
This function reads or writes an unlimited number of data words to/from the Flash memory. Note that if only Flash memory reads are required the function SR2_DLL_Bulk_Move_Offset should be used instead, since it does not require the presence of the Flash support code.

An attempt to write outside of the Flash memory will result in failure.

The writing process can change ones into zeros, but not change zeros back into ones. If a write operation is attempted that should result in a zero turning back into a one, then it results in failure. Normally an erasure should be performed prior to the write, so that all the bits of the selected write zone are turned back into ones.

Note: Contrary to previous generations of Flash devices that have been used in Signal Ranger boards, the Flash device used in Signal_Ranger_mk2 cannot be incrementally programmed. This means that a word location that has been previously programmed MUST be erased before reprogramming. This is true even if the reprogramming operation is only intended to turn some of the remaining "1s" into "0s".

Inputs:
- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board
- **ReadWrite**: Indicates the direction of transfer (1->read, 0->write).
- **DSPAddress**: Physical base DSP address for the exchange.
- **Size**: Represents the size of the allocated *Data* array, as well as the number of elements to read or write.

Outputs:
- **N/A**

In/Outs:
- **Data**: Data words to be read from or written to Flash memory. For a read, the caller must allocate the *Data* array prior to the call.

SR2_DLL_FPGA_LoadConfiguration

```
long __cdecl SR2_DLL_FPGA_LoadConfiguration(long BoardRef,
      char file_path[], char Tools_Version[], long lenTools,
      char Design_Name[], long lenDesign, char Architecture[],
      long lenArch, char Build_Date[], long lenBuild, char Device[],
      long lenDev);
```

Description:
This function downloads an "*.rbt" logic configuration file into the FPGA. The DSP is reset prior to the download. All DSP code is aborted. The .rbt file must be valid, and must be correct for the specified FPGA. Loading an invalid rbt file into an FPGA may damage the part.

Inputs:

- **BoardRef**: This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by SR2_DLL_Open_Next_Avail_Board.
- **file_path**: This is the file path leading to the ".rbt" file describing the FPGA logic. A dialog box is presented if the path is empty.
- **lenTools**: This is size of the string that has been allocated to contain the *Tools_Version* output. A minimum of 60 characters must be allocated.
- **lenDesign**: This is size of the string that has been allocated to contain the *Design_Name* output. A minimum of 60 characters must be allocated.
- **lenArch**: This is size of the string that has been allocated to contain the *Architecture* output. A minimum of 60 characters must be allocated.
- **lenBuild**: This is size of the string that has been allocated to contain the *Build_Date* output. A minimum of 60 characters must be allocated.
- **lenDev**: This is size of the string that has been allocated to contain the *Device* output. A minimum of 60 characters must be allocated.

Outputs:

- **Tools_Version**: This string contains the Tools Version information found in the .rbt file. A string of at least 60 Characters must be allocated by the caller prior to the call.
- **lenDesign**: This string contains the Design Name information found in the .rbt file. A string of at least 60 Characters must be allocated by the caller prior to the call.
- **lenArch**: This string contains the Architecture information found in the .rbt file. A string of at least 60 Characters must be allocated by the caller prior to the call.
- **lenBuild**: This string contains the Build Date information found in the .rbt file. A string of at least 60 Characters must be allocated by the caller prior to the call.
- **lenDev**: This string contains the Device information found in the .rbt file. A string of at least 60 Characters must be allocated by the caller prior to the call.

Error Codes

The following table lists the error codes that may be returned by the various functions of SRm2_HL.dll.

Note: The list is not complete. Only the most usual codes are listed.

Error Nb	Cause
00000002h	Memory is full
00000003h	Wrong memory zone accessed
00000004h	End-of-file encountered
00000005h	File already open
00000006h	File I/O error
00000007h	File not found
00000008h	File permission error
00000009h	Disk full
0000000Ah	Duplicate path
0000000Bh	Too many files open
BFFC0804h	No Board Detected
BFFC0805h	Open board generate error
BFFC0806h	Board info generate error
BFFC0807h	Close board generate error
BFFC0808h	DSP File not accessible
BFFC0809h	Reset DSP generate error

BFFC0812h	This DSP kernel is not for c5000
BFFC080Ah	Unreset_DSP genrate error
BFFC080Bh	DSP file not found
BFFC080Ch	DSP file not valid
BFFC080Dh	DSP file type not supported
BFFC080Eh	DSP kernel does not compare
BFFC080Fh	Write_hpi generate error
BFFC0810h	Read_hpi generate error
BFFC0811h	This DSP file is not for c5000
BFFC0813h	Write and read test error
BFFC0814h	DSP did not return… may have crashed
BFFC0815h	Symbol not found in the table
BFFC0816h	DSPInt generate error
BFFC0817h	DSP Timeout
BFFC0818h	Kernel not loaded
BFFC0820h	Close driver error
3FFC0104h	Error query not supported
BFFC0821h	Open driver error
BFFC0822h	Data move with kernel generates error
BFFC0823h	Exec low level generates error
BFFC0824h	Get pipe info generates error
BFFC0825h	Reset DSP generates error
BFFC0826h	HPI move generates error
BFFC0827h	DSP int generates error
BFFC0828h	Get configuration descriptor error
BFFC0829h	Write Led generates error
BFFC082Ah	Get device descriptor error
BFFC082Bh	Read and write HPI control error
BFFC082Ch	Change USB Timeout mode error
BFFC082Dh	DSP code section does not check
BFFC082Eh	FPGA file not valid
BFFC082Fh	FPGA configuration aborted before the end
BFFC0830h	Driver access error
BFFC0831h	DSP family not recognized or board not open
BFFC0832h	FPGA file not for target device family
BFFC0833h	Section too large to fit in Flash
BFFC0834h	DSP File may not be linked
BFFC0835h	Section not aligned on an even byte address
BFFC0836h	Erase Flash Error
BFFC0837h	Write Flash Error

Example Code

An example is provided, including the PC side and the DSP side. This example may be found in the following directory:
C:\Program Files\ SignalRanger_mk2\Examples\C_Examples\HLDLLCallExample.zip.
Simply unzip *HLDLLCallExample.zip* to deflate the example.
The example contains the following:

- A directory named *UserFunctionDSPCode*, which contains the DSP project and source code.
- A directory named *TestHLDLL*, which contains the Visual C++ .net project and source code.

Testing The Example Code

To test the example code, simply power-up the board and wait until its LED has turned green. Then run the *TestHLDLL.exe* application found in the *HLDLLCallExample\TestHLDLL\Debug* directory. The following figure shows the user-interface of the example application:

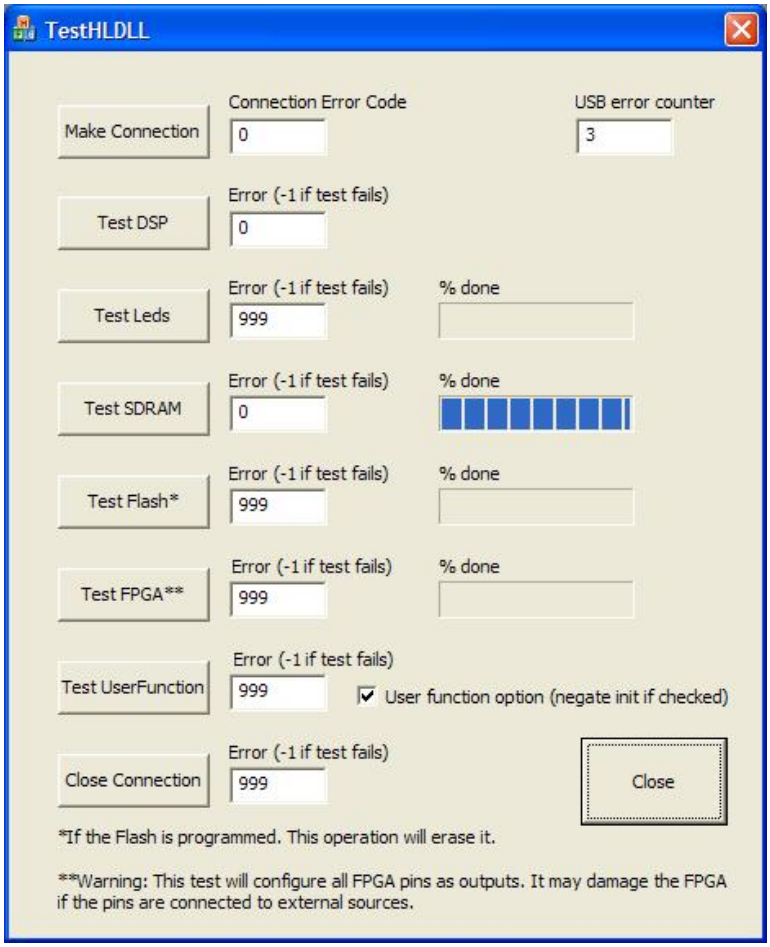


Figure 5 C Example Application

Before attempting to run any of the tests, press the *Make Connection* button, and wait for the application to return a zero error code. This first operation may take a few seconds, because it is at this time that the *SRm2_HL.dll* DLL is loaded.
After a board connection has been successfully established, any test may be performed in any order.
When all the tests have been performed press the *Close Connection* button to close the driver and perform the necessary clean-up operations.
Then Pressing the *Close* button terminates the application.

Compile-Time Required Files

The following line must be present in the main source file:

```
# include SRm2_HL.h
```

In the example it is placed at he beginning of the *TestHLDLLDlg.cpp* source file.
This include file itself includes the following files: *extcode.h, fundtypes.h and platdefines.h,*
All four files must be present in the same directory as *TestHLDLLDlg.cpp* for the compiler to build the object code.
The *SRm2_HL.lib* and *SRm2_HL.dll* files must be present in the project directory.

Project Setup

To statically link with the SRm2_HL.dll do the following:

- Go to the *Add New Item To Project* menu, and choose the *SRm2_HL.lib* file.
- Go to the *Add New Item To Project* menu, and choose the *delayimp.lib* to the project (in Visual C++ ".net" it can be found in *Program Files\Microsoft Visual Studio .NET 2003\Vc7\lib*\).
- Go to the *Project\Properties* menu, choose *Linker Options* and click on *Command Line*. Add the following linker command line:

```
/DELAYLOAD:SRm2_HL.dll
```

This command line delays the load of the DLL at run-time until the first call actually occurs. It is required to the proper operation of the interface. Alternately, the DLL may be linked dynamically (see *Building a Project Using Visual C/C++ ".net").*

Run-Time Required Files

To insure the proper operation of any executable built, the following support files must be present in the same directory as the executable:

- *Sranger2.dll*
- *SRm2_HL.dll*
- *SR2_Flash_Support.out*
- *SR2_FPGA_Support.out*
- *SR2_Kernel_HostDownload.out*
- *SR2_Kernel_PowerUp.out*

In addition, the following user-specific files must also be present:

- *UserFunctionDSPCode.out*. This is the DSP code required by the specific user application (in this case the *TestHLDLL.exe* application).
- *SR2_SelfTest.rbt.* This is the FPGA logic required by the specific user application.

Note: Particular user-specific files may have a different name. These are the files that are loaded by the example code.

Note: A specific FPGA logic may not be required. This example application loads this FPGA logic to be able to test the FPGA.

Other Details

Other details of the implementation can be found in the source code (both DSP-side and PC-side) in the form of comments.

DSP Code Development

When developing DSP code, two situations may arise:

- The DSP code is a complete application that is not intended to return to the previously executing DSP code. This is usually the case when developing a complete DSP application in C. In this case, the *main* of the DSP application is launched by a function called *c_int00* that is created by the compiler. When (if) the *main* returns, it goes back to a never-ending loop within *c_int00*. It never returns to the code that was executing prior to this code.
- The DSP code is a simple function, intended to run, then return to the previously executing DSP code (kernel or user-code). This process may be used to force the DSP to execute short segments of code asynchronously from other code running on the DSP in the background. The *other code* running in the background may either be the kernel or other user code that has been launched previously.

Several examples are provided to gain experience into the programming of the DSP board, and its interface to a PC application. The examples directory contains examples of DSP code written in C, as well as written in assembly. All DSP code developed for the Signal_Ranger_mk2 board must comply with the following requirements.

Code Composer Studio Setup

The setup of Code Composer Studio should use a C55x simulator or emulator. Otherwise the visual linker might not work as expected. If a simulator is used, we recommend using the C5502 “cycle-accurate” simulator. It is the closest to the physical DSP that is used on the board.

Project Requirements

In Code Composer Studio the project should be created for a C55x platform (NOT the C54x platform).

