Fieldbus

NI-FBUS™ Function Block Shell Reference Manual

January 1998 Edition
Part Number 321016C-01

Internet Support

support@natinst.com
E-mail: info@natinst.com
FTP Site: ftp.natinst.com
Web Address: http://www.natinst.com

Bulletin Board Support

BBS United States: (512) 794-5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support

(512) 418-1111

Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678

International Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, United Kingdom 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway   Austin, TX 78730-5039   Tel: (512) 794-0100

Important Information

Warranty

The Fieldbus hardware is warranted against defects in materials and workmanship for a period of one year from the date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

NI-FBUS™ is a trademark of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

About This Manual

Chapter 1
Function Block Shell Overview

Chapter 2
Functional Overview

Chapter 3
Registration Functions

Chapter 4
Callback Functions

Chapter 5
Utility Functions

Chapter 6
Alarm Functions

Chapter 7
Miscellaneous Functions

Chapter 8
Serial Functions

Appendix A
Customer Communication

Glossary

Tables

About This Manual

This manual describes the main features of the National Instruments NI-FBUS Function Block Shell and describes how to use it to develop Function Block Applications.

How to Use the Manual Set

Use the *Getting Started with Fieldbus* manual to install and configure your Fieldbus hardware, the Fieldbus Stack Interface Library, and the NI-FBUS Function Block Shell software.

Use the *MC68331-Based Fieldbus Round Card User Manual* or *Intel 80188EB-Based Fieldbus Round Card User Manual* to install your Fieldbus Round Card.

Use this *NI-FBUS Function Block Shell Reference Manual* to learn about writing Function Block server applications that interface to your AT-FBUS or are embedded in the Fieldbus Round Card.

Use the *NI-FBUS Monitor User Manual* to learn to use the interactive NI-FBUS Monitor utility with your Fieldbus hardware.

Use the *NI-FBUS Communications Manager User Manual* to learn to use the interactive Fieldbus dialog system with your Fieldbus hardware.

Use the *NI-FBUS Configurator User Manual* to learn to use the NI-FBUS Configurator to configure your Fieldbus network.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *Function Block Shell Overview*, gives an introduction to the Function Block Shell.
- Chapter 2, *Functional Overview*, introduces some of the key concepts of the Function Block Shell and provides an overview of some of the functional components of the interface.

- Chapter 3, *Registration Functions*, describes the registration process and the associated functions.
- Chapter 4, *Callback Functions*, describes the callback functions of the Function Block Shell.
- Chapter 5, *Utility Functions*, describes the utility functions of the Function Block Shell.
- Chapter 6, *Alarm Functions*, describes the alarm functions of the Function Block Shell.
- Chapter 7, *Miscellaneous Functions*, describes miscellaneous functions of the Function Block Shell.
- Chapter 8, *Serial Functions*, describes serial functions of the Function Block Shell.
- Appendix A, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.

Conventions Used in This Manual

The following conventions are used in this manual:

< > — Angle brackets enclose the name of a key on the keyboard (for example, <Esc>).

\- — A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-Alt-Delete>.

☞ — This icon to the left of bold italicized text denotes a note, which alerts you to important information.

bold — Bold text denotes the names of menus, menu items, parameters, dialog box, dialog box buttons or options, icons, windows, Windows 95 tabs, or LEDs.

bold italic — Bold italic text denotes an note, caution, or warning.

italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value, as in Windows 3.*x*.
`italic monospace`	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.
`monospace`	Text in this font denotes text or characters that should literally enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and for statements and comments taken from programs.

Related Documentation

The following document contains information that you may find helpful as you read this manual:

- *Fieldbus Foundation Specification, which includes the following items:*
  - *Fieldbus Foundation System Management Services*
  - *Function Block Application Process, Part 1*
  - *Function Block Application Process, Part 2*

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix A, *Customer Communication*, at the end of this manual.

Function Block Shell Overview

Chapter 1

This chapter gives an introduction to the Function Block Shell.

The National Instruments NI-FBUS Function Block Shell is an interface between a Function Block Shell application and the National Instruments Fieldbus Foundation Communication Protocol Stack. It requires minimal knowledge of the Fieldbus Communication protocol.

This section details the main features of the National Instruments Function Block Shell, which greatly eases the development of Function Block Applications.

- The Function Block Shell constructs and maintains the Object Dictionary (OD). Therefore, you do not have to construct or maintain the OD. The Function Block Shell constructs the OD during registration, when you inform the Function Block Shell about all the VFDs, Blocks, and parameters in the application. After the OD is constructed, the Function Block Shell automatically responds to GetOD requests from the Fieldbus without your intervention.
- The Function Block Shell maintains the linkage objects as defined in the *Function Block Application Process, Parts 1 and 2*. The Function Block Shell also establishes all types of connections, including trend and event connections (also known as QUU, or Queued User-triggered Unidirectional, connections) and publisher/subscriber connections. The Function Block Shell automatically responds to connection requests from a remote device.
- The Function Block Shell handles the communication aspects of the alert-processing state machine. At your request, the Function Block Shell sends an alert notification message and waits for an alert confirmation. It can repeat the notification, confirm when the notification is received, and alert you when an acknowledgment arrives.
- The Function Block Shell maintains and reports trends. The existence of trend objects can be entirely transparent once the initial registration process is complete. You specify the trend

information, and the Function Block Shell creates the trend objects and samples the trend. Then, it reports the trend when the trend buffer is full, or when a host device requests it.

- The Function Block Shell can handle FMS Read/Write requests without involving you. However, you have the option to be involved in read/write requests if necessary.
- The Function Block Shell *snaps* (reads from the communications stack) input parameters before function block execution, and snaps output parameters at the end of block execution. You do not have to make calls to the communications stack to perform these functions.
- The Function Block Shell is not dependent on the type of Function Block; it accommodates new blocks or parameters without change.

Chapter 2

Functional Overview

This chapter introduces some of the key concepts of the Function Block Shell and provides an overview of some of the functional components of the interface.

Application Program Structure

The `userStart` function is the starting point of your application. This function is invoked automatically after the kernel boots up. In the `userStart` function, you can call the registration function to set up the application, and then call `shInitShell` to initialize the Function Block Shell. After this, the Function Block Shell remains in a loop, listening to the requests from the communication stack and invoking your callback functions to service the requests when necessary.

Registration

Registration is the process by which a function block application informs the Function Block Shell of the characteristics of the application. For each VFD (Virtual Field Device), you must supply the following information to the Function Block Shell: user-defined data types, physical blocks, transducer blocks, function blocks, parameters in the blocks, and callback functions. In addition, you must specify some other general configuration information. Registration must take place before you can interact with the Function Block Shell or the Fieldbus.

The registration process is described in the next chapter, Chapter 3, *Registration Functions*.

Ownership of Data

The Function Block Shell constructs and owns the OD. However, there are several options for the ownership of the function block parameter data. Each physical block, function block, and transducer block parameter has an ownership attribute. The ownership attribute is required; it is not network visible, so it does not affect interoperability. The Function Block Shell receives the type of ownership (user-owned or Shell-owned) for every block parameter during registration. The type of ownership determines whether the Function Block Shell or you (the user) have direct access to data. The owner can access the data as if it were any other variable in the program. The non-owner can access the data by function call (user to Shell) or by callback (Shell to user).

User-owned parameters can be either **USER_ALONE** or **USER_PTR**, and Shell-owned parameters can be either **SHELL_ALONE** or **SHELL_NOTIFY**. National Instruments recommends the use of **USER_PTR** ownership because it has less overhead than other ownership types. To access parameters of **SHELL_ALONE** and **SHELL_NOTIFY** ownerships, you must call `shReadParam` and `shWriteParam` functions. These calls can create significant overhead for a function block algorithm that needs to access a lot of parameters. **USER_PTR** ownership is easier to work with than **USER_ALONE** ownership because it requires less work in callback functions.

