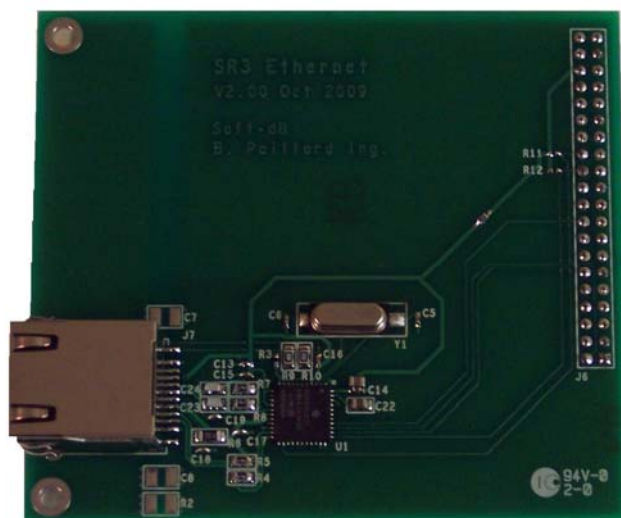


SR3-Ethernet

User's Manual



by



In association with



October 16 2009

1	MAIN FEATURES	4
2	TRANSFER PERFORMANCE	4
3	LED INDICATORS	4
4	LABVIEW INTERFACE	5
4.1	Preliminary Remarks	5
4.2	Opening the Target	5
4.2.1	Loading and Executing Code Dynamically	6
4.3	Network Interface Vis	6
4.3.1	Core Interface VIs	6
4.3.2	Flash Support VIs	11
5	C/C++ INTERFACE	16
5.1	Execution Timing and Thread Management	17
5.2	Calling Conventions	17
5.3	Building a Project Using Visual Studio	17
5.4	Exported Interface Functions	18
5.4.1	SR3_DLL_Net_Open_Next_Avail_Board	18
5.4.2	SR3_DLL_Net_Close_BoardNb	19
5.4.3	SR3_DLL_Net_Bulk_Move_Offset_U8	19
5.4.4	SR3_DLL_Net_K_Exec	21
5.4.5	SR3_DLL_Net_Flash_InitFlash	21
5.4.6	SR3_DLL_Net_Flash_EraseFlash	22
5.4.7	SR3_DLL_Net_Flash_FlashMove_U8	22
6	DSP SUPPORT CODE	23
6.1	Operation Of The <i>Net_Kernel</i>	23
6.2	Resources Used By The <i>Net_Kernel</i>	25
6.2.1	Code Size	25
6.2.2	Execution Time	25

1 Main Features

SR3_Ethernet is an add-on board that provides network connectivity to the *SignalRanger_mk3* platform. The network connection provides services similar to the USB connection, with the following differences:

- Using the network connection the *SignalRanger_mk3* board can be accessed in stand-alone mode from anywhere a network connection is available. Using this connection *SignalRanger_mk3* can be controlled from remote locations in a local network, or through the internet.
- The network connection is not as fast as the USB connection.
- The network connection is not as forceful as the USB connection. In particular it does not allow the controlling computer to take control of the board when its firmware has crashed.
- The network connection does not support dynamic reloading of DSP code in RAM. However it does support reloading of new firmware in Flash, and rebooting the board from this newly reflashed code.
- The network connection has a much larger footprint on the user code than the USB connection in terms of required memory and execution time.
- The network driver on the DSP runs in the foreground, usually in the main loop of the user code.

The *SR3_Ethernet* board and its associated firmware feature:

- IEEE 802.3-compliant fast Ethernet connection.
- Supports 10/100Base-T port with automatic polarity detection and correction.
- Supports auto-negotiation.
- Supports TCP/IP, DHCP, UDP, ARP, DNS, NetBIOS Name Service Server, DDNS, ICMP.

The software support, both at the level of the board and at the level of the host uses the TCP/IP protocol to communicate with and control the board. A higher-level protocol called *Net_DDCI* encapsulates all the communication and control functions.

The *SignalRanger_mk3* behaves as a TCP server. It responds to socket number 50000d.

2 Transfer Performance

The transfer speed and reliability is very much dependant on the particular LAN network to which the board is connected. The following bandwidths have been measured on a dedicated LAN:

Host Read	Host Write
3.75 Mb/s	3.75 Mb/s

3 LED Indicators

The Ethernet jack includes two LEDs:

- The green LED indicates that the link is established.
- The yellow LED flashes when Ethernet packets are received by the board.

4 LabVIEW Interface

4.1 Preliminary Remarks

The *SR3_Net* LabVIEW interface includes functions similar to those of the native USB interface. Operation of these functions requires that the DSP user code include the *Net_Kernel* library. See section on DSP support code below for details.

*Note: Even though they are similar in form and function, the *SR3_Net* interface and the *SR3_Base* interface are not compatible. Trying to connect any of the *SR3_Net* interface VIs to an *SR3_Base* BoardRef control will create a run-time error. So will the reverse.*

4.2 Opening the Target

Opening a target through the network connection is very similar to opening a target through the USB connection, except that board communications go through a TCP connection, rather than through a USB connection.

The *SR3_Net* LabVIEW interface can manage multiple simultaneous target connections. Each time a target connection is opened, the *SR3_Net_Open_Next_Avail_Board.vi* Vi opens a TCP connection to the target and creates a data structure that represents the target board. The VI returns a *BoardRef_Out* indicator. This *BoardRef_Out* indicator is used as a handle on the specific target. It represents the specific target and its associated data structure. All the VIs in the interface use this handle as an input.