C-Code Requirements

- When developing a function in C that will be launched by the K_Exec kernel process, and is required to return to the previously executing code after its execution, the function must be declared using the *interrupt* qualifier. This directs the compiler to protect the required DSP registers on the stack, and to end the function with a RETI instruction, which properly restores the context from the stack at completion. Furthermore, the function must include the *acknowledge* macro (see code examples), which should be executed within 5s of the function launch.
- When the function is not intended to return to the previously executing code (the *main* function of a C project for instance), it is not required to be declared with the *interrupt* qualifier. However, it still needs to contain the acknowledge within 5s of entry. The acknowledge signals to the PC that the function has been successfully launched (see sections on kernel for more details).

Assembly Requirements

- When developing a function in assembly that will be launched by the K_Exec kernel process, and is required to return to the previously executing code after its execution, the function must protect all the DSP registers that it uses on the stack. The function must end with a RETI instruction, which properly restores the context from the stack at completion. Furthermore, the function must include the *acknowledge* macro (see code examples), which should be executed within 5s of the function launch.
- When the function is not intended to return to the previously executing code, it is not required to protect the registers that it uses. However, it still needs to contain the acknowledge within 5s of entry. The acknowledge signals to the PC that the function has been successfully launched (see sections on kernel for more details).
- When developing a code section in assembly, the *.even* directive must be used at the beginning of the section. This insures that code sections always begin on an even byte-address. This is not required when developing code in C, because the C-Compiler itself uses the *.even* directive in the assembly code it generates. This requirement only applies to code sections.

Build Options

Compiler

- Basic : The *Full Symbolic Debug* option should be used. This option does not yield code that is larger or less efficient in any way. However, it provides the symbolic information in the ".out" file that is necessary to access the labels and variables from their symbolic names. In particular the symbolic information regarding structures is only provided in the ".out" file if this option is used.
- Advanced: The *Processor version (-v)* field must be initialized with *5502*. This way, the code is optimized for the TMS320VC5502 DSP. There are some platforms in the C55x family that do not have the full instruction set of the C5502. In addition, when a specific platform is not indicated, the compiler may need to work around silicon bugs that are not present in the C5502, but may be present in other DSPs in the same family, thereby creating less efficient code.
- For versions of Code Composer Studio 3 and up, the new DWARF format is used to store debug information in the COFF file. In order to have access to structure member information, the "`--symdebug:coff`" compiler option must be used. Simply insert the "`--symdebug:coff`" option at the end of the compiler command line.

Required Modules

Real-Time-Support

For a DSP code written in C, the rts55.lib or rts55x.lib (depending on the memory model) should be added to the project files.

Interrupt Vectors

- If the project uses interrupts, the *vectors.asm* module should be added to the project. The template file that is provided with the examples should be used as an example to generate an interrupt vector table that provides the vectors for the user code. Simply modify the table as required for the vectors that should be implemented.
- Do not modify the values of the following symbols:
  - ISRKernel
  - _RESET_K

- Do not modify the DSPINT interrupt vector. This would crash the DSP kernel that uses the interrupt.
- Do not add an interrupt vector for the last interrupt (TRAP #31). This vector has been intentionally omitted from the table because the kernel modifies it dynamically. Initializing this vector would crash the kernel as soon as the vector table is loaded.

- If no interrupts other than DSPInt and Trap #31 (the interrupts used by the kernel) are used in the project, it would normally not be required to provide a *vectors* section. Indeed, the vectors of the interrupts used by the kernel are already initialized when the user takes control of the DSP. However, it may be preferable to always include a vectors section for the reason explained below.

Note that for projects written in C, a default *vectors* section is automatically included in the *rts55.lib* or *rts55x.lib* if the project does not already define one. This default vectors section is not compatible with the kernel and should never be used. The easiest way to avoid this default vectors section is to include a user-defined *vectors* section that contains at least the proper vectors for the operation of the kernel. Simply include the template file that is provided with the

software installation in the project. This problem only occurs in projects that include the *rts55.lib* or *rts55x.lib*. This is generally only the case for projects written in C/C++.

Link Requirements

Memory Description File

To link the code using the visual linker, use the *Signal_Ranger_mk2.mem* description file. This file describes the memory map of the Signal Ranger_mk2 board. It includes the address ranges that are reserved and should not be modified.

Vectors Section

The *vectors* section must be placed in the *Vectors* memory range, between byte addresses 100_H and $1FF_H$. Notice that the *vectors* section is a little bit smaller than the *Vectors* memory range. This is because the vector for the TRAP #31 that is used by the kernel has been intentionally omitted from the section (see above). Be sure to not load anything in this little space at the end of the section as this would interfere with the operation of the kernel and most likely would crash the kernel during the load of the DSP code.

Unused_ISR Section

The small *Unused_ISR* section provides a simple RETI for all the interrupts that are not used. This way, if such an ISR is triggered, it harmlessly returns to the user code immediately upon being triggered. This small section can be loaded anywhere with the rest of the code. It must never be loaded at the end of the *vectors* section.

Global Symbols

Only the global symbols found in the DSP ".out" file are retained in the symbol table. This means that to allow symbolic access to the software interface, variables declared in C should be declared global (outside all blocks and functions, and without the *static* keyword). Variables and labels declared in assembly (function entry points for instance) should be declared with the assembly directive *.global*.

Preparing Code For "Self-Boot"

The on-board Flash circuit may be programmed to load DSP code and/or FPGA logic at power-up, and to launch the execution of the specified DSP code. More information about DSP and FPGA Flash boot tables is located in the following sections.
Once DSP code and/or FPGA logic have been developed and tested under control of the PC (possibly using the mini-debugger), they may be programmed into Flash using the appropriate functions of the mini-debugger.

The DSP code loaded in Flash must contain an acknowledge (see examples) at its beginning, within 5s of execution, even when the code is launched from Flash. Usually, when developing in C, this acknowledge is placed in the first lines of the *main* function.
Failure to include the acknowledge does not cause the DSP code to crash. However, it does cause the USB controller to fail to recognize that the Power-Up kernel has been loaded. In this circumstance it is necessary to reset the board in order to gain access from the PC. This reset in turn would cause the boot-loaded DSP code to abort.

Note: The Acknowledge is also required for code that is written to be loaded directly from the PC and executed using the SR2_K_Exec.vi interface VI, or using the ***Exec*** *button of the mini-debugger. Therefore DSP code that has been developed and tested using the mini-debugger, or*

code that is usually downloaded and executed from the PC should normally be ready to be programmed into the boot-table "as is".

More information about the acknowledge command may be found in the following sections about the kernels.

To allow the DSP code programmed into Flash to boot properly, its entry point must be properly defined in the link. This is not an absolute requirement for code that is loaded from the PC. Indeed, Code loaded from the PC may be launched from any existing label, or even an arbitrary address, using either the mini-debugger or the LabVIEW or C/C++ interfaces. However, the boot-loader that executes from the kernel only launches code from the defined entry point. If none is defined, the DSP code will most likely crash at power-up.

Mini-Debugger

The mini-debugger allows the developer to interactively:

- Reset the Signal Ranger mk2 board.
- Download a DSP executable file to DSP memory, or use the symbols of a DSP code already in memory.
- Launch the execution of code from a specified address or from a symbolic label.
- Read and write CPU registers.
- Read and write DSP memory with or without symbolic access.
- Clear, program and verify the Flash memory.
- Interactively download an FPGA logic file into the FPGA

The mini-debugger can be used to explore the DSP's features, or to test DSP code during development.
The mini-debugger is simply a user-interface shell that leverages the capabilities of the PC interface libraries to allow the developer to observe and modify DSP memory in real time, while the DSP code is running. Since these are the very same libraries that are provided to develop PC applications that use the board, the transition between debugging and field deployment is completely seamless.

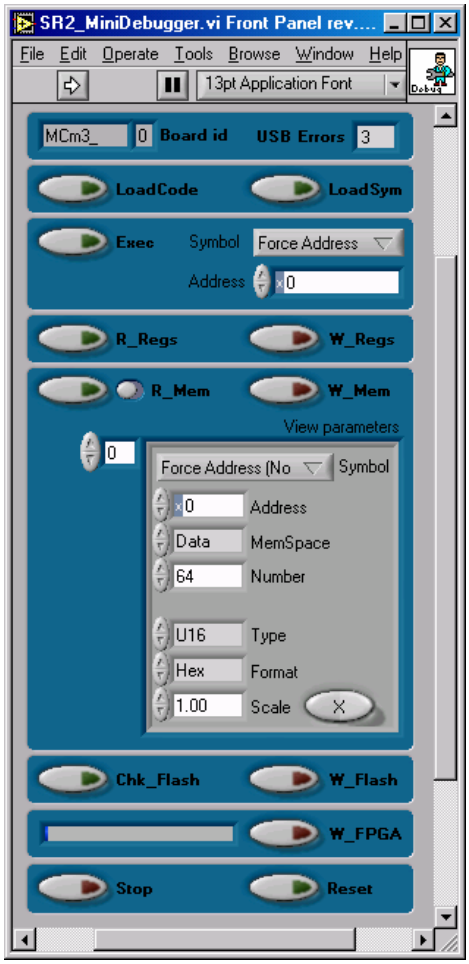


Figure 6 User Interface of the Mini-Debugger

Description Of The User Interface

- **Board id** This is a text field where the prefix of the board driver ID should be typed. The mini-debugger actually supports several related DSP boards. This field is used to indicate the type of board that the mini-debugger should take control of. The small field to the right of Board ID indicates the driver instance number for the board that has been opened by the mini-debugger. The mini-debugger always opens the first board of the specified type that is not already open. For Signal Ranger mk2 boards, the field should be initialized with *SRm2_*.
- **USB Errors** This indicator shows the number of USB errors in real time. It may be used to monitor the "health" of the USB connection. This indicator is a 4 bit wrap-around counter that is located within the hardware of the on-board USB controller. Note that many USB errors can be detected, which will not affect the transmission at high level. This is due to the intrinsic robustness of the USB protocol to errors.
- **LoadCode** Loads a DSP COFF file. The DSP is reset prior to the code download. The application presents a file browser to allow the user to select the file. The file must be a legitimate COFF (".out") file for the target DSP. After the COFF file has been successfully loaded into DSP memory, the corresponding symbol table is loaded into the PC memory to allow symbolic access to variables and labels. The code is not executed.
- **LoadSym** Loads the symbol table corresponding to a specified DSP code into the PC memory to allow symbolic access to variables and labels. Nothing is loaded into DSP memory. This is useful to gain symbolic access to a DSP code that may already be running,

such as code loaded from Flash by the bootload process. The application presents a file browser to allow the user to select the file. The file must be a legitimate COFF file for the target DSP.

- **Exec** Forces execution to branch to the specified label or address. The DSP code that is activated this way should contain an *Acknowledge* in the form of a *Host Interrupt Request (HINT)*. Otherwise the USB controller will time-out, and an error will be detected by the PC after 5s. The simplest way to do this is to include the *acknowledge* macro at the beginning of the selected code. This macro is described in the two demo applications.
  Symbol or ***Address*** is used to specify the entry point of the DSP code to execute. ***Symbol*** can be used if a COFF file containing the symbol has been loaded previously. Otherwise, ***Address*** allows the specification of an absolute branch address. ***Address*** is used only if ***Symbol*** is set to the "Force Address (No Symbol)" position.
  When a new COFF file is loaded, the mini-debugger tries to find the ***_c_int00*** symbol in the symbol table. If it is found, and its value is valid (different from 0) ***Symbol*** points to its address. If it is not found, ***Symbol*** is set to the "Force Address (No Symbol)" position.
- **R_Regs** Reads the CPU registers mapped in RAM at address 0000_H, and presents the data in an easy to read format.