User-Owned Parameters

If the user owns a parameter, the Function Block Shell may or may not have direct access (by pointer) to a user-owned parameter, depending on your choice.

- **USER_ALONE**: In **USER_ALONE** ownership, you own the parameter data. The Function Block Shell does not have direct access (via pointer) to the data. Whenever the Function Block Shell needs access to the parameter to respond to a remote FMS Read or FMS Write request, it executes one of the two callback functions you registered previously. To read a value, the Function Block Shell executes the callback function of type `CB_READ`. To write a value, the Function Block Shell executes the callback function of type `CB_WRITE`.
- **USER_PTR**: In **USER_PTR** ownership, you own the data, and you inform the Function Block Shell of the pointer to the parameter in the function block registration process. The Function Block Shell has direct access to the data. In this scheme, semaphores are created to ensure mutual exclusion for accessing parameter data. One semaphore is created for each function block, including the resource block and the transducer block. In your application program, before you can access a parameter with **USER_PTR** ownership, you must use the `shWaitBlockSem` function to acquire

the semaphore of the block to which the parameter belongs. After you access the parameter, you must use `shSignalBlockSem` to release the semaphore.

On a remote read or write request of parameters with this type of ownership, the Function Block Shell asks your permission to read or write the data. The Function Block Shell asks permission by executing user-registered callback functions of type `CB_NOTIFY_READ` or `CB_NOTIFY_WRITE`.

Shell-Owned Parameters

If the Shell owns a parameter, you do not have direct access (by pointer) to the parameter. You must make a local `shReadParam` or a `shWriteParam` function call to read or write the parameter. There are two variations of this type of parameter ownership:

- **SHELL_ALONE**: In **SHELL_ALONE** ownership, the Function Block Shell owns the data, and it responds to remote read and write requests to parameters with this attribute without your involvement. In this case, no application-specific validation can be performed before honoring the remote read or write request.
- **SHELL_NOTIFY**: In **SHELL_NOTIFY** ownership, the Function Block Shell owns the data, but it executes the `CB_NOTIFY_READ` or `CB_NOTIFY_WRITE` callback function to seek your permission on remote requests to read or write the parameter.

Registering Callback Functions

The Function Block Shell uses callback functions to request service from you. You should specify a set of callback functions for a given VFD during VFD registration.

If a certain callback is not needed, you must specify the value NULL. For example, if a VFD has no **USER_ALONE** parameters, the read and write callback functions are not needed, and you can register the callback functions as the NULL value. However, if the VFD has **USER_ALONE** parameters, you must provide proper read and write callback functions to handle the access of the parameters. In this case, a NULL callback function or a callback function that does not handle data access properly might crash the program.

The Function Block Shell invokes one of these callback functions depending on the type of service it needs from you. You can call the

Function Block Shell from within the callback functions. The callback function categories are parameter access, function block execution, and alert notification.

Callbacks for Parameter Access

You can register four callback functions to control remote FMS Read/Write access to function block, transducer block, or resource block parameters. The callback functions have the following parameters: the handle of the block to which the accessed parameter belongs, the offset of the accessed parameter within the block, and the subindex of the parameter. The subindex is only meaningful if the parameter is a record or an array; the subindex can be ignored if the parameter is a simple variable. In addition, each callback function has other function-specific parameters and defines a set of return values expected by the Function Block Shell.

- **CB_READ**: This callback function is called when the Function Block Shell receives a remote FMS Read request for parameters with the ownership type **USER_ALONE**. Depending on the return value of the callback function, the Function Block Shell responds positively or negatively to the read request. You can construct the FMS user data packet, or give the Function Block Shell a pointer to the data so it can form the FMS user data portion of the packet.
- **CB_WRITE**: This callback function is called when the Function Block Shell receives a remote FMS Write request for parameters with the ownership type **USER_ALONE**. Depending on the return value of the callback function, the Function Block Shell responds positively or negatively to the write request. You can decode the FMS user data and modify the parameter, or give the Function Block Shell a pointer to the data so it can decode the FMS user data and modify the parameter.
- **CB_NOTIFY_WRITE**: This callback function is called when the Function Block Shell receives a remote FMS Write request for parameters with ownership type **SHELL_NOTIFY** or **USER_PTR**. Depending on the return value of the callback function, the Function Block Shell responds positively or negatively to the write request. The parameter value is updated only on a positive response.
- **CB_NOTIFY_READ**: This callback function is called when the Function Block Shell receives a remote FMS Read request for parameters with ownership type **SHELL_NOTIFY** or **USER_PTR**. Depending on the return value of the callback

function, the Function Block Shell responds positively or negatively to the read request.

Callback for Function Block Execution

A callback function, `CB_EXEC`, must be registered within each VFD for executing function blocks. The System Management schedule determines when a function block in the VFD must be executed, at which time the Function Block Shell invokes this callback function. The Function Block Shell gives the handle or descriptor of the block to be executed to the callback function. Before invoking the callback function, the Function Block Shell snaps all the input, or *subscribed* parameters of the function block. To the Function Block Shell, a return from the callback function means the end of execution of the function block. After the end of execution of the Function Block, the Function Block Shell snaps the output, or *published* parameters, and updates the relevant trend objects.

Callbacks for Alert Notification

Two callback functions can be registered to handle alert notifications. `CB_ACK_EVENTNOTIFY` informs you of a successful notification and the receipt of a confirmation, or it informs you of an unsuccessful notification. `CB_ALARM_ACK` informs you of the receipt of an acknowledgment from the remote machine.

Registration Functions

Chapter 3

This chapter describes the registration process and the associated functions.

Registration Process

The registration process has several steps, as follows:

1. Construct an ASCII file of a defined format, called the *device template*, to describe the application.
2. Use the device code-generator utility to convert the device template to a C file.
3. Link and compile the C file with other modules of the application.
4. Call the registration functions in `userstart`.

Constructing the Device Template

The format of the device template is described in this section. See your NI-FBUS Function Block Shell distribution disk for a sample template file.

File Format

The template file contains several sections, each with a keyword followed by several lines of description. The sections should be in the following order:

```
VFD
USER_TYPE
BLOCKS
TRENDS
VARLISTS
```

As in the C++ language, the double slash ("//") is used for comments.

VFD

The keyword `VFD` should be followed by thirteen lines, as follows:

```
vendor name
model name
revision
profile number 1, profile number 2
number of user defined types
number of transducer blocks
number of function blocks
number of maximum linkage objects
number of maximum alert objects
number of maximum trend float objects
number of maximum trend discrete objects
number of maximum trend bitstring objects
number of maximum variable lists
```

User Types

Do not use the keyword `USER_TYPE` if the number of user-defined types is zero. Otherwise, descriptions of user-defined types should be provided after the keyword `USER_TYPE`, and the number of descriptions should match the number of user-defined types specified in the VFD section.

Each user type description has the following format:

```
number of entries (i.e., n)
typeindex, size, offset
... (total of n lines)
typeindex, size, offset
```

For each entry (subfield) of the user-defined type, the index of its type, *typeindex*, must be provided. See Table 3-1 for a list of type indices. The size should also be provided for the string type (octet string, visible string, and bit string). For types with a well-known size, provide a size of 0.

The offset of an entry in a data type is the offset of that entry in the C structure of the data type. The Function Block Shell must know the addresses of the structure members based on the address of the structure itself in order to process the read and write callback functions. Therefore, the offsets of members must be given to the Function Block Shell.

Blocks

The number of blocks (resource blocks, transducer blocks, and function blocks) should match the number specified in the `VFD` section, and the block descriptions should be in this order: first `RESOURCE`, then `TRANSDUCER`, then `FUNCTION` blocks.

Each block description starts with the keyword `BLOCK`, followed by descriptions of the block and each parameter of the block.

The description of the block consists of the following lines:

block tag
block type(RESOURCE, TRANSDUCER, or FUNCTION)
DD name, DD item, DD revision
profile and profile revision, execution time, execution period, next FB
number of parameters

If the block tag is `BLANK_TAG`, the tag of the block is blank.