The connection to a specific target is exclusive. Once a target is opened using *SR3_Net_Open_Next_Avail_Board.vi*, no other host or application can take control of this particular target until the present connection is closed.

After the connection to the board is no longer required, the *SR3_Net_Close_BoardNb* is used to close the TCP connection, release the associated data structure, and allow other host or applications to take control of the board.

A particular target can only have one connection with a host application. A host application on the other hand can have connections with - and manage - multiple target boards.

The *SR3_Net_Open_Next_Avail_Board.vi* uses an identifier string to open the target. This string points to a database within the LabVIEW interface that contains the hardware characteristics of the board, such as its board model, DSP type, Flash addresses... etc. When the target is opened, this information is stored in a global variable array. There is one array element, representing one target, for every target opened by the host.

After opening the board, the *SR3_Net_Open_Next_Avail_Board.vi* reads the DSP code and FPGA file names in Flash, if any, as well as their checksums. If a DSP firmware is present in Flash, the *SR3_Net_Open_Next_Avail_Board.vi* then loads the symbol table that follows the code in Flash. This symbol table is used to provide symbolic access to the DSP code, including the network functions themselves. This information is then also stored in the global variable array so that every VI in the interface can use it.

Optionally the *SR3_Net_Open_Next_Avail_Board.vi* can selectively open only the targets that have a DSP/FPGA file name pair that is part of a provided list. This provides the basis to selectively open the products and revision numbers provided in the list. Practically the *SR3_Net_Open_Next_Avail_Board.vi* briefly opens the target, reads the file names from Flash and closes the target back if the file names and checksums are not part of the provided list.

4.2.1 Loading and Executing Code Dynamically

Contrary to the native *SR3_Base* interface libraries, the *SR3_Net* libraries do not provide functions to load DSP code dynamically. The closest way to implement these functions is to reprogram the Flash and reboot the DSP code from the Flash.

4.3 Network Interface Vis

The Network interface is organized as several folders in the *Signal_Ranger_mk3.lvlib* library. 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.

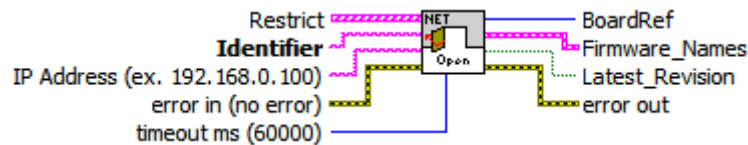
These VIs and libraries operate in the same way as the VIs and libraries that support the USB connection. They have similar names, often with a “Net” in their name.

4.3.1 Core Interface VIs

4.3.1.1 SR3_Net_Open_Next_Avail_Board

This Vi performs the following operations:

- Tries to find an available DSP board with the selected identifier or IP address, and optionally that has the firmware indicated in the *Restrict* control.
- If it finds one, creates an entry in the *Global Board Net Information Structure*.
- If a DSP firmware is detected in Flash (code has been loaded and started as part of the power-up sequence), loads the corresponding file name and symbol table from Flash.
- Places the symbol table of the kernel in the *Global Board Net Information Structure*.



Controls:

- **Restrict:** This is a structure used to restrict the access by firmware names and checksums. If this control is connected and not empty, the access is restricted to the boards having a pair of DSP and FPGA file names with corresponding checksums in the list provided. The firmware in Flash must match both names and checksums in an element of the array for the board to be accepted. Each element in the list usually represents a particular revision of the product's firmware. This control should be wired to restrict the opening to boards that have been configured as specific products, and avoid opening boards used by other OEMs, or other products of the same OEM.
- **Identifier:** This is a string used to identify the hardware type of board to open. This control must always be wired, and must be initialized with the correct symbolic name for the board. There are several hardware variations of the *SignalRanger_mk3* architecture, including custom implementations. For instance the standard *SignalRanger_mk3* board has an identifier equal to *SRM3*.
- **IP-Address:** If the IP address is wired, then the VI tries to open a board at the specified IP address, and verifies that it is the proper platform by checking it against the *Identifier*. If the *IP-Address* is not wired, then the VI tries to open the first board on the network that has the proper identifier.
- **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.
- **Timeout ms** If the TCP connection to the specified board does not occur within the specified time-limit, the VI returns with an error.

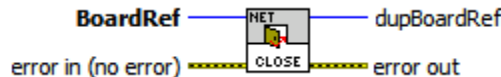
Indicators:

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the in *Global Board Net Information Structure*. The interface can manage a multitude of boards connected to the same PC. Each one has a corresponding *BoardRef* number allocated to it when it is opened. All other interface Vis use this number to access the proper board.
- **Firmware_Names:** Cluster containing the names of the DSP and FPGA firmware files that are found in Flash. The fields are empty if the Flash does not contain any firmware. The fields are also empty if the *ForceReset* control is true.
- **Latest_Revision:** This indicator is *true* if the pair of firmware file names and checksums found in Flash correspond to the last element provided in the *Restrict* array. In most implementations the *Restrict* array contains the name-pairs of different firmware revisions of a given product, in ascending order. When this is the case the *Latest_Revision* indicator is true when the firmware detected in the Flash of the board is indeed the latest firmware known to the controlling application.
- **Error out:** LabVIEW instrument-style error cluster. Contains error number and description.