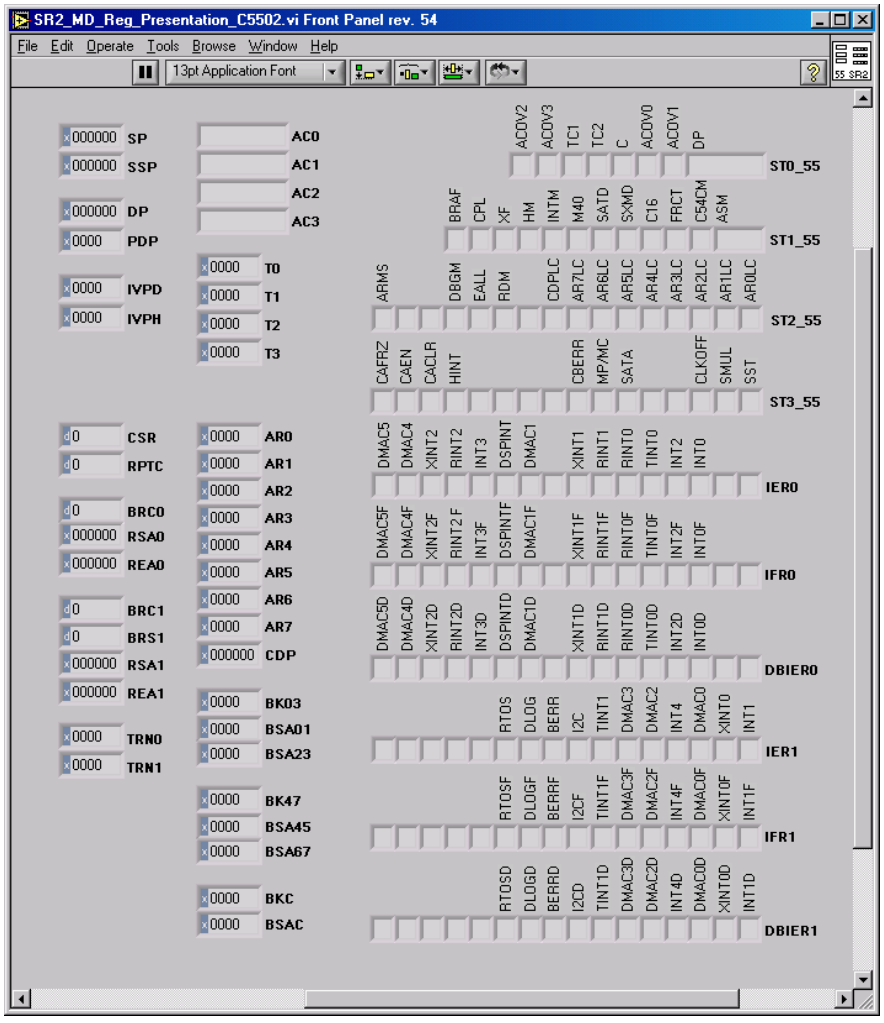


Figure 7 Register presentation panel

- **W_regs** Writes most of the CPU registers mapped in RAM at address 0000_H. Some fields are greyed out and cannot be written, either because the kernel uses them and would restore them after modification, or because their modification would compromise its operation.

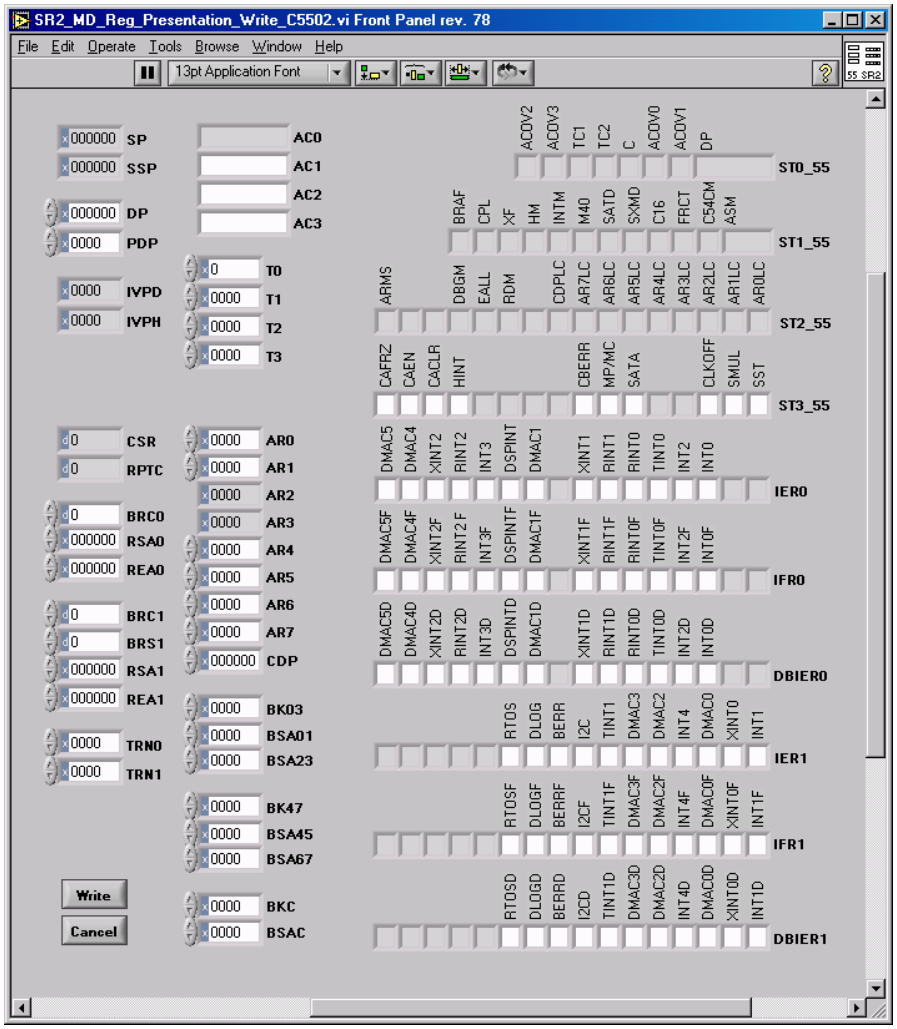


Figure 8 Register write presentation panel

- **R_Mem** Reads DSP memory and presents the data to the user.
  The small slide button beside the button allows a continuous read. To stop the continuous read, simply replace the slide to its default position.
  The *View parameter* array is used to select one or several memory blocks to display. Each index of the array selects a different memory block.
  To add a new memory block, simply advance the index to the next value, and adjust the parameters for the block to display. To completely empty the array, right-click on the index and choose the “Empty Array” menu. To insert or remove a block in the array, advance the index to the correct position, right-click on the *Symbol* field, and choose the “Insert Item Before” or “Delete Item” menu.

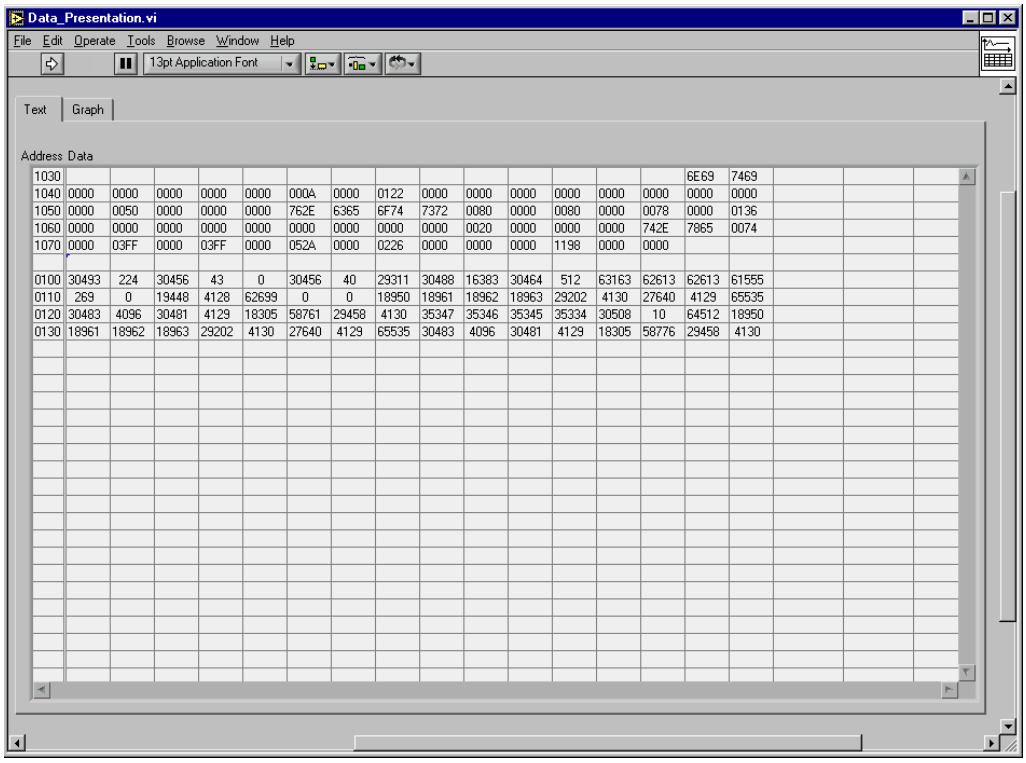
  For each block:
  - **Symbol** or **Address** is used to specify the beginning of the memory block to display. **Symbol** can be used if a COFF file containing the symbol has been loaded previously. If **Symbol** is set to a position other than “Force Address (No Symbol)”, **Address** and **MemSpace**

are forced to the value specified in the COFF file for this symbol. The list of symbols is cleared when a new COFF file is loaded, or when the Mini-Debugger is stopped and run again. It is not cleared when the DSP board is reset.
By right-clicking on the Symbol field, it is possible to remove or insert an individual element.

*Note for **MemSpace** to specify the correct space, the DSP code must be linked using the usual page-number convention:*
0 -> Program Space
1 -> Data Space
2 -> IO space

Note: Specified addresses are byte-addresses

- ***MemSpace*** indicates the memory space used for the access. The position "???" (Unknown) defaults to an access in the Data space. If ***Symbol*** is set to a specific symbol, ***MemSpace*** is forced to the value specified in the COFF file for this symbol.
- ***Number*** specifies the number of elements to display.
- ***Type*** specifies the data type to display. Three basic widths can be used: 8 bits, 16 bits, and 32 bits. All widths can be interpreted as signed (*I8, I16, I32*), unsigned (*U8, U16, U32*), or floating-point data. The native DSP representation is 16 bits wide. When presenting 8-bit data, the bytes represent the high and low parts of 16-bit memory registers. They are presented MSB first and LSB next. When presenting 32-bit data (I32, U32 or Float), the beginning address is NOT automatically aligned to the next even address. The first address is taken as the upper 16 bits and the next address is taken as the lower 16 bits. This follows the standard bit ordering for 32-bit data. It is the responsibility of the developer to make sure that the alignment is correct; otherwise the data does not make sense.
- ***Format*** specifies the data presentation format (Hexadecimal, Decimal or Binary).
- ***Scale*** specifies a scaling factor for the graph representation.
- ***X or 1/X*** specifies if the data is to be multiplied or divided by the scaling factor.



Address	Data															
1030															6E69	7469
1040	0000	0000	0000	0000	0000	000A	0000	0122	0000	0000	0000	0000	0000	0000	0000	0000
1050	0000	0050	0000	0000	0000	762E	6365	6F74	7372	0080	0000	0080	0000	0078	0000	0136
1060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0020	0000	0000	0000	742E	7865	0074
1070	0000	03FF	0000	03FF	0000	052A	0000	0226	0000	0000	0000	1198	0000	0000		
0100	30493	224	30456	43	0	30456	40	29311	30488	16383	30464	512	63163	62613	62613	61555
0110	269	0	19448	4128	62699	0	0	18950	18961	18962	18963	29202	4130	27640	4129	65535
0120	30483	4096	30481	4129	18305	58761	29458	4130	35347	35346	35345	35334	30508	10	64512	18950
0130	18961	18962	18963	29202	4130	27640	4129	65535	30483	4096	30481	4129	18305	58776	29458	4130

Figure 9 Data presentation (Text mode)

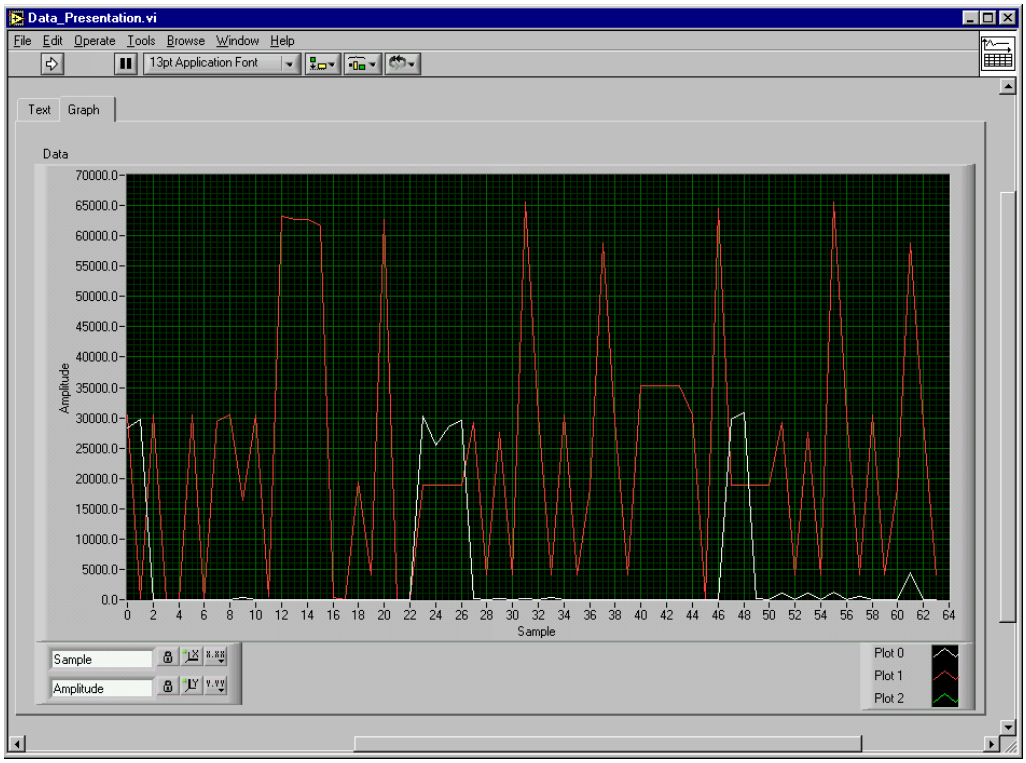


Figure 10 Data presentation (Graph mode)

The user can choose between *Text* mode (figure 9), and *Graph* mode (figure 10) for the presentation of memory data. In *Text* mode, each requested memory block is presented in sequence. The addresses are indicated in the first column. In *Graph* mode, each memory block is scaled, and represented by a separate line of a graph.