Each parameter must have one line of description. For the standard parameters defined in the Fieldbus Specification, the format of the description is as follows:

DD name, DD item, standard parameter name, owner

If you have a device description, you may put zero for the *DD name* and *DD item* fields. See the *Using the Code Generation Utility* section later in this chapter for more information.

For non-standard parameters, provide the data type and usage information. For non-array parameters of some types, the initial value is required. The format of the non-standard parameter description is as follows:

DD name, DD item, data meta type, data type, usage, storage, owner, initial value

or:

DD name, DD item, data meta type, data type, (# of elements for meta Type

where *data meta type* can be `SIMPLE`, `RECORD`, or `ARRAY`. *data type*, *usage*, *storage*, *owner*, and *initial value* are described in the following sections.

Data Type

Data types are types standardized in Fieldbus Specifications and user-defined types. Table 3-1 lists the standard types used in the template registration.

Table 3-1. Data Type Names Used in Template Registration

Index	Name	Need Initial Value?
1	Boolean	Yes
2	Int8	Yes
3	Int16	Yes
4	Int32	Yes
5	uint8	Yes
6	uint16	Yes
7	uint32	Yes
8	Float	Yes
9	Visible Str	No
10	Octet Str	No
11	Date	No
12	Time Of Day	No
13	Time Diff	No
14	Bit String	No
21	Time Value	No
32	Block	No
33	VS Float	No
34	VS Discrete	No
35	VS BitString	No
36	Scaling Struct	No
37	Mode Struct	No
38	Access Perm	No
39	Alarm Float	No
40	Alarm Discrete	No
41	Event Update	No

Table 3-1. Data Type Names Used in Template Registration (Continued)

Index	Name	Need Initial Value?
42	Alarm Summary	No
43	Alert Analog	No
44	Alert Discrete	No
45	Alert Update	No
46	Trend Float	No
47	Trend Discrete	No
48	Trend BitString	No
49	Linkage	No
50	Simulate Float	No
51	Simulate Discrete	No
52	Simulate BitString	No
53	Test	No
54	Action	No

For user-defined types, the type name for the *n*th type you define in the device template is as follows:

`USER_TYPEn`

where all data type names are case-sensitive.

Usage

A parameter can be contained, input, or output:

`C`	contained
`IN`	input
`OUT`	output

Storage

The storage of a parameter can be dynamic, nonvolatile, or static:

`D`	dynamic
`N`	nonvolatile
`S`	static

Owner

This attribute tells the Function Block Shell the ownership of the parameter. See the *Application Program Structure* section in Chapter 2, *Functional Overview*, for an explanation of parameter ownership.

`USER_ALONE`	You, the user, own the data.
`USER_PTR`	You own the data, and the Function Block Shell keeps the pointer.
`SHELL_ALONE`	The Function Block Shell owns the data.
`SHELL_NOTIFY`	The Function Block Shell owns the data, and whenever the network asks to read or modify the data, the Function Block Shell asks your permission with a callback function.

Initial Value

Supply initial values for integer and float parameters. `+INF` and `–INF` are for positive and negative infinite values.

Trends

There are three types of trends: float, discrete and bit string. They all follow the same format in the template, and they should be present in the order of float, discrete, and bit string after the keyword `TRENDS`.

The format of a float trend description is as follows:

```
number of float trends(i.e., n)
block number, parameter offset, sample type, sample interval
... (total of n lines)
block number, parameter offset, sample type, sample interval
```

If there is no trend float, the number of float trends should be zero.

The discrete trend and bit string trend have exactly same format as above.

There are two sample types: `INSTANT` and `AVERAGE`. The definition of these sample types can be found in the *Fieldbus Foundation Specification*.

Variable Lists

Following the keyword `VARLISTS` is the number of variable lists to be defined. Each variable list is defined in the following format:

```
VARLIST
```

```
block number, view type, variable list name
number of variables (i.e., n)
parameter offset
... (total of n lines)
parameter offset
```

The block number is the number of the block to which this variable list belongs. The *`view type`* can be `view1`, `view2`, `view3`, or `view4`.

When you define variable lists in the template, remember the following:

- Each block can have only one `view1` and one `view2`, but may have multiples of `view3` and `view4`.
- View lists of a block must be defined contiguously, and must be in the order `view1`, `view2`, `view3`, and `view4`.

Using the Code Generation Utility

The device code generation utility `codegen.exe` is an MS-DOS program. It is distributed as part of the National Instruments Fieldbus Device Interface Kit. It takes two required command line arguments and an optional third argument, as follows:

```
codegen input output [symbol_file]
```

where *`input`* is the file name of device template, and *`output`* is the generated C file. *`symbol_file`* is the name of the symbol file generated by the DD tokenizer when you tokenize your Device Description. If you are using standard blocks, you may use the `nifb.sym` file provided in the Fieldbus Device Interface Kit. The symbol file contains the name of the function block parameters and their *`DD names`* and *`DD items`* in a certain format. When you use this optional third argument for `codegen`, `codegen` searches for parameter names in the symbol file. If a parameter name is found, the `DD name` and `DD item` of the parameter in the symbol file are used in the output file. Otherwise, the `DD name` and `DD item` in the template file are used.

Calling Registration Functions

Call the `shRegisCallback` function in `userStart` to register the callback functions. You must also call `shRegisParamPtr` if you have any parameters of **USER_PTR** ownership.

Registration Functions

`shRegisCallback`	Register callback functions.
`shRegisParamPtr`	Register pointers of parameters with **USER_PTR** ownership.
`shStartExecLoop`	Start periodic execution of a function.

shRegisCallback

Purpose

Register callback functions of a VFD with the Function Block Shell.

Format

```
RETCODE shRegisCallback(
   HDL_VFD               hVfd,
   CB_READ               *cbRead,
   CB_WRITE              *cbWrite,
   CB_NOTIFY_READ        *cbNotifyRead,
   CB_NOTIFY_WRITE       *cbNotifyWrite,
   CB_EXEC               *cbExec,
   CB_ACK_EVENTNOTIFY    *cbAckEventNotify,
   CB_ALARM_ACK          *cbAlarmAck,
   void                  *reservedForFuture)
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `hVfd`	VFD handler.
IN `cbRead`	Callback function for read.
IN `cbWrite`	Callback function for write.
IN `cbNotifyRead`	Callback function for read notify.
IN `cbNotifyWrite`	Callback function for write notify.
IN `cbExec`	Callback function for block execution.
IN `cbAckEventNotify`	Callback function for acknowledgments of event notify.
IN `cbAlarmAck`	Callback function for acknowledgment of alarm
IN `reservedForFuture`	Reserved for future use. Pass a NULL for this parameter.

Return Values

`retcode`

shRegisCallback

Continued

Description

`shRegisCallback` is used to register callback functions of a VFD to the Function Block Shell. After `shRegisCallback` is called, the Function Block Shell is able to invoke these callback functions for various purposes, such as to create a block algorithm or to read and write parameters.

The pointers to the various callback routines detailed in Chapter 4, *Callback Functions*, are passed as input parameters. If a certain callback is not supported, you must pass the value NULL.

Possible Errors

`E_INVALID_VFD_HANDLE`	The VFD handle is not valid.
`E_CB_RW_NULL`	Read or write callback functions should not be NULL, because there are parameters with **USER_ALONE** ownership.
`E_CB_R_NOTIFY_NULL`	Read notify callback function should not be NULL, because there are parameters with **SHELL_NOTIFY** or **USER_PTR** ownership.
`E_CB_W_NOTIFY_NULL`	Write notify callback function should not be NULL, because there are parameters with **SHELL_NOTIFY** or **USER_PTR** ownership.

shRegisParamPtr

Purpose

Register pointers of parameters with **USER_PTR** ownership.

Format

```
RETCODE shRegisParamPtr(
   HDL_VFD        hVfd,
   HDL_BLOCK      hBlock,
   uint16         numParam,
   PARAM_PTR      paramPtr[]);
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `hVfd`	Handle of the VFD to which these parameters belong.
IN `hBlock`	Handle of the block to which these parameters belong.
IN `numParam`	Number of parameters with **USER_PTR** ownership in this block.
IN `paramPtr`	Offset and pointers of the parameters. This array should have the length `numParam`.

```
typedef struct PARAM_PTR{
   uint16         offset;
   void           *ptr;
}PARAM_PTR;
```

Return Values

`retcode`

Description

`shRegisParamPtr` is used to register the pointers of parameters with ownership **USER_PTR** on a per block basis. For example, if there are *n* function blocks in the application, and each of them has parameters with **USER_PTR** ownership, then this function is used *n* times.