Note: Once a connection has been opened with a board, the board will not respond to other clients requesting a connection. A consequence of this is that a board cannot be opened twice. A board that has already been opened using the SR3_Net_Open_Next_Avail_Board VI cannot be opened again until it is properly closed using the SR3_Net_Close_BoardNb VI. This is especially a concern when the application managing the board is shut-down under abnormal conditions. If the application is shut-down without properly closing the connection to the board, the next execution of the application may fail to find and open the board, simply because the corresponding TCP socket is still open on the board side. In such a case simply cycle the power on the board, or disconnect and reconnect the network cable to force the board to close its local socket.

4.3.1.2 SR3_Net_Close_BoardNb

This Vi Closes the TCP connection used to access the board, and frees the corresponding resources in the *Global Board Information Structure*. It is used after the last access to the board has been made, to release 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 *SR3_Net_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 *SR3_Net_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.

4.3.1.3 SR3_Net_Bulk_Move_Offset

This VI reads or writes an unlimited number of data words to/from the address space of the DSP, using the *Net_Kernel*.

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 DSP's C compiler.

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 U8 on the DSP side (cast the required type to an array of U8 to write it to the DSP, read an array of U8 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* may be required to be aligned to the proper width, depending on the specific platform.
- 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, not in elements of the specified type. This is required to access an individual member of an heterogeneous structure.
- In case of a write of a data type narrower than the native type for the platform, then additional elements are appended to complete the write to the next boundary of the native type. The appended values are set to all FF_H.

<i>Note: The Net_Kernel must be loaded and executing for this Vi to be functional.</i>
--

<i>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.</i>

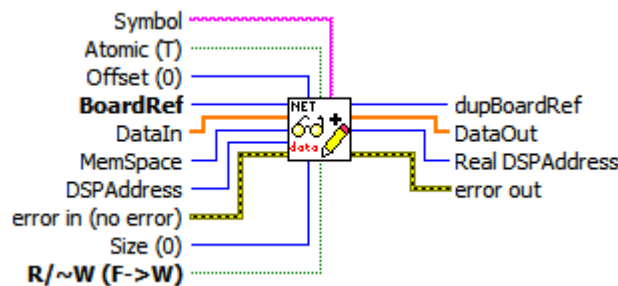
<i>Note: When reading or writing scalars, Size must be left unwired.</i>
--

4.3.1.3.1 Notes on Transfer Atomicity

Contrary to the native USB kernel, the *Net_Kernel* performs all its transfers synchronously to the execution of the foreground DSP user code. A consequence of this is that a host-initiated transfer cannot cut in the middle of a data operation executed in the foreground on the DSP. Symmetrically, a foreground DSP operation cannot cut in the middle of a host-initiated data transfer.

This remark does not include DSP code that executes under interrupts. A host-initiated transfer cannot cut in the middle of a data operation on the DSP executed in an interrupt service routine. However a data operation executed on the DSP in an interrupt service routine can cut in the middle of a host-initiated transfer.

In short, the integrity of the wide data transferred to and from the DSP is maintained when transferring data that is not processed under interrupts. The only case when the developer should be careful is when transferring data that is processed (written) by the DSP code under interrupts. In this case there is always the possibility of the interrupt processing cutting in the middle of the host-initiated transfer, and writing over parts of the data being transferred. This is especially a concern when the data transferred is wider than the byte. In this case, portions of a word can be overwritten by the DSP interrupt while the others are not, corrupting the wider transferred word. When this is a concern we recommend the use of a secondary buffer and semaphores to sequence 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 *SR3_Net_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.
- **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).
- **Atomic:** Boolean indicating if the transfer is made atomic or not. This control is present for compatibility with the equivalent native USB VI, but it has no effect in the network interface.

- **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 *SR3_Net_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. The value of this indicator is irrelevant in this interface.
- **Error out:** LabVIEW instrument-style error cluster. Contains error number and description.

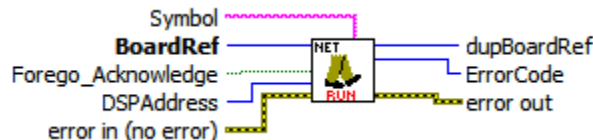
4.3.1.4 SR3_Net_K_Exec

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.

Contrary to the native USB interface there is no need for an acknowledge in the user DSP function. A USB acknowledge in the function will not cause any problems however. In other words, a DSP function called by a native USB *SR3_Base_K_Exec* can also be called by *SR3_Net_K_Exec*.

Note: Contrary to the native USB kernel, the Net_Kernel cannot be reentered. 2 conditions must be observed for SR3_Net_K_Exec to work properly:

- *The called function must return. Calling a function that does not return will stop the operation of the Net_Kernel*
- *The called function must be relatively short. The kernel is stopped, and the host blocks until the called function returns.*



Controls:

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_Open_Next_Avail_Board.vi*.
- **Forego_Acknowledge:** When this boolean is true, the VI does not wait for the acknowledge sent back by the board to signal the end of the function. This is only used in very rare circumstances when the VI is used to launch code in a “shoot and forget” manner. In normal circumstances the boolean should be left unconnected or set to false.
- **DSPAddress:** Physical branch address. It is used if for the branch if *Symbol* is 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 *SR3_Net_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. For the *Net_Kernel*, this completion code is 0x32.
- **Error out:** LabVIEW instrument-style error cluster. Contains error number and description.