- **W_Mem** Allows the memory contents to be read and modified. The function first reads the memory, using the *View_parameters*, and presents a Text panel similar to the one presented for the *R_mem* function. The user can then modify any value in the panel, and press the *Write* button to write the data back to memory.
  Several points should be observed:
  - Even though data entry is permetted in any of the cells of the panel, only those cells that were displayed during the read phase (those that are not empty) are considered during the write.
  - When an attempt is made to write an odd number of bytes, an additional byte is appended to the byte array. This byte is set to FF_H. This is necessary because all native transfers are performed 16-bits at a time.
  - Data must be entered using the same type and format as were used during the read phase.
  - During the write phase ALL the data presented in the panel is written back to DSP memory, not just the data that has been modified by the user. Normally this is the same data, however this may have an importance if the data changes in real time on the DSP, because it may have changed between the read and the write.

Note: When presenting, or writing 32 bit data words (I32, U32 or Float), the PC performs 2 separate accesses (at 2 successive memory addresses) for every transferred 32-bit word. In principle, the potential exists for the DSP or the PC to access one word in the middle of the exchange, thereby corrupting the data.

For instance, during a read, the PC could upload a floating-point value just after the DSP has updated one 16-bit word constituting the float, but before it has updated the other one. Obviously the value read by the PC would be completely erroneous.
Symmetrically, during a write, the PC could modify both 16-bit words constituting a float in DSP memory, just after the DSP has read the first one, but before it has read the second one. In this situation The DSP is working with an "old" version of half the float, and a new version of the other half.
These problems can be avoided if the following precautions are observed:
When the PC accesses a group of values, it does so in blocks of up to 256 16-bit words at a time (32 words if the board is connected in full-speed). Each of these 256-word block accesses is atomic (the DSP cannot do any operation in the middle of the PC access). Therefore the DSP cannot "interfere" in the middle of any single 32-bit word access.
This alone does not guarantee the integrity of the transferred values, because the PC can still transfer a complete block of data in the middle of a DSP operation on this data. To avoid this situation, it is sufficient to also render the DSP operation on any 32-bit data atomic (by disabling interrupts for the length of the operation), then the accesses are atomic on both sides, and data can safely be transferred 32 bits at a time.

- **W_Flash** Allows the developer to load a DSP code and/or FPGA logic boot table into Flash memory, or to clear the memory. A boot table directs the Power-Up kernel to load the DSP and/or FPGA with the specified code and/or logic at start-up. Boot tables are described in other sections of this document.
  The *W_Flash* button brings a browser that allows the developer to choose the DSP code (.out) and FPGA logic (.rbt) files. Not choosing a file does not preserve the Flash contents. If no file is chosen for DSP code or FPGA logic, the corresponding boot table is erased from Flash.
  The operation systematically resets the DSP and loads the Flash support code that is required for Flash programming operations.
  DSP file Indicates the path chosen for the DSP code.
  Nb Sections Indicates the number of sections of the DSP code to be loaded into Flash ROM. Empty sections in the .out executable file are eliminated.
  Entry Point Specifies the entry point of the code, as it is defined in the COFF file.
  DSP_Load_Address Indicates the load address of the boot table in Flash memory.
  DSP_Last_Address Indicates the last address of the boot table in Flash memory.
  FPGA file Indicates the path chosen for the FPGA logic file.
  Tools version The version of the ISE tools that were used to generate the .rbt file
  Design name The name of the design as it appears in the .rbt file
  Architecture The type of FPGA for which the rbt file is built.
  Device The model number of the FPGA for which the .rbt file is built.
  Date The build date of the .rbt file.
  FPGA_Load_Address This is the address of the beginning of the FPGA boot table in Flash memory.
  FPGA_Last_Address This is the last address of the FPGA boot table in Flash memory. It is normally $1FFFF_H$.
  Write Press this button to execute the Flash programming or erasure. The required sectors of Flash are erased prior to programming. Because the Flash is erased sector by sector, rather than word by word, the erasure will usually erase more words than what is strictly necessary to contain the DSP code or FPGA logic.
  Cancel Press this button to cancel all operations and return to the mini-debugger.
  Flash Size If the Flash is detected, this field indicates its size in kwords.

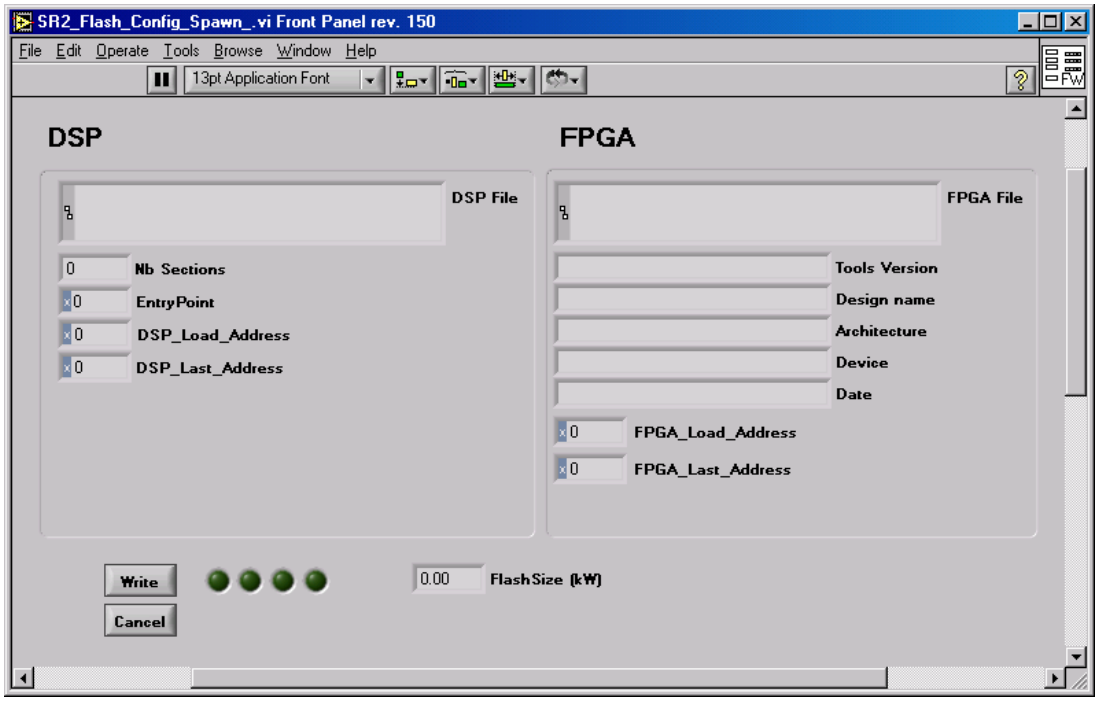


Figure 11

- **Chk_Flash** This button brings a panel that is very similar to the *W_Flash* button. The only difference is that this utility verifies the contents of the Flash, rather than program it. If no file is selected for the DSP and/or FPGA, the corresponding verification is cancelled and always indicates a positive result.

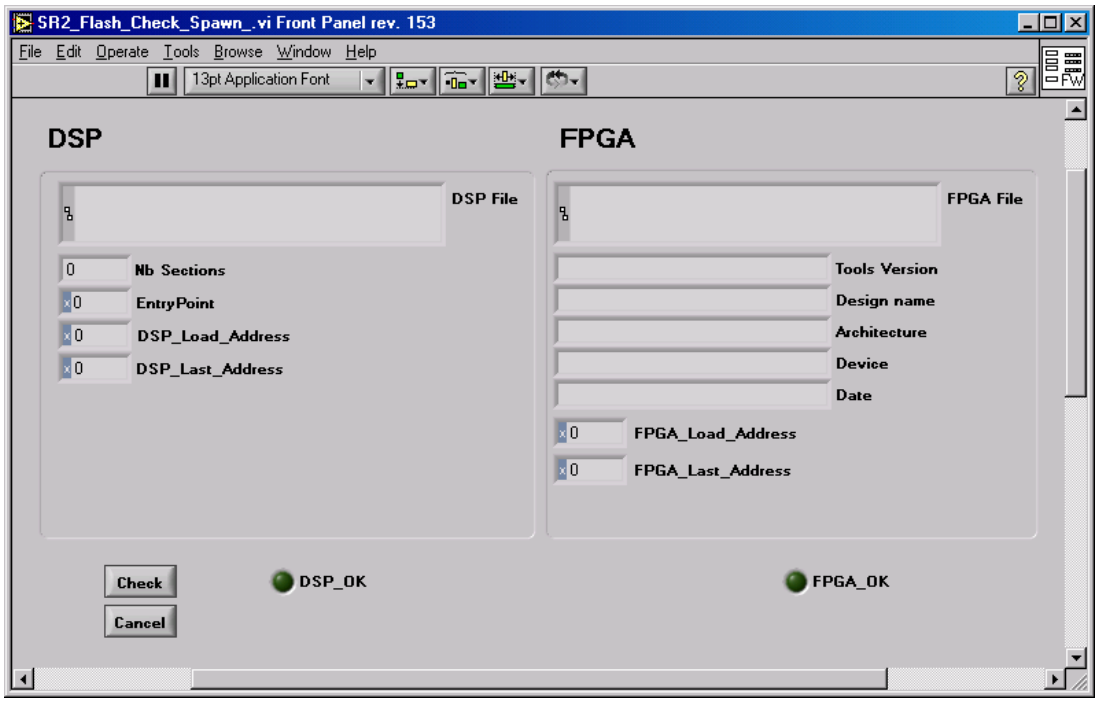


Figure 12

- **W_FPGA** Allows the interactive downloading of an FPGA logic (.rbt) file to the FPGA.
  A browser is presented when the button is pressed to allow the developer to choose the .rbt file.
  The operation systematically resets the DSP and loads the FPGA support code that is required for FPGA load operations.
- **Stop** Stops the execution of the mini debugger.
- **Reset** Forces a reset of the board and reloads the Host-Download kernel. All previously executing DSP code is aborted. This may be necessary to take control of a DSP that has crashed. The DSP is not reset by default when the mini-debugger takes control of the board. This allows code that may have been loaded and run by the Power-Up kernel to continue uninterrupted.

"Under the hood"

USB Communications

Through the USB connection, the PC can read and write DSP memory, and launch the execution of DSP code. PC-to-DSP USB communication use two mechanisms:

- The PC uses control pipe 0, via "Vendor Requests" to perform the following operations:
  - Reset the DSP
  - Change the color of the LED
  - Read or write on-chip DSP memory using the HPI hardware.
  - Read or write various USB registers, such as *DSPState*, and *USB Error Count*.
- The PC uses high-speed bulk pipes 2 (out) and 6 (in) to transfer data to and from any location in any DSP space, as well as launch the execution of user DSP code.

Operations performed using the control pipe zero are slow and limited in scope, but very reliable. They allow the PC to take control of the DSP at a very low-level. In particular, the memory transfers do not rely on the execution of a kernel code on the DSP. All these operations can be performed even when the DSP code is crashed. In particular, these operations are used to initialize the DSP.
Operations performed using high-speed bulk pipes 2 and 6 are supported by the resident DSP kernel. They provide access to any location in any memory space on the DSP. Transfers are much faster than those using the control pipe 0. However, they rely on the execution of the kernel. This kernel must be loaded and running before any of these operations may be attempted. These operations may not work properly when the DSP code is crashed.
Operations performed on the DSP through the kernel must follow a protocol described in the following sections.