The memory location of parameters with **USER_PTR** ownership should not change in the entire application. Otherwise, the Function Block Shell might read and write to an illegal

shRegisParamPtr

Continued

memory location and crash the application. The data of the parameters with **USER_PTR** ownership must be stored as global variables or in allocated memory that is never freed.

The dynamic registration of parameter pointers is not supported. This function can be called only before the `shInitShell` function is called. It cannot be called in any of the callback functions.

Possible Errors

`E_INVALID_VFD_HANDLE`	The VFD handle is invalid.
`E_INVALID_BLOCK_HANDLE`	The block handle is invalid.
`E_NUM_PARAM_MISMATCH`	The number of parameters is not equal to the number of parameters with **USER_PTR** ownership in this block.
`E_INVALID_PARAM_OFFSET`	There is an invalid parameter offset in `paramPtr`.
`E_PARAM_TYPE_MISMATCH`	There is a parameter with ownership that is not **USER_PTR** in `paramPtr`.

shStartExecLoop

Purpose

Start a function that executes periodically.

Format

```
RETCODE shStartExecLoop(
    LOOP_EXEC       func,
    uint16          period);
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `func`	A function to be executed periodically.
IN `period`	The length of the period in milliseconds.

Return Values

`retcode`

Description

Some applications need to run certain functions periodically. For example, a transducer block might need to have an algorithm that runs periodically, or a simulation might need to generate a random number periodically. Because these functions are not part of the function block, they cannot be invoked by the `cbExec` function. The `shStartExecLoop` function provides a mechanism for such functionality. It calls the specified function at the specified rate. After this function is called, there is no way to stop the execution of the specified function.

☞ **Note:** ***If a function runs too frequently, it might consume too much processor time. Therefore, a minimum period of 20 ms is enforced. If you call the function with a period of less than 20 ms, 20 ms is used as the period. Also, the periodic function should generally run very fast. A slow periodic function with a small period will affect the schedule of the function block execution, possibly causing stale data in communication between devices.***

The `shStartExecLoop` ***function can be called no more than five times.***

shStartExecLoop

Continued

Possible Errors

`E_NO_MEMORY`	There is not enough memory to start the function.

Chapter 4

Callback Functions

This chapter describes the callback functions of the Function Block Shell.

Callback Functions

`CB_EXEC`	Function Block execution callback function.
`CB_NOTIFY_READ`	The Function Block Shell obtains permission to read parameters with ownership type **SHELL_NOTIFY** or **USER_PTR**.
`CB_NOTIFY_WRITE`	The Function Block Shell obtains permission to write parameters with ownership type **SHELL_NOTIFY** or **USER_PTR**.
`CB_READ`	Read **USER_ALONE** parameters when responding to a remote read request.
`CB_WRITE`	Write to **USER_ALONE** parameters when responding to a remote write request.

The alarm-related callback functions `CB_ALARM_ACK` and `CB_ACK_EVENTNOTIFY` are described in Chapter 6, *Alarm Functions*.

CB_EXEC

Purpose

The callback function for function block execution.

Definition

```
typedef void
(CB_EXEC(HDL_BLOCK     hBlock));
```

Parameters

IN `hBlock`	Block handle.

Return Values

`retcode`

Description

This function is called when a function block is scheduled to run. Before calling this function, the Function Block Shell updates the status and value of all the input parameters. If there is no connection to an input parameter, the Function Block Shell sets the quality of the status of this parameter to BAD, the sub-status to NOT CONNECTED, and the limit to NOT LIMITED. If no updated value is received from the network for an input parameter, then the quality of the status is set to BAD, the sub-status to either NO COMMUNICATION WITH LAST USABLE VALUE or NO COMMUNICATION WITH NO USABLE VALUE, and the limit is set to NOT LIMITED. Otherwise, the status is what is received from the network connection for that input.

After calling this function, the Function Block Shell sends the output parameter values for this block to the Fieldbus, and then updates the trends if there are any parameters in this block that need trends.

While this function is executing, remote read and write requests of the parameters of this function block are not permitted. Therefore, the integrity of the data throughout the function block execution period is guaranteed.

To the Function Block Shell, the return of this function means that the function block has finished executing. Therefore, it is your responsibility to ensure that this function returns within the maximum execution time that you specified for it in the function block registration.

You should examine the alarm conditions in this function. If an alarm occurs, you should call the `shAlertNotify` function, so the Function Block Shell can report the alarm.

CB_NOTIFY_READ

Purpose

Obtains the user's permission to read parameters with ownership type **SHELL_NOTIFY** or **USER_PTR**.

Definition

```
typedef RETCODE
(CB_NOTIFY_READ(HDL_BLOCK    hBlock,
                uint16       offset,
                uint16       subindex,
                Bool_t       byShell));
```

Parameters

IN `hBlock`	Block handle.
IN `offset`	Offset of the parameter in the block. (The first parameter is offset 1.)
IN `subindex`	1-relative subindex within the parameter (for records or arrays).
IN `byShell`	Reason for the Function Block Shell to call this function.

Return Values

`retcode`

Description

When responding to a remote read request of a parameter with ownership type **SHELL_NOTIFY** or **USER_PTR**, the Function Block Shell invokes this function to get your permission to read the data. You should return `R_SUCCESS` if the request is allowed, and the FB service error reason code otherwise. In the latter case, the Function Block Shell returns a negative response, with the reason code, to the remote read request.

CB_NOTIFY_WRITE

Purpose

Obtain the user's permission to write parameters with ownership of types **SHELL_NOTIFY** or **USER_PTR**.

Definition

```
typedef RETCODE
(CB_NOTIFY_WRITE(HDL_BLOCK      hBlock,
                 uint16         offset,
                 uint16         subindex,
                 bool_t         byShell,
                 void           *data));
```

Parameters

IN `hBlock`	Block handle.
IN `offset`	Offset of the parameter in the block. (The first parameter is offset 1.)
IN `subindex`	1-relative subindex within the parameter (for records or arrays).
IN `byShell`	Reason for Function Block Shell to call this function.
IN `data`	New data for the parameter.

Return Values

`retcode`

Description

When responding to a remote write request of a parameter with ownership type **SHELL_NOTIFY** or **USER_PTR**, the Function Block Shell invokes this function to get your permission to modify the data. You can check to see if the new data is valid to be written to the parameter. You should return `R_SUCCESS` if the request is allowed. In this case, the Function Block Shell updates the parameter to the new value in `data`. You can also return `R_USER_DONE` if you prefer to update the data for **USER_PTR** parameters yourself. In this case, the Function Block Shell responds positively to the remote write request without updating the parameter. If you return an FB service error reason code, the Function Block Shell responds negatively to the remote write request without updating the parameter.

CB_READ

Purpose

Function Block Shell reads **USER_ALONE** parameters when responding to remote read requests.

Definition