4.3.2 Flash Support VIs

These VIs are provided to support Flash-programming operations. The VIs equally support Flash-reading operations and Flash programming operations for symmetry. However reading the Flash does not require these special functions and can be carried out by *SR3_Net_Bulk_Move_Offset*.

The VIs in this library require that the DSP code be placed in a special *Park* state. This is done by the *SR3_Net_Flash_InitFlash* VI. In this state the DSP disables ALL interrupts and only services the *Net_Kernel* by polling. In effect, all previously running code is aborted. Even the native USB kernel is rendered inoperative. The only way to get out of this state is to call the VI *SR3_Net_Reload_from_Flash*.

4.3.2.1 SR3_Net_Flash_InitFlash

This VI places the DSP in the *Park* state. All previously running DSP code is aborted. Contrary to the native USB kernel, the Flash-support code is already resident in the *Net_Kernel* and does not need to be loaded.



Controls:

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_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 *SR3_Net_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. The flash is not detected, the VI returns a constant value.
- **Error out:** LabVIEW instrument-style error cluster. Contains error number and description.

4.3.2.2 SR3_Net_Reload_from_Flash

This VI exits the *Park* state by the following sequence:

- Branches to the *BootFlashUserCode* function of the kernel. This function:
 - Loads the DSP firmware and FPGA logic that it finds in Flash
 - Branches to it

- This causes the board to reboot, therefore losing its current network link. A short time after rebooting the board recovers its link status, IP address...etc.
- Closes the current TCP connection on the host.
- Reopens the connection to the board using the same IP address that was in effect before the close.

For the new TCP connection to work, it is required that:

- The DHCP server on the LAN allocates the same IP address to the board that was in effect just before the reboot.
 - The newly loaded code includes an implementation of the *Net_Kernel*.
- In effect this sequence implements the newly flashed code.



Controls:

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_Open_Next_Avail_Board.vi*
- **Timeout ms** If the operation does not occur within the specified time-limit, the VI returns with an error. It is normal for the operation to take a few seconds, since the board must be rediscovered on the LAN by the DHCP server.
- **Error in:** LabVIEW instrument-style error cluster. Contains error number and description of the previously running VI.

Indicators:

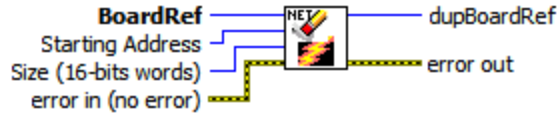
- **DupBoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_Open_Next_Avail_Board.vi* Use this output to propagate the reference number to other Vis.
- **Error out:** LabVIEW instrument-style error cluster. Contains error number and description.

4.3.2.3 SR3_Net_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 than are actually selected. For instance, if the starting address is not the first word of a sector, words in the same sector before the starting address will be erased. 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: On SignalRanger_mk2_Next_Generation the sector size is 32 kwords. On SignalRanger_mk3 the sector size is 64 kwords (128 kBytes).

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 *SR3_Net_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 *SR3_Net_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.

4.3.2.4 SR3_Net_FlashFlashMove

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 *SR3_Net_Bulk_Move_Offset* should be used instead, since it does not require that the Flash be placed in the *Park* state.

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 U8 on the DSP side (cast the required type to an array of U8 to write it to the DSP, read an array of U8 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 write 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 the SignalRanger_mk2_NG generation, incremental programming (programming the same address multiple times) is permitted on SignalRanger_mk3. Each programming operation cannot set individual bits from 0 to 1. This can only be done by an erasure cycle on a whole sector.

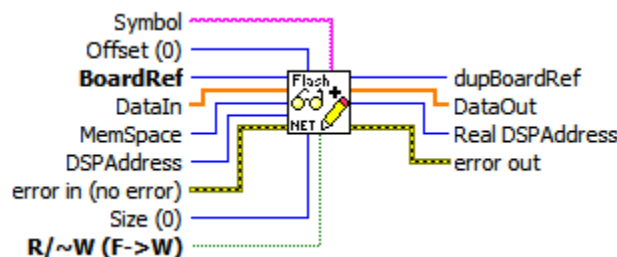
However the programming operations can reset individual bits from 1 to 0 at any time without intervening erasure.

In case of a write of a data type narrower than 16-bits (the native type for the Flash) additional elements are appended to complete the write to the next 16-bit boundary. The appended values are set to all 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 Flash's internal representation is 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 *SR3_Net_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.
- **MemSpace:** This control is not used. It is only presented for compatibility with other transfer VIs.
- **DSPAddress:** Physical base DSP address for the transfer.
- **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 *SR3_Net_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.

4.3.2.5 SR3_Net_Flash_Config_NoDialog

This VI automatically programs a DSP file and/or an FPGA file into Flash. It does not require user interaction, but presents its front-panel to the user during the programming so that the operation can be monitored. The VI presents error and/or completion dialogs.



Controls:

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_Open_Next_Avail_Board.vi*
- **DSP File In:** File path of the DSP file to be programmed in Flash. No file is programmed if the path is empty. New in *SignalRanger_mk3*, *File Path* can point to a firmware container VI, as well as an actual COFF (.out) file.
- **FPGA File In:** File path of the FPGA file to be programmed in Flash. No file is programmed if the path is empty. New in *SignalRanger_mk3*, *File Path* can point to a firmware container VI, as well as an actual FPGA (.rbt) file.
- **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 *SR3_Net_Open_Next_Avail_Board.vi* Use this output to propagate the reference number to other Vis.
- **Success:** Returns at *true* if the load succeeded. Otherwise returns at *false*.
- **Load_Attempted:** Always returns at *true*. This boolean is provided for compatibility reasons.
- **Error out:** LabVIEW instrument-style error cluster. Contains error number and description.