Communication Via The Control Pipe 0

The following Vendor Requests support the operations that the PC can perform on the DSP board via control pipe 0. All these operations are encapsulated into library functions in the PC software interfaces.

Request	Direction	Code	Action
DSPReset	Out	10_H	Assert or release the DSP reset. *NOTE: Asserting the reset of the DSP also resets the KPresent and K_State variables to zero, indicating that the kernel is not present (see below).* **wValue** = 1: assert / 0: release. **wIndex** = N/A

			wLength = N/A
DSPInt	Out	11_H	Send a DSPInt interrupt through the HPI interrupt process (interrupt vector $xx64_H$).
W_Leds	Out	12_H	Change the color of the bi-color LED (green, red, orange or off). **wValue** = 0 : off **wValue** = 1 : red **wValue** = 2 : green **wValue** = 3 : orange **wIndex** = N/A **wLength** = N/A
HPIMove	In/Out	13_H	Read or write 1-to2048 16-bit words (2 to 4096 bytes) in the DSP memory accessible through the HPI. **wValue** = Lower transfer address in DSP memory map. **wIndex** = Upper transfer address in DSP memory map (XHPIA). **wLength** = Nb of bytes to transfer (must be even) **DataBlock** 1 to 2048 16-bit words can be transported in the Data Stage of the request. *NOTE: For OUT transfers, the address stored in wIndex-wValue must be pre-decremented (i.e. it must point to the address just before the first word is to be written).*
W_HPI_Control	Out	14_H	Write the HPI control register. This can be used to interrupt the DSP (DSPInt), or to clear the HINT interrupt (DSP to Host interrupt). *Note : The DSPInt and HINT signals are used by the kernel to communicate with the PC. Developers should only attempt to use this if they understand the consequences (see the kernel description).* *The BOB bit (bits 0 and 8 of the control register should always be set (never cleared). Otherwise the DSP interface will not work properly.* **wValue** = 16-bit word to write to HPIC NOTE: Both LSB and MSB bytes must be identical. **wIndex** = N/A **wLength** = N/A
Set_HPI_Speed	Out	15_H	Sets the HPI speed to slow or fast. *Note: The HPI speed is automatically set to slow at power-up and whenever the DSP is reset via the DSPReset command. Use this command to set the HPI to fast after either event.* **wValue** = 1: Fast / 0: Slow. **wIndex** = N/A **wLength** = N/A
Move_DSPState	In/Out	20_H	Reads or writes the state of the DSP. **wValue** = N/A **wIndex** = N/A **wLength** = N/A **DataBlock** : 2 bytes representing the state of the DSP: **bKpresent (byte)**:

			- Kernel not Loaded -> 0 - Power-Up Kernel Loaded -> 1 - Host-Download Kernel Loaded -> 2 The Kpresent variable is 0 at power-up. It takes a few seconds after power-up for the USB controller to load and launch the kernel. The host should poll this variable after opening the driver, and defer kernel accesses until after the kernel is loaded. **bKstate (byte)**: - Kernel_Idle -> 0
R_ErrCount	In	22_H	Returns the USB error count **wValue** = N/A **wIndex** = N/A **wLength** = N/A **DataBlock** 1 word is transported in the data block. This word represents the present USB error count (between 0 and 15).
Reset_ErrCount	Out	23_H	Resets the USB Error Count register to zero.

Communication Via The DSP Kernel :

The communication kernel enhances communications with the PC. Memory exchanges without the kernel are limited to the memory space directly accessible from the HPI. Redirection of DSP execution is limited to the boot-load of code at a fixed address immediately after reset. The kernel allows Reads, and Writes to/from any location in any space (data, program or I/O), and allows redirection of execution at any time, from any location, even in a re-entrant manner.

Actually two kernels may be used at different times in the life of a DSP application:

- Immediately after power-up, the USB controller loads a *Power-Up Kernel* in DSP memory and executes it. The USB controller performs this function on its own, whether a host PC is connected to the board or not. The kernel being functional is indicated by the LED turning orange. After this the *READ_DSPState* Vendor command will return a *KPresent* value of 1 (see Vendor Commands above).

Note: the host should only invoke kernel commands after the KPresent variable reaches a non-zero value.

This Power-Up Kernel performs the following functions:

  - It checks in Flash memory if an FPGA descriptor file is present, and if it is, loads the FPGA with the corresponding logic.
  - It then checks in Flash memory if an executable DSP file is present, and if it is it loads and executes it.
  - It stays resident to respond to kernel commands from the host (K_Read, K_Write and K_Exec - see below) once the board has been connected to a PC.
- Whenever the board is connected to a PC, the PC may reset the board and load a simpler *Host-Download Kernel* into memory at any time. This Host-Download Kernel does not check in Flash memory for FPGA logic or DSP code. It only waits for and responds to K_Read, K_Write and K_Exec commands from the host. This gives the host PC a way to take control of the DSP, and reload FPGA logic and DSP code different than what is described in the Flash memory. In particular, this is required to reprogram the Flash memory with new FPGA logic and/or DSP code.

After either of these kernels is on-line, the host PC may send K_Read, K_Write and K_Exec commands. Each command launches a DSP operation (Data move or code branch) and waits for the DSP to acknowledge completion of the operation. The DSP code that responds to the

command must include this acknowledge within 5s of execution, otherwise a timeout occurs. The intrinsic kernel functions supporting the K_Read and K_Write commands do include this acknowledge. User DSP code that is launched through the K_Exec command must absolutely include the acknowledge. For user DSP functions invoked by the K_Exec command, it is possible that (by design or not) the acknowledge take a long time to be returned. USB spec 1.0 specifies that a request should not be NAKed for more than 5s. For this reason, the acknowledge should be returned within 5s. Normally the acknowledge is used to signal the completion of the function, but for functions which take a long time to complete, the acknowledge should be returned at the beginning of the function (to signal the branch), and another means of signaling completion should be considered (polling a completion flag in DSP memory for instance).

FPGA Boot Table

An FPGA configuration table may be programmed in the Flash memory for the Power-Up Kernel to load the FPGA. This table begins at the end of the Flash, at byte-address $9FFFFF_H$ (word-address $4FFFFF_H$), and goes backwards toward byte-address $9C0000_H$ (word-address $4E0000_H$). Byte-addresses $9C0000_H$ to $9FFFFF_H$ (4 sectors) of the Flash are reserved for the FPGA configuration table.

Byte-Address	Data	Description
…	Last word	Last configuration word
…		
…		Data word
…		Data word
$9FFFF6_H$	3^{rd} and 4^{th} bytes	Data word
$9FFFF8_H$	1^{st} and 2^{nd} bytes	Data word
$9FFFFA_H$	File length (LSB)	Number of words that follow (LSB)
$9FFFFC_H$	File length (MSB)	Number of words that follow (MSB)
$9FFFFE_H$	3008_H	Magic number.

The configuration bytes are arranged LSB first. This means that the lower byte of the first word at address $9FFFF8_H$ is actually the first byte to be sent to the FPGA. The upper byte is the second, and so on…

The Power-Up Kernel checks for the presence of the Magic Number 3008_H at byte-addresses $9FFFFE_H$- $9FFFFF_H$. If it is found, it proceeds to load the data into the FPGA. The presence of the magic number alone is enough to direct the kernel to load the data. No checks are performed to make sure that the data is valid.

Note: The magic number should not be written if configuration data is not present in the Flash, because then the FPGA will be configured with erroneous data. THIS MAY DAMAGE THE FPGA.

DSP Boot Table

A DSP boot table may be programmed in the Flash memory for the Power-Up Kernel to load the DSP. This table begins at the beginning of the Flash, at byte-address 800000_H (word-address 400000_H), and goes toward byte-address $9BFFFF_H$ (word-address $4DFFFF_H$).
The DSP and FPGA boot-tables grow in opposite direction to allow the DSP boot table to have a maximum size of 917504 words (28 sectors) without interfering with the FPGA boot table. If the DSP code is smaller, the sectors between the two boot tables may be used for general purpose storage.

The “Power-Up Kernel” checks byte-address 800000_H for the presence of the Magic-Number if it is present, it proceeds to load the following sections into RAM, and branches to the entry point of the code.
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.

Note: This is a boot table. The DSP does not execute code from these locations. Instead, it reads the various sections and loads them into on-chip RAM, from which the code is then executed.

Byte-Address	Data	Description
800000_H	3009_H	Magic Number
800002_H	Nb_Sections	Total Number of sections to load into RAM
800004	Section Space 1st Section	Memory space where to load the first section
800006_H	Section Length (MSB) 1st Section	MSB of the number of words that follow
800008_H	Section Length (LSB) 1st Section	LSB of the number of words that follow
$80000A_H$	Address (MSB) 1st Section	MSB of the load address for the following section
$80000C_H$	Address (LSB) 1st Section	LSB of the load address for the following section
$80000E_H$	1st Word 1st Section	Data words
800010_H	2nd Word 1st Section	…
800012_H	…	…
…	…	…
…	Last Word 1st Section	…
…	Section Space 2nd Section	Memory space where to load the second section
…	Section Length (MSB) 2nd Section	MSB of the number of words that follow
…	Section Length (LSB) 2nd Section	LSB of the number of words that follow
	Address (MSB) 2nd Section	MSB of the load address for the following section
	Address (LSB) 2nd Section	LSB of the load address for the following section
…	1st Word 2nd Section	Data word
…	2nd Word 2nd Section	Data word
…	…	…
…	Entry Point (MSB)	After loading the various sections, the DSP transfers execution to this address.
…	Entry Point (LSB)	After loading the various sections, the DSP transfers execution to this address.

Note: The Section Lengths, Load Addresses, and Entry Point are 32-bit words. They are stored in the table MSB first. However, they do not need to be aligned on even addresses.

Constraints On The DSP Code Loaded In The Boot Flash

The DSP code loaded in Flash must contain an acknowledge at its beginning (within 5s of execution). Usually, when developing in C, this acknowledge is placed in the first lines of the main function.
Failure to include the acknowledge does not cause the DSP code to crash. However, it does cause the USB controller to fail to recognize that the Power-Up Kernel has been loaded. In this circumstance it is necessary to reset the board in order to gain access from the PC. This reset in turn causes the boot-loaded DSP code to abort.

Note: This requirement also applies to code that is written to be loaded directly from the PC and executed using the SR2_K_Exec.vi interface VI. Therefore DSP code that has been developed and tested using the mini-debugger, or code that is usually downloaded and executed from the PC should normally be ready to be programmed into the boot-table as it is.

HPI Signaling Speed

On Signal Ranger_mk2, the signaling speed of the HPI must be slow immediately after the DSP is taken out of reset. This includes after a power-up reset, and after reception of the *DSPReset* vendor command. This is because in these circumstances the DSP (and the HPI) is running at slow speed. It is only after the power-up or host-download kernel has been loaded and has had time to adjust the CPU and HPI speed to the maximum for the DSP that the USB controller may use the fast HPI signaling. This command is normally used to set the HPI speed to the maximum after the power-up kernel has been detected or after the host-download kernel has been downloaded. Note that it may take up to 500us after the kernel has adjusted the PLL, until the DSP and HPI are clocked using the fast rate. Therefore the PC software should wait for at least that amount before sending the command. The switch to high HPI signaling speed is automatically performed by the board initialization functions of the LabVIEW and C/C++ interfaces.

USB Benchmarks

The *Signal Ranger_mk2* board has a USB 2.0 port. This port connects at High-Speed (480mb/s) on any USB 2.0-capable root or hub. It connects at Full-Speed (12Mb/s) on any root or hub that is only USB 1.1-capable.

Full-Speed Timing

At Full-Speed (USB 1.1-compatible), USB frames occur at 1ms intervals. It may take up to 6 frames (up to 6ms) to perform a data transfer between the PC and the *Signal Ranger_mk2* board. The exact number of frames depends on the type of Host-Controller Software on the PC, and its ability to schedule pending USB transactions in the leftover time of the current USB frame. This means that every function of the LabVIEW and C/C++ interfaces that must get access to the USB bus may take up to 6ms to complete, even if the amount of data transferred is minimal. This represents a *minimum access time*. It is usually a factor for the functions that carry a small amount of data.
At Full-Speed, the maximum transfer rate is 4.9Mb/s for reads, 5.8Mb/s for writes. This limited transfer rate adds to the execution time of functions of the interface. The limited transfer rate is usually a factor for the functions that transport large amounts of data.

High-Speed Timing

At high-Speed (USB 2.0-capable), USB micro-frames occur 8 times more frequently than at Full-Speed (every 125μs). This means that the minimum access time is normally 8 times shorter than at Full-Speed (up to 750μs). Data throughput is limited to 18Mb/s for reads, and to 22Mb/s for writes.