```
typedef RET CODE
(CB_READ(HDL_BLOCK      hBlock,
         uint16         offset,
         uint16         subindex,
         bool_t         byShell,
         void*          buf,
         uint8*         bufLen,
         void**         paramPtr));
```

Parameters

IN `hBlock`	Block handle.
IN `offset`	Offset of the parameter in the block.
IN `subindex`	Subindex within the parameter.
IN `byShell`	Reason for the Function Block Shell to call this function.
IN `buf`	Data buffer to be filled.
IN/OUT `bufLen`	Length of the data buffer.
OUT `paramPtr`	Pointer to the parameter data.

Return Values

`retcode`

Description

The Function Block Shell calls this function to read **USER_ALONE** parameters. The block handle and offset identify the parameter or one of its members. If the subindex is zero, the Function Block Shell is reading the whole parameter. Otherwise, the Function Block Shell is reading the member of the parameter specified in the subindex. The Function Block Shell checks the validity of the block handle, offset, and subindex when a network read request comes, so you do not need to check these parameters in a callback function. This also applies to the callback functions `CB_NOTIFY_READ`, `CB_WRITE`, and `CB_NOTIFY_WRITE`.

CB_READ

Continued

The Function Block Shell needs to access parameters for two reasons:

- To service read or write requests from the network.
- To access the parameters for internal use. For example, the Function Block Shell updates the input parameters of a block before the block begins executing. Also, the Function Block Shell needs to read parameters such as **ALERT_KEY** to create an alert during the `shAlertNotify()` function.

The `byShell` parameter specifies whether the Function Block Shell is calling your callback for internal reasons (`byShell` = TRUE) or to service Fieldbus network read and write requests (`byShell` = FALSE). You might give different permissions in each case. National Instruments recommends that you always allow the Function Block Shell to access the parameters for its internal use, because if you do not, the Function Block Shell might not operate properly.

The `byShell` parameter is also present in the callback functions `CB_NOTIFY_READ`, `CB_WRITE`, and `CB_NOTIFY_WRITE`, and has the same meaning as in `CB_READ`. For example, the Function Block Shell calls the `CB_WRITE` function to modify the input parameter before function block execution. Therefore, the `byShell` parameter would be TRUE. In this case, you should grant the permission to perform the write. The Function Block Shell might also call the `CB_WRITE` function to modify an input parameter upon receiving a network write request. `byShell` would be FALSE in this case, and you might want to refuse this type of request for your own reasons.

When using this function, you have three options:

- You encode data in FMS format, store it in `buf`, set the `bufLen`, and then return `R_SUCCESS`. In this case, the Function Block Shell assumes that `buf` points to the buffer containing the correct data.
- Instead of encoding the data, you pass a pointer to the parameter in `paramPtr`, and return `R_DELEGATE`. The Function Block Shell then encodes the data to FMS format. If the parameter is a record or array, even if a read request is on a subindex, you should still pass the pointer to the whole record or array instead of the pointer to the element of the record or array. The Function Block Shell handles locating the element.
- Return the FB service error reason code for refusing service if reading is not permitted. In this case, the Function Block Shell returns a negative response, with the reason code, to the remote read request instead of encoding the data. The

CB_READ

Continued

reason codes are E_PARAM_CHECK, E_EXCEED_LIM, E_WRONG_MODE, E_WRITE_PROHIBITED, and E_DATA_NOT_WRITABLE. See the *Function Block Application Process, Part 1* for the meaning of these reason codes. These codes will be returned to the requesting device across the Fieldbus if this was a network request.

CB_WRITE

Purpose

Function Block Shell writes to **USER_ALONE** parameters.

Definition

```
typedef RETCODE
(CB_WRITE(HDL_BLOCK    hBlock,
          uint16       offset,
          uint16       subindex,
          bool_t       byShell,
          void*        data,
          void**       paramPtr);
```

Parameters

IN `hBlock`	Block handle.
IN `offset`	Offset of the parameter in the block (First parameter starts at 1).
IN `subindex`	Subindex within the parameter (for records and arrays).
IN `byShell`	Reason for the Function Block Shell to call this function.
IN `data`	New data for the parameter.
OUT `paramPtr`	Pointer to the parameter data.

Return Values

`retcode`

Description

The Function Block Shell calls this function to modify the parameter you own. `data` points to new data. The data type of the new data depends on the data type of the parameter and the subindex parameter. For example, if the write request is on a parameter of type `FF_VsFloat`, which is a record, *`data`* points to a record of `FF_VsFloat` if the subindex is 0, or to a float if the subindex is 2.

When using this function, you have three options:

- Modify the parameter with the new data, and then return `R_SUCCESS`.

CB_WRITE

Continued

- Instead of copying the data, pass the pointer to the parameter in `paramPtr` and return `R_DELEGATE`. The Function Block Shell then modifies the parameter. Just as in `CB_READ`, even if the parameter is a record or array and the request is on a subindex, the pointer you pass to the Function Block Shell should still point to the whole record or array, and not to a specific element of a record or an array.

  If the Function Block Shell is passing the user a record or array element, the `data` parameter points to the actual member, *not* to the entire record or array. However, when you pass the Function Block Shell a pointer, the pointer should always be to the entire record or array.

  For example, suppose a device on the Fieldbus requests a write to subindex 2 of a Function Block Shell-owned parameter of type `FF_VsFloat` (which is a record). The `buf` parameter that the Function Block Shell passes in would point to a floating point number instead of to the `FF_VsFloat` record. However, if you own a parameter, and you want the Function Block Shell to copy the data, you should return `R_DELEGATE` and a pointer to the entire `FF_VsFloat` data structure.
- Return the FB service error reason code to reject the service. In this case, the Function Block Shell returns a negative response, with the reason code, to the remote write request instead of decoding the data.
- You can modify the contents of `data`, then return `R_DELEGATE` to allow the write to succeed with data that you supplied. You must also supply the pointer to the parameter in the `paramPtr` argument in this case.

Chapter 5

Utility Functions

This chapter describes the utility functions of the Function Block Shell.

Utility Functions

`shGetTime`	Get the application time.
`shSignalBlockSem`	Release the semaphore of a function block.
`shReadParam`	Read a shell-owned parameter or object.
`shWaitBlockSem`	Acquire the semaphore of a function block.
`shWriteParam`	Write a shell-owned parameter or object.
`shWriteNVM`	Save the non-volatile parameters whose ownership is **USER_ALONE** and **USER_PTR** in non-volatile memory.

shGetTime

Purpose

Get the application time maintained by the System Management of the Stack.

Format

```
FF_Time
shGetTime();
```

Includes

```
#include "fbsh.h"
```

Return

```
FF_Time

typedef struct FF_TIME{
    uint32          upper;
    uint32          lower;
}FF_Time;
```

Description

You can use this function to get the application time maintained by System Management. The application time is the number of 1/32 ms periods that have passed since January 1, 1972.

The application time on a Fieldbus device comes from the Time Master that the device connects to. If the time in the Time Master is not set correctly, the application time is not the number of 1/32 ms periods that have passed since January 1, 1972. However, even if the Time Master is not set correctly, you can still use this function to measure time elapsed in your application. For example, you can use this function to measure how long a function block execution takes.

shSignalBlockSem

Purpose

Release the semaphore of a function block.

Format

```
RETCODE
shSignalBlockSem(
   HDL_VFD         hVfd,
   HDL_BLOCK       hBlock);
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `hVfd`	VFD handle.
IN `hBlock`	Block handle.

Return Values

`retcode`

Description

This function is used to release the semaphore of a function block. The VFD handle and block handle together identify the block. To ensure that the user and the Function Block Shell do not access a parameter at the same time, `shWaitBlockSem` must be used before accessing any parameters with **USER_PTR** ownership to ensure mutual exclusion, and `shSignalBlockSem` must be called after accessing those parameters to allow the Function Block Shell to access the parameters.

Possible Errors

`E_INVALID_VFD_HANDLE`	The VFD handle is invalid.
`E_INVALID_BLOCK_HANDLE`	The block handle is invalid.

shReadParam

Purpose

Enables you to read parameter data owned by the Function Block Shell.