4.3.2.6 SR3_Net_Flash_Config_Dialog

This VI programs a DSP file and/or an FPGA file into Flash. Contrary to *SR3_Flash_Config_NoDialog* the VI is interactive. It prompts the user for the input files and the action (*Write* or *Clear*). The VI presents error and/or completion dialogs. New in *SignalRanger_mk3*, The DSP and FPGA files can be contained in firmware container VIs, as well as actual .out or .rbt files.



Controls:

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_Open_Next_Avail_Board.vi*
- **Error in:** LabVIEW instrument-style error cluster. Contains error number and description of the previously running VI.

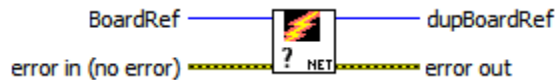
Indicators:

- **DupBoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_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.

4.3.2.7 SR3_Flash_Check_Dialog

This VI checks the Flash contents against a DSP file and/or an FPGA file. The VI is interactive. It prompts the user for the input files and the action (*Check* or *Cancel*). The VI presents error and/or completion dialogs. New in *SignalRanger_mk3*, The DSP and FPGA files can be contained in firmware container VIs, as well as actual .out or .rft files.



Controls:

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_Net_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 *SR3_Net_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.

5 C/C++ Interface

The C/C++ interface is provided in the form of a DLL named *SRm3_HL.dll*. This interface has been designed in a mirror image of the LabVIEW interface. The documentation of the LabVIEW interface to a large extent also applies to the C/C++ interface.

The *SR3_Net* functions of the *SRm3_HL.dll* interface are similar to those of the native USB interface. They are recognized by the *_Net* in their name. Operation of these functions requires that the DSP user code include the *Net_Kernel* library. See section on DSP support code below for details.

Note: Even though they are similar in form and function, the SR3_Net interface and the SR3_Base interface are not compatible. Trying to call any of the SR3_Net interface functions using a BoardRef control from the SR3_Base interface will create a run-time error. So will the reverse.

This interface has been designed with C/C++ development in mind, and has only been tested on version 2005 of Microsoft's Visual Studio. However, it may be possible to use it with other development environments allowing the use of DLLs.

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

- *SRm3_HL.dll* The main interface DLL

Furthermore, the LabView 2009 run-time engine must be installed on the computer that needs to use the DLL. The LabView 2009 run-time engine is installed automatically during the **SignalRangerMk3** software installation. 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 developed, the LabView 2009 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 Studio. This example is found in *C:\Program Files\SignalRanger_mk3\Visual_Studio_Code_Example*.

5.1 Execution Timing and Thread Management

Two functions of the *SRm3_HL* DLL accessing the same *SignalRanger_mk3* DSP board cannot execute concurrently. The first function must complete before the second one can be called. Care should be taken in multi-threaded environments to ensure that separate functions of the DLL do not run at the same time (in separate threads). The simplest method is to ensure that all calls to the DLL functions are done in the same thread. However, functions of the interface accessing different boards can be called concurrently.

All the functions of *SRm3_HL* DLL are blocking. They do not return until the requested action has been performed on the board.

5.2 Calling Conventions

The functions are called using the C calling conventions.

All the functions return a *USB_Error_Code* in the form of a 32-bit signed integer. This error code is zero if no error occurred.

Whenever a function must return an array or string, the corresponding space (of sufficient size) must be allocated by the caller, and a pointer to this space is passed to the function. In addition the size of the element that has been allocated by the caller is passed to the function. The *size* argument associated with the array or string normally follows the array or string in the argument line.

5.3 Building a Project Using Visual Studio

To build a project using Visual Studio the following guidelines should be followed. An example is provided to accelerate the learning curve (see last section of the current chapter).

- If the project is linked statically to the *SRm3_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 Studio 2005, it can be found in *Program Files\Microsoft Visual Studio 8\VC\lib*); in *Project Properties*, under *Linker\Command Line\Additional Options*, add the command */DELAYLOAD:SRm3_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 *SRm3_HL.lib* library without using the *DELAYLOAD* function.
- Add *#include "SRm3_HL.h"* in the main.
- If using the *DELAYLOAD* function to link statically to the *SRm3_HL.lib* library, add *SRm3_HL.lib* to the project.
- The following files must be placed in the folder containing the project sources:
 - *cvilvsb.h*
 - *extcode.h*
 - *fundtypes.h*
 - *hosttype.h*
 - *ILVDataInterface.h*
 - *ILVTypeInterface.h*
 - *platdefines.h*

- SRm3_HL.h

All these files are part of the provided example.

5.4 Exported Interface Functions

5.4.1 SR3_DLL_Net_Open_Next_Avail_Board

5.4.1.1 Prototype

```
int32_t SR3_DLL_Net_Open_Next_Avail_Board(char Identifier[], char IP_Address[], int32_t timeout_ms, int32_t *BoardRef, char DSP_Firmware_Name[], int32_t DSP_Firmware_Name_Size, char FPGA_Logic_Name[], int32_t FPGA_Logic_Name_Size)
```

5.4.1.2 Description

This function performs the following operations:

- Tries to find an available DSP board with the selected identifier or IP address
- If it finds one, creates an entry in the *Global Board Net Information Structure*.
- If a DSP firmware is detected in Flash (code has been loaded and started as part of the power-up sequence), loads the corresponding file name and symbol table from Flash.
- Places the symbol table of the kernel in the *Global Board Net Information Structure*.