Typical Benchmarks

In High-Speed on an Enhanced Host Controller, the typical access time is 475µs, and the bandwidth is 18Mb/ for reads, and 22Mb/s for writes.
In Full-Speed, on a Universal Host Controller, the typical access time is 6ms, and the bandwidth is 4.9Mb/s for reads, 5.8Mb/s for writes.

DSP Communication Kernel

Differences With Previous Versions

There are several differences between the kernels used in this version of Signal Ranger, and the kernels that were used in previous versions:

Location Of The Mailbox

The MailBox is not located at address 1000_H anymore. It is now located within the code of the kernel at address 0100_H (byte-address 0200_H). This facilitates the linking of user code, which does not have to avoid the MailBox at address 1000_H anymore. It is reflected in the *SignalRanger_mk2.mem* memory description file.

Contents Of The Mailbox

The Branch-Address and Transfer-Address in the mailbox are now 32-bit words, to support transfers and branches beyond the first 64k words of the memory spaces.
An Error-Code has been added in the mailbox, to return a possible error (or completion) code when executing user functions. This error-code may be used by the user code. Kernel functions always return the same completion code (1 for reads and 2 for writes) because they cannot fail (see further in the text).

Size Of The Mailbox

The data field of the mailbox is now 256 words when the DSP board is connected to the PC in High-Speed (480Mb/s). It is still 32 words when the DSP board is connected in Full-Speed (12Mb/s).

Addresses Of The Various Functions Of The Kernel

The various functions of the kernel are not located at fixed addresses anymore, to allow for a more compact kernel. The entry-points of the functions are different between the Power-up kernel, and the Host-Download kernel. To accommodate this, the access to these functions is now symbolic, with the symbols automatically loaded in an entry of the *Global Board Information Structure* whenever the kernel is loaded or reloaded.

Size Of The Kernel

The overall size of the kernel has changed. However, both kernels (Power-Up and Host-Download) fit completely before byte-address 0800_H (word-address 0400_H). It is reflected in the *SignalRanger_mk2.mem* memory description file.

Overview

On the DSP side, the kernel that is used to support PC communications is extremely versatile, yet uses minimal DSP resources in terms of memory and processing time. Accesses from the PC wait for the completion of time critical processes running on the DSP, therefore minimizing interference between PC accesses and real-time DSP processes.

Three USB commands (K_Read, K_Write and K_Exec), are used to trigger kernel operations. The exchange of data and commands between the PC and the DSP is done through a 262-word mailbox area in page 0 of the on-chip DSP RAM.
The DSP interface works on 3 separate levels:

- **Level 1**: At level 1, the kernel has not yet been loaded onto the DSP. The PC relies on the hardware of the DSP (HPI and DMA), as well as the USB controller, to exchange code and/or data with DSP RAM. At this level, the PC has only a limited access to the on-chip DSP RAM. For instance on the C5502, the PC can access byte-addresses between $00FE_H$ and $FFFF_H$. This level is used, among other things, to download the kernel into DSP RAM, and launch its execution.
- **Level 2**: At level 2, the kernel is loaded and running on the DSP. Through intrinsic functions of the kernel, the PC can access any location in any memory space of the DSP, and can launch DSP code from an entry point anywhere in memory. This level is used to load user code in DSP memory, and launch it. Level 1 functions are still functional at level 2, but rarely used because Level 2 functions provide more access. However, one possible advantage of Level 1 function is that they do not rely on DSP software. Therefore they always succeed, even when the DSP code is crashed.
- **Level 3:** Level 3 is defined when user code is loaded and running on the DSP. There is no functional difference between levels 2 and 3. The level 1 and 2 functions are still available to support the exchange of data between the PC and the DSP, and to redirect execution of the user DSP code. The main difference is that at level 3, user functions are available too, in addition to intrinsic functions of the kernel. At Level 3, with user code running, the K_Exec Level 2 command can still be invoked to force DSP execution to branch to a new address.

Boot Modes

As is described in earlier sections, there are actually two kernels that are used:
A *Power-Up Kernel* is downloaded directly by the on-board USB controller shortly after power-up. This kernel looks in Flash ROM to see if a DSP user-code and/or FPGA configuration file are programmed. If so, it first loads the FPGA, then loads the DSP user code and branches to its entry point. Therefore when the host PC takes control of the DSP, it is already at level 2.
Later on, the host PC may reset the DSP and reload a more limited *Host-Download Kernel*. This kernel is similar to the Power-Up Kernel, except that it ignores the contents of the Flash ROM. This way, it is possible for the host PC to completely take control of the DSP, without any interference from user-code residing in Flash ROM.

Processor State After Reset

The processor state after reset depends on the boot mode:

Host-Download Boot:

In this case the kernel performs the following initializations:

- Places the Interrupt vector table at byte-address $0x0100_H$ in the on-chip RAM.
- Initializes the following vectors that are used by the kernel:
  - **Reset vector:** Points to reset routine of the kernel
  - **DSPInt vector:** Points to DSPInt vector of the kernel. Used to manage DSP-PC communications.
  - **TRAP #31 vector:** Points to an address that is adjusted dynamically by the USB controller. Used to manage DSP-PC communications.
- Adjusts the clocks as follows:
  - CPU Core Clock: 300 MHz
  - SYSCLK1 (Fast Peripherals): 150 MHz
  - SYSCLK2 (Slow Peripherals): 75 MHz
  - SYSCLK3 (EMIF): 75 MHz

- Adjusts the stack as follows:
  - Dual stack with fast return
  - System-stack at word-address $0x8000_H$ (top of on-chip memory)
  - User-stack at word-address $0x7E00_H$ (leaves 512 words for the system-stack).
- Sets the C55x CPU mode, rather than the C54x-compatible mode (bit C54CM = 0).
- Takes the on-chip boot ROM out of the memory map (bit MPNMC = 1)
- Initializes the EMIF so that the following peripherals may be used:
  - FLASH ROM
  - SDRAM
  - FPGA
- Unmask DSPInt interrupt to allow DSP-PC communications:

Power-Up Boot And Neither User-Code Or FPGA Logic Present In Flash:

The kernel initializations are the same as for the Host-Download boot.

Power-Up Boot And FPGA Logic Detected In Flash:

In this case, in addition to the initializations performed normally, the FPGA logic specified in Flash is also downloaded into the FPGA. The FPGA logic is functional when the user-code takes control of the CPU. Also, the C environment has been established. The C environment is in effect when the user takes control of the DSP.
The contents of the CPU registers conform to the following:

- ST1_55: CPL=1 M40=0 SATD=0 SXMD=1 C16=0 FRCT=0 C54CM=0
- ST2_55: ARMS=1 RDM=0 CDPLC=0 AR[0-7]LC=0
- ST3_55: SATA=0 SMUL=0

Power-Up Boot And User-Code Executed From Flash:

In this case, in addition to the initializations performed normally, the user-code is downloaded and run. Also, the C environment has been established. The C environment is in effect at the beginning of the user-code.
The contents of the CPU registers conform to the following:

- ST1_55: CPL=1 M40=0 SATD=0 SXMD=1 C16=0 FRCT=0 C54CM=0
- ST2_55: ARMS=1 RDM=0 CDPLC=0 AR[0-7]LC=0
- ST3_55: SATA=0 SMUL=0

Resources Used By The Kernel On The DSP Side

To function properly, the kernel uses the following resources on the DSP. After the user code is launched, those resources should not be used or modified, in order to avoid interfering with the operation of the kernel, and retain its full functionality.

- The kernel resides between byte-addresses 0000_H and $07FF_H$ in the on-chip RAM of the DSP. The user should avoid loading code into, or modifying memory space below byte-address 0800_H.
- The PC (via the USB controller) uses the DSPInt interrupt from the HPI to request an access to the DSP. This interrupt in turn triggers TRAP#31, which branches to user-code or an intrinsic kernel function. If necessary, the user code can temporarily disable the DSPInt interrupt through its mask, or the global interrupt mask INTM. During the time this interrupt is disabled, all PC access requests are latched, but are held until the interrupt is re-enabled. Once the interrupt is re-enabled, the access request resumes normally.

- The kernel locates the interrupt vector table at byte-address 0100_H. The user-code should not relocate the interrupt vectors anywhere else.
- The communication kernel initializes the stack as a dual stack with fast return. The stack pointers are initialized at word-addresses $7FFF_H$ (system stack) and $7DFF_H$ (data stack). The user code can relocate the stack pointers temporarily, but should replace them before the last return to the kernel (if this last return is intended). Examples where a last return to the kernel is not intended include situations where the user code is a never-ending loop that will be terminated by a board reset or a power shutdown. In these cases, the stack can be relocated without concern.

Note: When branching to the entry point of a program that has been developed in C, the DSP first executes a function called c_int00, which establishes new stacks, as well as the C environment. This function then calls the user-defined "main". When main stops executing (assuming it is not a never-ending loop), it returns to a never-ending loop within the c_int00 function. It does not return to the kernel.

- The kernel uses the DSP in the C55x mode (not C54x compatibility mode).

Functional Description Of The Kernel

After the Power-Up Kernel finishes initializing the DSP, if it finds DSP code in Flash memory, this DSP code is normally running when the user takes control of the DSP board.
After the Host-Download Kernel finishes initializing the DSP, or after the Power-Up Kernel finishes initializing the DSP and did not find DSP code in Flash memory, the kernel is normally running when the user takes control of the DSP. The kernel is simply a never-ending loop that waits for the next access request from the PC. PC access requests are triggered by the DSPInt interrupt from the HPI.

Launching A DSP Function

At levels 2 and 3, the kernel protocol defines only one type of action, which is used to read and write DSP memory, as well as to launch a simple function (a function which includes a return to the kernel or to the previously running code) or a complete program (a never-ending function that is not intended to return to the kernel or to the previously running code). In fact, a memory read or write is carried out by launching a *ReadMem* or *WriteMem* function, which belongs to the kernel (intrinsic function) and is resident in DSP memory. Launching a user function uses the same basic process, but requires that the user function be loaded in DSP memory prior to the branch.
The mailbox is an area in page 0 of the HPI-accessible RAM of the DSP.
The function of each field in the mailbox is described below.

Address	Name	Function
0100_h-0101_h	BranchAddress	32-bit branch address (intrinsic read and write functions, or user function)
0102_h-0103_h	TransferAddress	32-bit transfer address (for K_Read and K_Write commands)
0104_h	NbWords	Number of words to transfer
0105_h	ErrorCode	User-application-specific Error Code or completion code.
0106_h-0205_h	Data	256 data words used for transfers between the PC and the DSP. Only the first 32 words are used when the board is connected as a Full-Speed USB device.

To launch a DSP function (intrinsic or user code), the PC, via the USB controller, does the following operations:
Initiates a K_Read, K_Write or K_Exec command. This command contains information about the DSP address of the function to execute (user or intrinsic). For K_Read and K_Write, it also contains information about the transfer address; and in the case of a Write transfer, it contains the data words to be written to the DSP.

- The USB controller places the DSP branch address into the *BranchAddress* field of the mailbox. This branch address may be a user branch address (in the case of a K_Exec command), or may be a kernel function address (in the case of a K_Read or K_Write command).
- If data words are to be written to the DSP (K_Write), the USB controller places these words in the *Data* field of the mailbox.
- If words are to be transferred to or from the DSP (K_Read or K_Write), the USB controller places the number of words to be transferred, between 1 and 32 (Full-Speed USB Connection), or between 1 and 256 (High-Speed USB connection) into the *NbWords* field of the mailbox.
- If words are to be transferred to or from the DSP (K_Read or K_Write), the USB controller places the DSP transfer address into the *TransferAddress* field of the mailbox.
- The USB controller clears the HINT (host interrupt) signal, which serves as the DSP function acknowledge.
- The USB controller sends a DSPInt interrupt to the DSP, which forces the DSP to branch to the intrinsic or user function.