Format

```
RETCODE
shReadParam(
      HDL_VFD         hVfd,
      HDL_BLOCK       hBlock,
      uint16          offset,
      uint16          subindex,
      void            *data,
      uint16          datalen);
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `hVfd`	VFD handle.
IN `hBlock`	Block handle. Valid range is one to the number of blocks in the VFD.
IN `offset`	Offset of the parameter in the block. Valid range is zero to the number of parameters in the block. When the offset is zero, you are reading the block itself.
IN `subindex`	Subindex within the parameter. Valid range is one to the number of members in the parameter, or zero (see Description).
IN `data`	Data buffer to be filled.
IN `dataLen`	Length of the data buffer.

Return Values

`retcode`

Description

When the Function Block Shell owns the data of the parameter, you can call this function to read the data. The VFD handle, block handle, and offset identify the parameter. If the subindex is 0, you are reading the whole parameter. Otherwise, you are reading the member of the parameter specified by the subindex.

shReadParam

Continued

You should know the data type of the parameter or parameter component you want to access. `data` should be a pointer to that data type, and `dataLen` should be the size. `dataLen` is used mainly to ensure you have allocated enough space for the Function Block Shell to write.

Possible Errors

`E_INVALID_VFD_HANDLE`	The VFD handle is invalid.
`E_INVALID_BLOCK_HANDLE`	The block handle is invalid.
`E_INVALID_OFFSET`	The parameter offset is invalid.
`E_USER_DATA`	The data is owned by you, not the Function Block Shell.
`E_INVALID_SUBINDEX`	The subindex is out of range for this parameter.
`E_BUFFER_TOO_SMALL`	The buffer is too small to write the data of this parameter.

shWaitBlockSem

Purpose

Acquire the semaphore of a function block.

Format

```
RETCODE
shWaitBlockSem(
   HDL_VFD          hVfd,
   HDL_BLOCK        hBlock);
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `hVfd`	VFD handle.
IN `hBlock`	Block handle.

Return Values

`retcode`

Description

This function is used to acquire the semaphore of a function block. One semaphore is created for each function block in the Function Block Shell. For a parameter with **USER_PTR** ownership, the Function Block Shell keeps the pointer to a parameter, and may access it any time on a remote read or write request. To ensure that you and the Function Block Shell do not access the parameter at the same time, `shWaitBlockSem` must be used before accessing any parameters with **USER_PTR** ownership to ensure mutual exclusion.

The VFD handle and block handle together identify the block.

Possible Errors

`E_INVALID_VFD_HANDLE`	The VFD handle is invalid.
`E_INVALID_BLOCK_HANDLE`	The block handle is invalid.

shWriteParam

Purpose

Write to parameter data owned by the Function Block Shell.

Format

```
RETCODE
shWriteParam(
   HDL_VFD          hVfd,
   HDL_BLOCK        hBlock,
   uint16           offset,
   uint16           subindex,
   void             *data,
   uint16           datalen);
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `hVfd`	VFD handle.
IN `hBlock`	Block handle. Valid range is one to the number of blocks in the VFD.
IN `offset`	Offset of the parameter in the block. Valid range is zero to the number of parameters in the block. When the offset is zero, you are reading the block object itself. Only members 8 and 10 of the block object are writable.
IN `subindex`	Subindex within the parameter. Valid range is one to the number of members in the parameter.
IN `data`	Data to write.
IN `dataLen`	Length of the data buffer.

Return Values

`retcode`

Description

When the Function Block Shell owns the data of the parameter, you can call this function to modify the data. The VFD handle, block handle, and offset identify the parameter. If the subindex is 0, you are reading the whole parameter. Otherwise, you are reading the member of the parameter specified by the subindex.

shWriteParam

Continued

You should know the data type of the parameter or parameter component you want to access. `data` should be a pointer to that data type, and `dataLen` should be the size. `dataLen` is used mainly to ensure that `data` points to the correct amount of data.

Possible Errors

E_INVALID_VFD_HANDLE	The VFD handle is invalid.
E_INVALID_BLOCK_HANDLE	The block handle is invalid.
E_INVALID_OFFSET	The parameter offset is invalid.
E_USER_DATA	The data is owned by you, not the Function Block Shell.
E_INVALID_SUBINDEX	The subindex is out of range for this parameter.

shWriteNVM

Purpose

Save the non-volatile parameters whose ownership is **USER_ALONE** and **USER_PTR**, in non-volatile memory.

Format

```
RETCODE
shWriteNVM(
   HDL_VFD        hVfd,
   HDL_BLOCK      hBlock,
   uint16         offset,
   void           *data);
```

Includes

```
#include "fbsh.h"
```

Parameters

IN `hVfd`	VFD handle.
IN `hBlock`	Block handle of the parameter.
IN `offset`	Offset of the trend object to be configured.
IN `data`	Data to write.

Return Values

`retcode`

Description

This function saves non-volatile parameters that have **USER_ALONE** and **USER_PTR** ownership in non-volatile memory. Every time you change the values of such parameters in your function block application program, you need to use this function to store the new value in non-volatile memory.

These parameters can also be changed by a network FMS write. The Function Block Shell automatically handles the non-volatility in this case. Therefore, you do not need to use `shWriteNVM` in your write callback function to store these parameters in non-volatile memory.

`hVfd`, `hBlock`, and `offset` together identify which parameter is to be put in non-volatile memory. `data` points to the parameter to be stored in non-volatile memory

shWriteNVM

Continued

Possible Errors

E_INVALID_VFD_HANDLE	The VFD handle is invalid.
E_INVALID_BLOCK_HANDLE	The block handle is invalid.
E_INVALID_OFFSET	The offset of the trend object is invalid.
E_SHELL_DATA	The Function Block Shell owns the data, so the data does not have to be stored in non-volatile memory with this function.

Chapter 6

Alarm Functions

This chapter describes the alarm functions of the Function Block Shell.

Alarm Functions

`CB_ALARM_ACK`	The Function Block Shell notifies you of an alarm acknowledgment.
`CB_ACK_EVENTNOTIFY`	The Function Block Shell notifies you of an alert object transmission confirmation.
`shAlertNotify`	Create an alert object and wait for the acknowledgment.
`shClearAlert`	Clear an alert object.

CB_ALARM_ACK

Purpose

Callback function for the Function Block Shell to notify you of the acknowledgment of an alarm.

Definition

```
typedef RETCODE
(CB_ALARM_ACK(HDL_BLOCK      hBlock,
              uint16         offset));
```

Parameters

IN `hBlock`	Block handle.
IN `offset`	Offset of the alarm parameter.

Description

When it receives the acknowledgment of the alarm, the Function Block Shell invokes this callback function to inform you. `hBlock` and `offset` identify the `alarm` parameter, and the return code indicates the result. In this callback function, you should update the `unAck` attribute of the alarm, and return `R_SUCCESS` afterwards. If the alarm has already been acknowledged, you should return `E_ALARM_ALREADY_ACKED`.

CB_ACK_EVENTNOTIFY

Purpose

Callback function for the Function Block Shell to notify you of the confirmation of an alert object transmission.

Definition

```
typedef void
(CB_ACK_EVENTNOTIFY(HDL_BLOCK   hBlock,
                    uint16      offset,
                    RETCODE     status ));
```

Parameters

IN `hBlock`	Block handle.
IN `offset`	Offset of the alarm parameter.
IN `status`	Status of the event acknowledgment.

Description

`hBlock` and `offset` identify the alarm parameter, and `status` indicates the result.

This callback function is called by the Function Block Shell in three cases:

- The Function Block Shell fails to send the alert object to the network. If there is no open connection for sending alarms, this function is called with the `status` `E_NO_OPEN_ALARM_LINK`. If there is a communication layer failure, `status` is `E_COMM_FAILURE`.
- The Function Block Shell has sent the alert object `MAX_ALT_RESEND_TIMES`, and still no confirmation has been received. The `status` in this case is `E_ALT_SENT_TIMES_OVERFLOW`.
- The Function Block Shell sent the alert object and received the acknowledgment. The `status` is `R_SUCCESS` in this case.

shAlertNotify

Purpose

Create an alert object to be sent to the network and wait for the acknowledgment.

Format