5.4.1.3 Inputs

- **Identifier:** This is a string used to identify the hardware type of board to open. This argument must be initialized with the correct symbolic name for the board. There are several hardware variations of the *SignalRanger_mk3* architecture, including custom implementations. For instance the standard *SignalRanger_mk3* board has an identifier equal to *SRM3*.
- **IP-Address:** If the IP address is not empty, then the function tries to open a board at the specified IP address, and verifies that it is the proper platform by checking it against the *Identifier*. If the *IP-Address* is empty, then the VI tries to open the first board on the network that has the proper identifier.
- **Timeout ms** If the TCP connection to the specified board does not occur within the specified time-limit, the function returns with an error.
- **DSP_Firmware_Name_Size** A string of sufficient length must be allocated for the function to return the name of the DSP file present in Flash. *DSP_Firmware_Name_Size* must be set to the actual size allocated for the string.
- **FPGA_Logic_Name_Size** A string of sufficient length must be allocated for the function to return the name of the FPGA file present in Flash. *FPGA_Logic_Name_Size* must be set to the actual size allocated for the string.

5.4.1.4 Outputs

- **BoardRef** This is a number pointing to the entry corresponding to the board in the in *Global Board Information Structure*. The interface can manage a multitude of boards connected to the same PC. Each one has a corresponding *BoardRef* number allocated to it when it is opened. All other interface functions use this number to access the proper board.
- **DSP_Firmware_Name[]** This string contains the name of the DSP firmware file that is present in Flash. The string is empty if the Flash does not contain any firmware. The string is also empty if the *ForceReset* control is true.
- **FPGA_Logic_Name[]** This string contains the name of the FPGA logic file that is present in Flash. The string is empty if the Flash does not contain any FPGA logic. The string is also empty if the *ForceReset* control is true.

Note: The handle that the interface provides to access the board is exclusive. This means that only one application and one host 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 `SR3_DLL_Net_Open_Next_Avail_Board()` function cannot be opened again until it is properly closed using the `SR3_DLL_Net_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 may fail to find and open the board, simply because the corresponding driver instance is still open. In such a case simply disconnect and reconnect the board to force the PC to re-enumerate the board.

5.4.2 SR3_DLL_Net_Close_BoardNb

5.4.2.1 Prototype

`int32_t SR3_DLL_Net_Close_BoardNb(int32_t BoardRef)`

5.4.2.2 Description

This function Closes the TCP connection used to access the board, and frees the corresponding resources in the *Global Board Information Structure*. It is used after the last access to the board has been made, to release resources that are not used anymore.

5.4.2.3 Inputs

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by `SR3_DLL_Net_Open_Next_Avail_Board()`

5.4.2.4 Outputs

None

5.4.3 SR3_DLL_Net_Bulk_Move_Offset_U8

5.4.3.1 Prototype

`int32_t SR3_DLL_Net_Bulk_Move_Offset_U8(int32_t BoardRef, uint16_t ReadWrite, char Symbol[], uint32_t DSPAddress, uint16_t MemSpace, uint32_t Offset, uint8_t Data[], int32_t Size)`

5.4.3.2 Description

This function reads or writes an unlimited number of bytes to/from the address space of the DSP, using the *Net_Kernel*.

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

- If *Symbol* is not empty, 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 empty, then *DSPAddress* is used as the byte-address for the transfer, and *MemSpace* is used as the memory space.
- Note that *DSPAddress* may be required to be aligned to the proper width, depending on the specific platform.
- 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.

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

5.4.3.2.1 Notes on Transfer Atomicity

Contrary to the native USB kernel, the *Net_Kernel* performs all its transfers synchronously to the execution of the foreground DSP user code. A consequence of this is that a host-initiated transfer cannot cut in the middle of a data operation executed in the foreground on the DSP. Symmetrically, a foreground DSP operation cannot cut in the middle of a host-initiated data transfer.

This remark does not include DSP code that executes under interrupts. A host-initiated transfer cannot cut in the middle of a data operation on the DSP executed in an interrupt service routine. However a data operation executed on the DSP in an interrupt service routine can cut in the middle of a host-initiated transfer.

In short, the integrity of the wide data transferred to and from the DSP is maintained when transferring data that is not processed under interrupts. The only case when the developer should be careful is when transferring data that is processed (written) by the DSP code under interrupts. In this case there is always the possibility of the interrupt processing cutting in the middle of the host-initiated transfer, and writing over parts of the data being transferred. This is especially a concern when the data transferred is wider than the byte. In this case, portions of a word can be overwritten by the DSP interrupt while the others are not, corrupting the wider transferred word. When this is a concern we recommend the use of a secondary buffer and semaphores to sequence the transfer.

5.4.3.3 Inputs

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_DLL_Net_Open_Next_Avail_Board()*
- **ReadWrite:** 1->Read, 0->Write.
- **Symbol:** Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* and *MemSpace* are used.
- **DSPAddress:** Physical base DSP address for the exchange. *DSPAddress* is only used if *Symbol* is empty.
- **MemSpace:** Memory space for the exchange (data, program or 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.
- **Data:** Array of bytes to be written to or read from DSP memory.
- **Size:** Represents the number of bytes to transfer. For a read or a write the *Data* array allocated must be larger or equal to *Size*.