When the DSP receives the DSPInt interrupt from the HPI, the kernel does the following:
If the DSP is not interruptible (because the DSPInt interrupt is temporarily masked, or because the DSP is already serving another interrupt), the DSPInt interrupt is latched until the DSP becomes interruptible again, at which time it will serve the PC access request.
If - or when - the DSP is interruptible, it:

- Fetches the branch address from the *BranchAddress* field in the mailbox, and writes it back into the vector for the TRAP#31 software interrupt.
- Clears INTM. From this instant, the DSP becomes interruptible again, and can serve a critical user Interrupt Service Routine (ISR). If no user ISR is pending, the DSP starts executing the function requested by the PC (intrinsic or user function).
- Triggers the TRAP #31 interrupt, which causes the DSP to branch to the function or code specified by *BranchAddress*.
- If data words are to be transferred from the PC (K_Write), the function reads those words from the *Data* field of the mailbox and places them at the required DSP addresses.
- If data words are to be transferred to the PC (K_Read), these words are written by the DSP to the *Data* field in the mailbox.
- If an error or completion code should be returned to the PC, the DSP function should update the *ErrorCode* field of the mailbox.
- After the execution of the requested function, and the update of the *ErrorCode* field, the DSP asserts the HINT signal, to signal the USB controller that the operation has completed. This operation has been conveniently defined in a macro *Acknowledge* in the example codes, and can be inserted at the end of any user function. Note that from the PC's point of view, the command seems to "hang" until the Acknowledge is issued by the DSP. User code should not take too long before issuing the Acknowledge. If the acknowledge is not returned within 5s of the transfer request, the request is aborted on the PC, and an error is issued.
- If data words are to be transferred to the PC, upon reception of the HINT signal, the USB controller fetches those words from the *Data* field in the mailbox area and sends them to the PC in the data stage of the request (K_Read).
- Once the DSP function has completed its execution, a RETI instruction must be used to return control to the kernel, or the previously executing user code, after the acknowledge. This is not necessary however, and user code can keep executing without any return to the kernel if this is what the user intends. In this case, subsequent PC accesses are still allowed, which means the kernel is re-entrant.

Note: The size of the data field of the mailbox is 256 words. However the maximum size of a transferred block of data depends on whether or not the Signal Ranger_mk2 board is connected to the PC using a Full-Speed USB connection (12Mb/s), or a High-Speed USB connection

(480Mb/s). The board is USB 2.0 compliant and enumerates at High-Speed on a USB 2.0 root or hub. When the board is connected at High-Speed, the transfers are performed 256-words at a time. When the board is connected at Full-Speed, the transfers are performed 32-words at a time. Only the first 32 words of the data field of the mailbox are used in this case.

Since a PC access is requested through the use of the DSPInt interrupt from the HPI, it can be obtained even while user code is already executing. Therefore it is not necessary to return to the kernel to be able to launch a new DSP function. This way, user functions can be re-entered. Usually, the kernel is used at level 2 to download and launch a user main program, which may or may not return to the kernel in the end. While this program is running, the same process described above can be used at level 3 to read or write DSP memory locations, or to force the execution of other user DSP functions, which themselves may, or may not return to the main user code, and so on… It is entirely the decision of the developer. If a return instruction (RETI) is used at the end of the user function, execution is returned to the code that was executing prior to the request. This code may be the kernel at level 2, or user code at level 3.

The acknowledgment of the completion of the DSP function (intrinsic or user code) is done through the assertion of the HINT signal. This operation is encapsulated in the *Acknowledge* macro in the example code for the benefit of the developer. This acknowledge operation is done for the sole purpose of signaling to the initiating host command that the requested DSP function has been completed, and that execution can resume on the PC side. Normally, for a simple user function, which includes a return (to the main user code or to the kernel), this acknowledge is placed at the end of the user function (just before the return) to indicate that the function has completed. As a matter of good programming, the developer should not implement DSP functions which take a long time before returning an acknowledge. In the case of a function which may not return to the kernel or to previously executing user code, or in any case when the user does not want the host command which initiated the access to hang until the end of the DSP function, the acknowledge can be placed at the beginning of the user function. In this case it signals only that the branch has been taken. Another means of signaling the completion of the DSP function must then be used. For instance the PC can poll a completion flag in DSP memory.

During the PC access request, the DSP is only un-interruptible during a very short period of time (between the taking of the DSPInt interrupt and the branch to the beginning of the user or intrinsic function – the equivalent of 10 cycles). Therefore, the PC access process does not block critical tasks that might be executing under interruption on the DSP (managing analog I/Os for instance).

During a transfer to and from the DSP, the DSP is also uninterruptible during the actual transfer of each a 32-word block (or 256-word block for a HighSpeed USB connection). The actual uninterruptible time depends on the target peripheral. For a read or a write from/to on-chip DSP RAM, the uninterruptible time in CPU cycles is equal to the number of words transferred. Making the actual transfer uninterruptible presents the advantage that any transfer up to 32-words (Full-Speed) or 256-words (High-Speed) is atomic from the DSP's perspective. In particular it insures that when transferring double words, both the high and low part of the word are transferred simultaneously. However, the block size is small enough that the transfer is quick and should not pose a problem in most situations.

When transferring data to and from a slow peripheral such as the SDRAM, the uninterruptible time may be up to 240ns/word. If the USB connection is high-speed, the block size is 256 words. In such a case, the uninterruptible time may be so long that it interferes with the service of other interrupts in the system. In such a situation two actions can be taken to solve the problem:

- Either split the transfer at high-level so that only small blocks are transferred at a time.
- Or use a custom DSP function to handle the transfer in place of the intrinsic kernel function. Design this custom function to be non-atomic and interruptible. The examples folder contains an example of such a custom function.

Note: At the beginning of a DSP user function, it is required to protect (store) all the DSP registers that are used within the function, and to restore them just before the return. Since the function is launched through the DSPInt interrupt, it is done asynchronously from the main executing code (kernel or user code). Responsibility of register protection must then be assumed by the called user function. The situation is the same as for any interrupt function (it actually is an interrupt function). All the intrinsic functions of the kernel perform this protection and will not interfere with user code.
When the function is written in C it must be declared with the "interrupt" qualifier. Otherwise all the registers used within the function may not be protected.
See sections on DSP Code Development for details.

Memory Read And Write

The kernel includes 6 intrinsic functions (*ReadMem*, *WriteMem*, *ReadProg*, *WriteProg*, *ReadIO* and *WriteIO*), which are part of it, and are resident in memory at the time the kernel is executing. These 6 functions allow the PC to read or write the DSP memory.

ReadMem, ReadProg, ReadIO :

These functions read *nn* successive words from the DSP memory at address *aaaaaaaa* (1<=*nn*<=32 or 1<=*nn*<=256, depending on the USB connection speed).
To launch the execution of any of these functions, the PC (via the USB controller and the invocation of K_Read) does the following:

- Writes the 32-bit entry point of the function (ReadMem, ReadProg or ReadIO resp.) into the *BranchAddress* in the mailbox.
- Writes the 32-bit DSP transfer address *aaaaaaaa*, the *TransferAddress* in the mailbox.
- Writes the number *nn* of words to transfer, into *NbWords* in the mailbox.
- Sets the *ErrorCode* field of the mailbox to 1
- Clears the *HINT* signal.
- Sends a *DSPInt* interrupt to the DSP.

In response to these actions, the kernel does the following:

- Branches to the beginning of the function and becomes interruptible again.
- Pushes all the registers used within the function on the top of the stack (context protection).
- Reads the beginning transfer address *aaaaaaaa* in the *TransferAddress* field of the mailbox
- Reads *nn* words in the DSP memory and writes them into the *Data* field of the mailbox.
- Places the address directly following the block that has just been transferred into the *TransferAddress* variable in the mailbox. This way subsequent accesses do not have to reinitialize the address, to begin transferring the following words.
- Restores the context.
- Asserts the *HINT* signal, which signals the completion of the operation.
- Returns to the caller.

At this point, the USB controller does the following:

- Reads the *nn* words from the data field of the mailbox and sends them to the PC.
- Reads the *ErrorCode* field of the mailbox and sends it back to the PC. This error code is 1 for reads.

The USB pipe (Pipe 6) that supports the exchange has a size of 64 bytes (32 words) for a Full-Speed USB connection, and a 512 bytes (256 words) size for a High-Speed connection. For transfers larger than the pipe size, the transfer is broken down into blocks of the pipe size. The USB controller invokes the function and carries out the above operations for each of the pipe-sized segments. Assertion of the *HINT* signal at the end of each segment triggers the transfer of the block to the PC.

In this case, the *ErrorCode* field of the mailbox is only set to one by the on-board USB controller before the first segment. The value of the *ErrorCode* field that is carried back to the PC is the value that has been last updated by the DSP before the last acknowledge.

Note: The number of words to read must not be zero.

WriteMem, WriteProg, WriteIO:

These functions write *nn* successive words into the DSP memory, from address *aaaaaaaa* (1<=*nn*<=32 or 1<=*nn*<=256, depending on the USB connection speed).
To launch the execution of any of these functions, the PC (via the USB controller and the invocation of K_Write) does the following:

- Places the *nn* words to write to DSP memory into the *Data* field of the mailbox.
- Writes the entry point of the function (WriteMem, WriteProg or WriteIO resp.) into the *BranchAddress* variable in the mailbox.
- Writes the DSP transfer address (*aaaaaaaa*), into the *TransferAddress* variable in the mailbox.
- Writes the number *nn* of words to transfer, into the *NbWords* field in the mailbox.
- Sets the *ErrorCode* variable of the mailbox to 2.
- Clears the *HINT* signal.
- Sends a *DSPInt* interrupt to the DSP.

In response to these actions, the kernel does the following:

- Branches to the beginning of the function and becomes interruptible again.
- Pushes all the registers used within the function on the top of the stack (context protection).
- Reads the transfer address in the *TransferAddress* variable of the mailbox
- Reads the *nn* words from the *Data* field of the mailbox and writes them back to the DSP memory at the *aaaaaaaa* transfer address.
- Places the address directly following the block that has just been transferred into the *TransferAddress* variable in the mailbox. This way subsequent accesses do not have to reinitialize the address, to begin transferring the following words.
- Restores the context.
- Asserts the *HINT* signal, which signals the completion of the operation.
- Returns to the caller.

At this point, the USB controller does the following:

- Reads the *ErrorCode* field of the mailbox and sends it back to the PC. This error code is 2 for writes, since kernel functions do not use the code.

The USB pipe (pipe 2) that supports the exchange has a size of 64 bytes (32 words) for a Full-Speed USB connection, and a 512 bytes (256 words) size for a High-Speed connection. For transfers larger than the pipe size, the transfer is broken down into blocks of the pipe size. The USB controller invokes the function and carries out the above operations for each of the pipe-sized segments. Assertion of the *HINT* signal at the end of each segment triggers the transfer of the next block from the PC.
In this case, the *ErrorCode* field of the mailbox is only set to 2 by the on-board USB controller before the first segment. The value of the *ErrorCode* field that is carried back to the PC is the value that has been last updated by the DSP before the last acknowledge.

Note: The number of words to read must not be zero.

Use Of The K_Read And K_Write Requests For User Functions

In principle, the K_Read and K_Write requests are used only to invoke the 6 intrinsic kernel functions. However, nothing bars the developer from using these requests to invoke a user

function. This may be useful to implement user functions that need to receive or send data from/to the PC, because it gives them a way to efficiently use the mailbox, and the on-board USB controller transfer process. To achieve this, the user function should behave in exactly the same manner as the *intrinsic* functions do for Read resp. Write transfers. The *BranchAddress* field of the mailbox should contain the entry point of a user function, rather than the address of an intrinsic kernel function.
More details about this can be found in the description of the *SR2_Base_User_Move_Offset* VI in the *LabView Interface* section, and in the *SR2_DLL_User_Move_Offset_I16* function in the *C/C++ Interface* section.

It should be noted that arguments, data, and parameters can alternately be passed to/from the PC into static DSP structures by regular (kernel) K_Read or K_Write requests after and/or before the invocation of any user function. This provides another, less efficient but more conventional way to transfer arguments to/from DSP functions.

High-Speed Communication Protocol

High-Speed communication with the DSP board is achieved through the use of bulk pipes 2 (out) and 6 (in). The PC uses bulk pipe 2 to send packets to the DSP, and pipe 6 to receive packets from the DSP. Communication through these pipes must follow the protocol described below.

The host PC can trigger three types of operation on the DSP:
- K_Read: The execution of a DSP function which brings back data from the DSP via the Mailbox.
- K_Write: The execution of a DSP function which sends data to the DSP via the Mailbox.
- K_Exec: The execution of a DSP function without any data transport.

The kernel provides intrinsic functions to perform basic read and write operations from/to any location in any memory space of the DSP. However, the user may also provide other read and write functions that use the same K_Read and K_Write mechanism. This gives the developer the ability to override the kernel functions with DSP code providing more specialized functions. For instance, read and write user functions might also handle the synchronization and pointer operations required to manage a FIFO on the DSP.

All three types of operations described above progress in the same way:
- The PC first sends a Setup Packet to the DSP board, indicating various information elements such as the direction of transfer, transfer address, DSP branch address…etc. The structure of the Setup Packet is the same for all three types of command.
- For a K_Write, the PC sends the data to the DSP board, for a K_Read, the PC collects the data sent back by the DSP board. For a K_Exec, this step is skipped.
- The PC then waits for a Completion Packet from the DSP board, indicating that the requested operation has completed.