```
RETCODE
shAlertNotify(
   HDL_VFD          hVfd,
   HDL_BLOCK        hBlock,
   uint16           offset,
   uint8            mfgrType,
   uint8            stdType,
   uint8            mesgType,
   uint8            prio;
   uint16           unitIndex)
```

Includes

```
#include "fbsh.h"
```

Parameter

IN `hVfd`	Handle of VFD.
IN `hBlock`	Block handle.
IN `offset`	Offset of the alarm parameter.
IN `mfgrType`	Manufacturer type of the alarm.
IN `stdType`	Standard type of the alarm.
IN `mesgType`	Message type.
IN `prio`	Priority of the alert.
IN `unitIndex`	Unit index of the alarm parameter.

Return Values

`RETCODE`

Description

You should call this function to notify the shell that your block has detected an alarm condition. In this function, the Function Block Shell finds out if the parameter is a valid alarm parameter first. If it is a valid alarm parameter, the Function Block Shell reads the data of this alarm parameter, creates an alert object, and inserts the alert object in the alert-sending list. The alert objects in the alert-sending list are sent to the network when there are no time-critical tasks running.

shAlertNotify

Continued

Possible Errors

`E_INVALID_VFD_HANDLE`	The VFD handle is invalid.
`E_ALT_FULL`	All alert objects are in use.
`E_INVALID_BLOCK_HANDLE`	Invalid block handle.
`E_INVALID_OFFSET`	Invalid offset for this block.
`E_PARAM_IS_NOT_ALM`	The parameter is not an alarm parameter.
`E_NO_MEMORY`	Out of memory.
`E_CANNOT_GET_ALARM_DATA`	Failed to read the alarm parameter data.

shClearAlert

Purpose

Clear the alert object associated with the alarm parameter.

Format

```
RETCODE
shClearAlert(
    HDL_VFD          hVfd,
    HDL_BLOCK        hBlock,
    uint16           offset);
```

Includes

```
#include "fbsh.h"
```

Parameter

IN `hVfd`	Handle of VFD.
IN `hBlock`	Block handle.
IN `offset`	Offset of the alarm parameter.

Return Values

`RETCODE`

Description

This function is used to cancel the alert object created by `shAlertNotify`. When an alert object is created, it sends alert data several times, and waits for confirmation. If you no longer need to send out the alert after calling `shAlertNotify`, you can use this function to clear the alert object.

Possible Errors

`E_INVALID_VFD_HANDLE`	VFD handle is invalid.
`E_INVALID_BLOCK_HANDLE`	Invalid block handle.
`E_INVALID_OFFSET`	Invalid offset for this block.
`E_NO_ALERT`	No alert object is associated with this alarm.

Miscellaneous Functions

Chapter 7

This chapter describes miscellaneous functions of the Function Block Shell.

Miscellaneous Functions

`shInitShell`	The Function Block Shell initializes its data structures and communication layer.
`userStart`	The starting point of user applications.

shInitShell

Purpose

The Function Block Shell initializes its data structures and communication layer.

Definition

```
RETCODE
shInitShell(bool_t       *firstTime)
```

Includes

```
#include "fbsh.h"
```

Return Values

`retcode`

Description

The Function Block Shell initializes its data structures and communication layer, and starts the alert-processing tasks before the operation loop.

When the `shInitShell` function is called for the first time, the Function Block Shell sets `firstTime` to TRUE. The Function Block Shell saves non-volatile parameters in non-volatile memory. If the program is restarted, and `shInitShell` is called again, then the Function Block Shell sets `firstTime` to FALSE, and loads the values of non-volatile parameters from non-volatile memory.

In some cases, this function never returns, because the communication layer is not initialized. This typically indicates one of the following problems:

- The communication layer is not in the running state. For example, the application program running on the Function Block Shell is not physically connected to a link master, so the communication layer cannot be started.
- There is a resource shortage, such as low memory. In this case, the communication layer could not be initialized.
- You should call this function from your `userStart` routine.

Possible Errors

`E_COMM_FAILURE`	The Function Block Shell fails to initialize the communication layer.
`E_ALT_TASK_FAILURE`	Alert-processing tasks cannot be started.

shInitShell

Continued

`E_CANNOT_CREATE_SEM`	The Function Block Shell cannot create a semaphore for function blocks.
`E_NO_MEMORY`	Out of memory.
`E_NVM_FAILURE`	Cannot initialize non-volatile memory.

userStart

Purpose

The starting point of user applications.

Definition

```
void userStart()
```

Description

In your applications, you should use the `userStart` routine as the starting point of the application instead of the `main` routine. The `userStart` routine, which you must write, is called only once by the Function Block Shell after the kernel boots up. Your application must define this function. In this function, you must register callback functions, initialize the Function Block Shell by calling `shInitShell`, and initialize the serial driver. You can also perform application-specific initializations here.

Chapter 8

Serial Functions

This chapter describes serial functions of the Function Block Shell.

Serial Functions

`nihOpenDevice`	Opens and initializes a user-configured device descriptor.
`nihCloseDevice`	Closes a previously-opened device.
`nihDefineSequence`	Starts the definition of a new command sequence to be sent to the device.
`nihSendCommand`	Sends a command and optionally waits for a response. Adds a command to a defined sequence.
`nihGetData`	Retrieves the latest reply data to a command that is part of a sequence.
`nihPutData`	Updates the data associated with a command that is part of a sequence.
`nihCancelSequence`	Cancels a previously defined sequence.
`nihSetParam`	Configures certain communication parameters for the network or device.

Overview of Serial Functions

The serial functions provide a general-purpose method of moving data between the Round Card and a serial device. The serial functions are intended to be independent of the communication protocol used to communicate to the serial device. The serial functions are also intended to be independent of the Function Block Shell function calls.

Generic Serial

You can transmit data between the Round Card and the serial device using a generic master/slave command/response serial protocol. Command packets are transmitted to the serial device and, optionally, the serial device can respond to the command packets if needed. When you use the generic serial protocol, you are responsible for encoding and decoding the entire serial packet for the commands and responses.

Hart Serial

Optionally, the serial functions support the HART protocol across a serial line. When you enable the HART protocol, the serial driver is able to transmit HART commands using the message format and timing specifics given by the *HART Field Communications Protocol Specification*. To send HART commands you need only provide the HART command number and any associated command data. The response data and the transmission status returns to you. If the HART command number is a transmitter-specific HART command, you also need to provide the size of the command data and the size of the response data. The serial functions take care of encoding and decoding the entire HART data-link protocol. If a HART command fails, the serial driver resends the command a configured number of times. If the communication failure persists, the device is initialized through the HART `cmd #0` command. The serial driver issues the HART initialization command when the device is initialized (through `nihOpenDevice` function) and it returns the device identification information to you.

Defining Repeated Command Sequences

The serial functions can be configured to continuously send/receive a defined sequence of commands. The sequence of commands is defined by the `nihDefineSequence` and `nihSendCommand` functions. The `nihDefineSequence` function defines the number of commands in the sequence to send to the device. The `nihSendCommand` function is then used to define the commands in the sequence. A *command* is a serial transaction that involves a transmit, a receive, or both. When the last command in a sequence has been defined, the sequence sends all defined commands in the order the commands were defined, and waits for responses on each command that included a receive. The sequence will then run continuously in the background until the `nihCancelSequence` or `nihCloseDevice` function call is made. Functions `nihPutData` and `nihGetData` are provided to update the command data and to retrieve the latest reply data of a command in a command sequence.

You can call `nihPutData` to place the current value of some variable data in the transmit buffer to be sent out when the sequence next executed. Similarly, you can call `nihGetData` to retrieve the data from a packet that was received from the serial device on the last time the sequence was executed.

Sequences are *free-running*; that is, their execution is not tied to the execution of your Function Block Application. However, the serial functions address synchronization issues for you; you will not receive any partially updated data buffers when using `nihGetData` or `nihPutData`.

nihOpenDevice

Purpose

Open and initialize a user-configured device descriptor.

Format