5.4.3.4 Outputs

- **Data:** Array of bytes written to or read from DSP memory. The *Data* array passed in argument to the function must be larger or equal to *Size*.

5.4.4 SR3_DLL_Net_K_Exec

5.4.4.1 Prototype

int32_t **SR3_DLL_Net_K_Exec**(int32_t BoardRef, char Symbol[], uint32_t DSPAddress)

5.4.4.2 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.

Contrary to the native USB interface there is no need for an acknowledge in the user DSP function. A USB acknowledge in the function will not cause any problems however. In other words, a DSP function called by a native USB *SR3_DLL_K_Exec* can also be called by *SR3_Net_K_Exec*.

Note: Contrary to the native USB kernel, the Net_Kernel cannot be reentered. 2 conditions must be observed for SR3_Net_K_Exec to work properly:

- *The called function must return. Calling a function that does not return will stop the operation of the Net_Kernel*
- *The called function must execute in a timely fashion. The kernel is stopped, and the host blocks until the called function returns.*

5.4.4.3 Inputs

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_DLL_Net_Open_Next_Avail_Board()*
- **Symbol:** Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* is used.
- **DSPAddress:** Physical base DSP address for the exchange. *DSPAddress* is only used if *Symbol* is empty.

5.4.4.4 Outputs

None

5.4.5 SR3_DLL_Net_Flash_InitFlash

5.4.5.1 Prototype

int32_t **SR3_DLL_Net_Flash_InitFlash**(int32_t BoardRef, double *FlashSize)

5.4.5.2 Description

This function places the DSP in the *Park* state. All previously running DSP code is aborted. Contrary to the native USB kernel, the Flash-support code is already resident in the *Net_Kernel* and does not need to be loaded.

5.4.5.3 Inputs

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_DLL_Net_Open_Next_Avail_Board()*

5.4.5.4 Outputs

- **FlashSize:** This argument returns the size of the Flash. The flash is not detected, the function returns a constant value.

5.4.6 SR3_DLL_Net_Flash_EraseFlash

5.4.6.1 Prototype

int32_t **SR3_DLL_Net_Flash_EraseFlash**(int32_t BoardRef, uint32_t StartingAddress, uint32_t Size)

5.4.6.2 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 than are actually selected. For instance, if the starting address is not the first word of a sector, words in the same sector before the starting address will be erased. 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: On SignalRanger_mk2_Next_Generation the sector size is 32 kwords. On SignalRanger_mk3 the sector size is 64 kwords (128 kBytes).

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

5.4.6.3 Inputs

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_DLL_Net_Open_Next_Avail_Board()*
- **Starting Address:** Address of the first word to be erased.
- **Size:** Number of words to be erased.

5.4.6.4 Outputs

None

5.4.7 SR3_DLL_Net_Flash_FlashMove_U8

5.4.7.1 Prototype

int32_t **SR3_DLL_Net_Flash_FlashMove_U8**(int32_t BoardRef, uint16_t ReadWrite, char Symbol[], uint32_t DSPAddress, uint8_t Data[], int32_t Size)

5.4.7.2 Description

This function reads or writes an unlimited number of bytes to/from the Flash memory. Note that if only Flash memory reads are required the function *SR3_DLL_Net_Bulk_Move_Offset()* should be used instead, since it does not require that the Flash be placed in the *Park* state.

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

The write 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 the SignalRanger_mk2_NG generation, incremental programming (programming the same address multiple times) is permitted on SignalRanger_mk3. Each programming operation cannot set individual bits from 0 to 1. This can only be done by an erasure cycle on a whole sector. However the programming operations can reset individual bits from 1 to 0 at any time without intervening erasure.

The Flash's internal representation is 16-bit words. In case of a write of an odd number of bytes an additional byte is appended to complete the write to the next 16-bit boundary. The appended byte is set to FF_H.

5.4.7.3 Inputs

- **BoardRef:** This is a number pointing to the entry corresponding to the board in the *Global Board Information Structure*. It is created by *SR3_DLL_Net_Open_Next_Avail_Board()*
- **ReadWrite:** 1->Read, 0->Write.
- **Symbol:** Character string of the symbol to be accessed. If *Symbol* is empty, *DSPAddress* is used.
- **DSPAddress:** Physical base DSP address for the exchange. *DSPAddress* is only used if *Symbol* is empty.
- **Data:** Array of bytes to be read from or written to Flash.
- **Size:** Represents the number of bytes to transfer. For a read or a write the *Data* array allocated must be larger or equal to *Size*.

5.4.7.4 Outputs

- **Data:** Array of bytes read from or written to Flash. The *Data* array passed in argument to the function must be larger or equal to *Size*.

6 DSP Support Code

To support network connectivity the user DSP code must include the following additions:

- The library *Net_Kernel_xxx.lib* must be linked with the user DSP code. This library is platform-dependant. There is one version of the library for each variant of the *SignalRanger_mk3* platform. For instance the library for the standard variant of the *SignalRanger_mk3* is *Net_Kernel_SRM3.lib*.
- The function *InitStackIP()* must be called once at the beginning of the code, before any other call to network functions.
- The function *StackTaskMain()* must be called at regular intervals. This function manages all the operations of the *Net_Kernel*. The rate at which this function is called has an impact on the transfer performance. Generally this function is called from within the main application loop.