Setup Packet

The Setup Packet contains the following information:
- A 32-bit DSP Branch Address ***BranchAddress***. This is the address in DSP memory where the function to be executed resides (read, write or user function).
- A 32-bit DSP Transfer Address ***TransferAddress***. This is the address in DSP memory where the data is to be read or written. For an execution without data transfer (K_Exec) the address is ignored.
- A 16-bit Transfer Count ***NbWords*** number. For an execution without data transfer (K_Exec) the Transfer Count is ignored. The transfer count is expressed in words. It must be between 1 and 32768.

- A 16-bit Operation Type ***OpType*** code. This code is 0 for a K_Exec, 1 for K_Read, 2 for K_Write.

Byte Nb	Data
0	BranchAddress (LSB)
1	BranchAddress (byte 1)
2	BranchAddress (byte 2)
3	BranchAddress (MSB)
4	TransferAddress (LSB)
5	TransferAddress (byte 1)
6	TransferAddress (byte 2)
7	TransferAddress (MSB)
8	NbWords (LSB)
9	NbWords (MSB)
10	OpType (LSB)
11	OpType (MSB)

Note: The data in the setup packet is copied into the corresponding fields of the mailbox by the USB controller before the DSP function residing at BranchAddress is called. These are the same fields that are present at the beginning of the mailbox. The OpType field of the setup packet corresponds to the ErrorCode field of the mailbox. Therefore in the case of a user DSP function, this is the data that resides in the mailbox when the user function executes.

Note: In the case of a multi-packet data transfer (packets being 32-byte long in Full-Speed and 256-byte long in High-Speed), the above fields of the mailbox are only updated before the FIRST call of the DSP function, corresponding to the transfer of the first packet. Subsequent calls corresponding to subsequent packets do not modify these fields. However, this statement only holds true if the multi-packet transfer is smaller than 32768 words, and if the transfer does not cross a 64 kWords boundary. This is because transfers larger than 32768 words are segmented at high-level into 32768-word transfers. At the beginning of each such transfer the whole contents of the mailbox are updated. Similarly, transfers that cross a 64 kWords boundary are split into two transfers that do not straddle the boundary. For each of these transfers, the whole contents of the mailbox are updated.

Completion Packet

After the operation, the DSP board sends back a Completion Packet to the PC, indicating that the operation has completed. This Completion Packet is built as follows:

- A 16-bit **Error Code**. This code is the content of the *ErrorCode* field of the mailbox that has been updated by the DSP function prior to sending the last acknowledge to the USB controller. This error code should be used in an application-specific manner. Consequently, the user should define the relevant error codes if any are to be used.

Byte Nb	Data
0	ErrorCode (LSB)
1	ErrorCode (MSB)

Note: In case of a multi-packet data transfer (K_Read or K_Write), the error code that is returned to the PC as part of the completion packet represents the content of the ErrorCode field of the mailbox before the acknowledge corresponding to the LAST call of the DSP function.

Note: This Error Code does not represent conditions that may occur if the transfer does not complete (for instance if the kernel is not on-line or if the DSP code is crashed at the time of the transfer), in such conditions, the PC operation completes abnormally, and indicates a USB error. The transmission of the Error Code can only occur if the transfer otherwise completes normally. An Error Code may be used to signal application-specific conditions, such as the DSP not being able to accept data at this time because a FIFO is full, or not being able to execute a function because of the state it is in at the moment it receives the command.

Note: In the case of a K_Read or K_Write transaction, whether or not the Completion Code is set to a meaningful value, and whether or not the DSP is able to absorb or supply the data, the DSP function that handles the transfer must send an acknowledge for each elementary pipe-sized transfer of the complete transaction. For instance, if the Signal_Ranger_mk2 board is connected to the PC as a Full-Speed device, the pipe size is 64 bytes (32 words). For a K_Read of 68 words, the complete transaction is segmented into two 32-word transactions, followed by one 4-word transaction. In this case, the DSP must acknowledge each of the three elementary segment transfers, even if it is not able to supply the data. In such a case, an appropriate Error Code could be used to indicate to the PC in an application specific manner that the data collected from the mailbox is not valid. The Error Code is read by the USB controller after the last acknowledge is received from the DSP. In the above example, the DSP can set the error code at any stage in the transfer. However the value that is returned to the PC is the value last updated before the last acknowledge.

DSP Support Code

DSP Flash Driver And Flash Programming Support Code

Several levels of support code are provided to the developers:

- A DSP driver library is provided to the developers who wish to include Flash programming functions into their DSP code. This driver code is described below.
- A Flash programming DSP application is provided to support the Flash programming functionality that is part of the interface libraries (LabVIEW and C/C++), as well as the mini-debugger interface. This code is not described below. It is provided as an executable file named *SR2_Flash_Support.out*. This DSP code is loaded and executed by the interface functions that require its presence.

Overview Of The Flash Driver

A DSP driver is provided to help the user's DSP code development. This driver takes the form of a library named **SR2FlashBootDriver.lib**.
The driver is found in the *C:\Program Files\SignalRanger_mk2\DSPSupport_Code*. Simply unzip the file to access its contents.

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. The driver uses a software 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 60 μs per word. Erasure is asynchronous, and may be quite long (typ 0.5s/sector, max 3.5s per sector). Erasure functions wait until all writes are completed before beginning. They are blocking, which means that execution is blocked within the erasure function as long as the erasure is not completed.

Read, write and erase addresses are 32 bits.

*Note: All addresses passed to and from the driver are **byte-addresses**. This is true, even though the Flash memory is not byte-addressable. At the lowest level all operations are performed and counted in 16-bit words. The numbers of operations to perform (read, write and erase) are specified to the driver in **number of 16-bit words**. Nonetheless, all addresses are byte-addresses. This is done to provide consistency with the rest of the Signal_Ranger_mk2 interface software that only uses byte-addresses. Since the Flash is not byte-addressable, the byte-addresses that are passed to the driver are divided by two internally to point to the correct 16-bit words. Addresses that are returned from the driver are multiplied by two internally before being returned.*

Reads are very simple. They are performed asynchronously using the **SR2FB_Read** function. This function returns the content of any 32-bit address.

Writes are performed sequentially using the **SR2FB_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 **SR2FB_SetAddress** function, or the **SR2FB_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 within the sector.

Note: Contrary to previous generations of Flash devices that have been used in Signal Ranger boards, the Flash device used in Signal_Ranger_mk2 cannot be incrementally programmed. This means that a word location that has been previously programmed MUST be erased before reprogramming. This is true even if the reprogramming operation is only intended to turn some of the remaining "1s" into "0s".

The **SR2FB_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 **SR2FB_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 **SR2FB_FIFOState** function.

Setup Of The Driver

Note: Contrary to the Flash drivers that have been provided in the past with previous versions of Signal Ranger boards, this driver requires the C environment to work properly. This means that driver functions should only be called from code written in C, or from code that has setup the C environment prior to calling any of the functions (see details below).

- All the functions of the driver that are defined below are contained in the **SR2FlashBootDriver.lib** library. The user code must be linked with this library to function properly (the library must be added to the project source files).
- When linking the library with a C project, the project must use the LARGE memory model. This is required to be able to access all sectors of the FLASH, which span multiple pages of 64k.
- Since the writes are performed under INT1 interrupts, the INT1 interrupt vector must be initialized to the "**SR2FBINT**" label. This label is the entry point of the INT1 interrupt routine. The INT1 interrupt routine is defined in the **SR2FlashBootDriver.lib** library. See the C example code provided with the board for an example of interrupt vector setup.
- A header file named **SR2_FB_Driver.h** is provided that declares all the functions of the driver.
- Before any other function of the driver is called, the driver must be initialized using the **SR2FB_Init** function.

C-Environment

Because the functions are C-Callable, the driver assumes the presence of the C-environment. This environment is in effect by default when calling any function of the driver from code that has been written in C. However, when calling these functions from code written in assembly the following requirements should be observed:
The contents of the CPU registers should conform to the following:

- ST1_55: CPL=1 M40=0 SATD=0 SXMD=1 C16=0 FRCT=0 54CM=0
- ST2_55: ARMS=1 RDM=0 CDPLC=0 AR[0-7]LC=0
- ST3_55: SATA=0 SMUL=0

In accordance to the C environment rules, the following registers are not protected by the driver functions, and may be modified by any of the functions:
AC[0-3], XAR[0-4], T[0,1], ST0_55, ST1_55, ST2_55, ST3_55, RPTC, CSR, BRC0, BRC1, BRS1, RSA0, RSA1, REA0 and REA1.

For information about C-calling rules, see Texas Instrument's relevant documentation.

Data Structures

FB_WriteFIFO

FB_WriteFIFO is a 32-word buffer accessed with FIFO access logic. The FIFO itself is not user-accessible. It may only be written by the **SR2FB_Write** function. It is only emptied under INT1 interrupt service routine.

FB_WriteAddress

FB_WriteAddress is a 32-bit unsigned word that always contains the address of the next word to be written. **FB_WriteAddress** can be initialized by the **SR2FB_SetAddress** function or the **SR2FB_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 **SR2FB_FIFOState** function.

FB_WriteEraseError

FB_WriteEraseError is an integer that contains various error status bits. It is returned by several functions, including **SR2FB_SetAddress**, **SR2FB_FIFOState**, **SR2FB_WritePrepare** and **SR2FB_Write**.
Once an error bit is set to one, indicating an error, it stays one until the error word is cleared using **SR2FB_ErrorClear**. Execution of **SR2FB_Init** also clears the error word.

Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
													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 at an address outside of the useable address range, or the ROM is not working 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. Either an erasure was attempted at an address outside the useable address range, or the Flash is not working properly.

*Note: When a write is attempted at an address that was not previously erased the usual behaviour is that the process locks up indefinitely. The WP bit stays one indefinitely. There is no timeout to unlock the write process. The next writes to the write FIFO may be accepted, but the FIFO is not being emptied, therefore at some point the FIFO gets full and the **SR2FB_Write** function blocks.*

User Functions

unsigned long SR2FB_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 detected. 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.

Input: no input
Output: no output
Return: The size of the flash ROM in word

unsigned int SR2FB_SetAddress(unsigned long FB_WAddress)

This function waits for all pending writes to complete. Then it sets the **FB_WriteAddress** pointer to the 32-bit byte-address passed in argument. The function does not check to make sure that the address passed in argument is inside the allowable address range. If it is not, the subsequent writes will simply fail. The function returns the current **FB_WriteEraseError** status.

Input: unsigned long **FB_WAddress** (byte): this is the 32-bit address for the next write
Output: no output
Return: The current WriteEraseError

unsigned short SR2FB_FIFOState(unsigned int *FB_FIFOCount, unsigned long *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

(*FB_Waddress* is converted to a byte-address internally before being returned). 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.*

Input: ushort * FB_FIFOCount: This is the pointer to a variable for FIFOCount output
unsigned long* WAddress : This is the pointer to a 32-bit variable FB_WAddress output (sp(2))
Output: ushort FB_FIFOCount and unsigned long FB_WAddress
Return: The current FB_WriteEraseError

void SR2FB_ErrorClear()

The function clears the current **FB_WriteEraseError** status register.
Input: no input
Output: no output
Return: no return

int SR2FB_Read(unsigned long FB_RAddress)

The function returns the word read from the *FB_RAddress* byte-address. Note that no check is performed to insure that the read occurs in the memory space occupied by the Flash ROM. This function may be used to return the contents of any memory, regardless of its type.
Input: unsigned long FB_RAdress (byte), this is the 32-bit read byte-address.
Output: no output
Return: The word read at the specified byte-address

unsigned int SR2DFB_WritePrepare(unsigned long FB_WAddress, unsigned long 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* byte-address. It then initializes the **FB_WriteAddress** register to the value of *FB_WAddress*, so that the next call to **SR2FB_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.
The function does not check to make sure that the erasure does not include any addresses outside the useable address range of the Flash.
If during the preparation a sector erase is attempted outside the range of useable addresses, the function simply fails.
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.

Note: The behaviour of the function is undefined if the requested FB_WSize is zero.

Input: unsigned long FB_Waddress This is the starting byte-address of the segment to prepare
unsigned long FB_WSize This is the size of the segment to prepare

Output: no output
Return: The current FB_WriteEraseError

unsigned int SR2FB_Write(int Data)

The function places the value of *Data* in the write FIFO. It normally returns without waiting for the write to be completed. The low-level 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 60µs per word to program, so if the function is called while the FIFO is full, it may not return before 60µs 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**.
The function does not check to make sure that the write is attempted inside the range of useable addresses. If it is not, then the write will simply fail. The failure will not be detected until the data is actually written from the FIFO to the Flash however. 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.

Input: short Data : this is the data to place in the FIFO
Output: no output
Return: The current FB_WriteEraseError