6.1 Operation Of The *Net_Kernel*

All the tasks performed by the *Net_Kernel* are executed within the function *StackTaskMain()*. This function performs all the tasks necessary to manage the network connection at every protocol level. The execution time of this function is highly variable, and depends a great deal upon the state of the network connection and the data coming in through it.

The function *StackTaskMain()* is a large finite-state machine and only performs the operations required at each state. This function never blocks. It takes care automatically of all the network management tasks, including Link management, DHCP, TCP-Open...etc. In addition, this function manages the execution of *Net_Kernel* tasks at the highest level.

Contrary to the native USB interface, the network interface only operates by polling. If *StackTaskMain()* is not called, the interface stops working. A corollary of this is that all the operations of the *Net_Kernel* are executed synchronously from within *StackTaskMain()*. No operation can be executed asynchronously of the user code as in the case of the native USB interface.

All communications between the host and the *SignalRanger_mk3* board follow a master-slave protocol called *Net_DDCI* that is constructed on top of TCP. For all the transactions the *Net_DDCI* protocol follows the sequence below:

Before any transaction can take place, the host initiates a TCP connection to the board at socket No 50000. This is done by the *SR3_Net_Open_Next_Avail_Board* VI. When no more transactions are required the host closes the TCP connection. This is done by the *SR3_Net_Close_BoardNb* VI.

For each transaction:

- The host (client) sends a header to the *SignalRanger_mk3* board. This header defines the operation and various parameters (see below).
- If the transaction is a write, the host sends all the bytes to write to the *SignalRanger_mk3* board. If the transaction is a read the *SignalRanger_mk3* board sends all the requested bytes back to the host.
- Finally the *SignalRanger_mk3* board sends an acknowledge to the host. The acknowledge indicates that the operation is complete. The acknowledge is defined on one byte. It has always the same value (0x32)

Note: For a write the bytes sent to the board and the header can be merged into a single transfer.

Note: Contrary to the native USB interface, the Net_DDCI interface does not require the user code to send an acknowledge to signal completion of a user function. Because user functions cannot be launched asynchronously (they are launched from within StackTaskMain()) The acknowledge is send automatically when the user function is entered. A USB acknowledge can be present in the function without problems.

The transaction header is constructed as follows:

Field	Size (bytes)	Usage
<i>TaskCode</i>	1	Defines the type of transaction (see below for list of Task Codes)
<i>Address</i>	4	This is a byte address. It has different meanings depending on the transaction. It is transmitted LSB-first
<i>Length</i>	4	This is a byte length It has different meanings depending on the transaction. It is transmitted LSB-first

There are 6 types of transactions:

Transaction Name	TaskCode	Function
KNet_Read	1	Performs a read from the DSP address space. <i>Address</i> represents the transfer address in the DSP space. <i>Length</i> represents the number of bytes to transfer.
KNet_Write	2	Performs a write to the DSP address space. <i>Address</i> represents the transfer address in the DSP space. <i>Length</i> represents the number of bytes to transfer.

KNet_Exec	3	Calls the DSP function at the address specified by <i>Address</i> . Length is irrelevant.
KNet_Park	4	Enters a section of code where all interrupts are disabled, and the only task is managing the network connection. This is important in some situations to avoid interference between critical tasks and the user code that may have been executing prior to the park. For instance this transaction is executed before writing to the Flash. The only way to exit this state is to branch to the beginning of the kernel to reload the DSP firmware present in Flash and execute it.
KNet_FlashWrite	5	Performs a programming of the Flash. <i>Address</i> represents the transfer address in the Flash space. Length represents the number of bytes to transfer. Length must always be an even number.
KNet_Identity	6	Returns a 16-byte string that represents the platform type. The string is NULL terminated. In addition the first space indicates the end of the useful part of the string.
KNet_FlashErase	7	Performs an erasure of the Flash. <i>Address</i> represents the first byte to erase in the Flash space. Length represents the number of bytes to erase. Erasure can take a long time (up to 1s per sector) so the host code must be prepared to wait adequately for the acknowledge.

6.2 Resources Used By The *Net_Kernel*

Contrary to the native USB kernel, the *Net_Kernel* uses large amounts of CPU execution time and memory. This is why it does not run under interrupts. Even running in the foreground, the developer should be aware of typical execution times to understand and avoid interference with other higher priority code. The execution time of the *StackTask()* function is typically very short, with large peaks when processing occurs.

6.2.1 Code Size

Type	Size
Program Code	44 672 bytes
Initialized Data	1 172 bytes
Uninitialized Data	929 bytes

6.2.2 Execution Time

Operation/Condition	Peak Time	Typical Time
---------------------	-----------	--------------

<i>InitStackIP()</i>	4 ms	4 ms
<i>StackTask()</i> Link establishment, DHCP discover...etc.	1.9 s	16 μ s
<i>StackTask()</i> No specific communication	250 μ s	16 μ s
<i>StackTask()</i> <i>Net_Debugger</i> connection establishment	2 ms	17 μ s
<i>StackTask()</i> <i>Net_Debugger</i> disconnection	500 μ s	16 μ s
<i>StackTask()</i> 1-byte read or write	340 μ s	16 μ s
<i>StackTask()</i> 1000-byte read or write	2 ms	16 μ s
<i>StackTask()</i> 10k-byte read or write	14 ms	16 μ s