```
nihDesc_t nihOpenDevice(nihDesc_t busno, serialAddr_t serialAddr,
                        hartDeviceInfo *hartDevInfo, uint16 *stat)
```

Includes

```
#include "types.h"

#include "hart.h"
```

Parameters

IN `busno`	The numeric identifier of the serial bus in which this device is connected. The serial buses are numbered starting with zero. Currently, only bus 0 is supported.
IN `serialAddr`	For the `SERIAL_HART` protocol, this is the HART address of the device. For the `SERIAL_GENERIC` protocol, this represents a logical address used solely by the serial driver, and is not used in any serial communications.
OUT `hartDevInfo`	The address of the HART device information structure into which the device identification information is returned. It is `NULL` if you are using the `SERIAL_GENERIC` protocol.
OUT `stat`	A HART error code is returned detailing the error, if any. It is `NULL` if you are using the `SERIAL_GENERIC` protocol.

Return Values

A unique identifier used to identify the device or, if the call is unsuccessful, the error code. Following is an example of a unique identifier that might be returned by this call:

```
struct hartDevInfo {
uint8   manuf ;         /* manufacturer's */
                        /*identification code */
uint8   devType ;       /* mfr's device type */
uint8   numPreambles ;
```

nihOpenDevice

Continued

```
uint8   univCmdRev ;
uint8   swRev ;
uint8   hwRev ;
uint8   devFunction ;    /* device function flags */
uint32  devID ;          /* device id number */
} ;
```

Description

If you are using the `SERIAL_HART` protocol, the HART Read Unique Identifier command (`command #0`) is sent to the device at address `serialAddr`. If a response is received, the received device identification information is returned in `devInfo`, and a valid descriptor to the device is returned. On detection of a communication error, the HART Read Unique Identifier command is automatically sent to the device at address `serialAddr`.

If you are using the `SERIAL_GENERIC` protocol, the serial bus at `busno` is initialized with the current communication parameters for `busno` and a valid descriptor to the device is returned. If you need several descriptors for the `SERIAL_GENERIC` protocol, you should supply a different `serialAddr` to `nihOpenDevice`.

You must use the descriptor returned by this function for subsequent commands to the device. A set of default communication parameters is associated with busno. If the communication parameters that you need are different than the default parameters, you must call `nihSetParam` to change the needed parameters of the bus *before* calling `nihOpenDevice` to open the device. The descriptor is valid until the device is closed by a `nihCloseDevice` call. The serial driver does not allow the device at `serialAddr` to be opened multiple times.

If an error occurred, a negative error code is returned and the device remains unopened.

Possible Errors

`E_WRONG_ARGUMENT`	The busno entered is invalid, or `stat` is NULL.
`E_ALREADY_OPEN`	The device at address `serialAddr` is already open.
`E_INTERNAL_ERROR`	Any internal error, such as insufficient resources
`E_COMM_ERROR`	Communication failed.

nihCloseDevice

Purpose

Closes a previously-opened device.

Format

```
int16 nihCloseDevice(nihDesc_t desc)
```

Includes

```
#include "types.h"
#include "hart.h"
```

Parameters

IN `desc`	Descriptor of the device to be closed.

Description

This function terminates all valid command sequences to the device `desc`. Communication to or from the device using `desc` is no longer possible.

Possible Errors

`E_WRONG_ARGUMENT`	The device descriptor is invalid.

nihDefineSequence

Purpose

Start the definition of a new command sequence to be sent to the device.

Format

```
nihDesc_t nihDefineSequence(nihDesc_t desc, uint8 numCmds)
```

Parameters

IN `desc`	A descriptor to an opened device.
IN `numCmds`	The number of commands in this sequence. The maximum is 255.

Return Values

The identifier of the created sequence.

Description

A new command sequence is defined for the device `desc`. After `numCmds` number of commands are added through the function `nihSendCommand`, the sequence is executed. The order in which the commands are executed is the order in which they are defined.

The sequence is not executed until `numCmds` number of commands are added. Once started, the sequence is executed until the sequence is deleted or the device is closed. If any of the commands in the sequence is not successfully executed and the sequence is started over. If you are using HART protocol, the device will be initialized with HART command #0 before the sequence restarts.

Possible Errors

`E_WRONG_ARGUMENT`	The descriptor is invalid, or `numcmds` is 0.
`E_INTERNAL_ERROR`	Any internal error, such as insufficient resources.

nihSendCommand

Purpose

Sends a command and optionally waits for a response or adds a command to a defined sequence.

Format

```
nihDesc_t nihSendCommand(nihDesc_t desc, uint8 cmd,
                         uint8 *cmdData, uint8 *rcvData,
                         uint8 cmdDataSize, uint8 rcvDataSize,
                         uint16 *stat)
```

Includes

```
#include "types.h"
#include "hart.h"
```

Parameters

IN `desc`	The device descriptor of the device to which the command should be sent or the descriptor of the sequence to which this command should be added.
IN `cmd`	The HART command number. It is zero if you are using the `SERIAL_GENERIC` protocol.
IN `cmdData`	The address of the buffer containing command data if any, or `NULL` if there is no data.
IN `cmdDataSize`	Specifies the command data size in `cmd`. It is zero if you are using `SERIAL_HART` protocol and `cmd` is not a Transmitter-Specific Hart Command.
IN `rcvDataSize`	Specifies the size of the data to be received for `cmd`. It is zero if you are using `SERIAL_HART` protocol and `cmd` is not a Transmitter-Specific HART Command.
OUT `rcvData`	The address of the buffer into which data received from the device is to be read; or NULL if no data is to be read.
OUT `stat`	The address of buffer to receive the communication status. It is NULL if you are using the `SERIAL_GENERIC` protocol.

Return Values

The descriptor to the command if `desc` is a command sequence; a negative error code otherwise.

nihSendCommand

Continued

Description

Descriptor Describes a Device

If `desc` is the descriptor of a device, and the protocol is `SERIAL_HART`, the command identified by `cmd` and the command data `cmdData`, if any, are sent out the serial bus. If there is an associated reply, the reply data is returned in `rcvData`. The caller must ensure that the buffer is of sufficient length to receive the user data portion of the packet (not the whole packet). The caller is blocked until the entire transaction, with any retries, is completed.

If `desc` is the descriptor of a device, and the device is using the `SERIAL_GENERIC` protocol, `cmdDataSize` bytes from `cmdData` are sent out the serial port. If there is an associated reply, the reply data is returned in `rcvData`. The caller must ensure that the buffer is of sufficient length to receive the entire packet defined by the caller. The caller is blocked until the entire transaction, with any retries, is completed.

Descriptor Describes a Sequence

If the descriptor describes a sequence, the command `cmd` and associated data `cmdData` are added to the sequence. The command is not sent on the serial link at this time. The function returns a descriptor to the command in the sequence. This descriptor can be used in subsequent calls to `nihGetData` to retrieve response data or to `nihPutData` to change the transmitted command. `rcvData` and `stat` must be `NULL`, since these are not needed. If the command stored is the last in the sequence, a new thread of execution is started, which continues to send the commands in order of definition until the thread is canceled or the device is closed. This thread of execution runs independently of your Function Block Application, and is not synchronized with it in any way. If a command fails, the sequence is restarted. If the protocol is `SERIAL_HART` and a command fails, communication is reset with `command #0` and the sequence is restarted. The transmission of the commands within the new thread occurs as described in the previous section, *Descriptor Describes a Device.*

Possible Errors

`E_WRONG_ARGUMENT`	The descriptor is invalid or `cmd` is unknown.

nihGetData

Purpose

Retrieves the latest reply data to a command that is part of a sequence.

Format

```
int16 nihGetData(nihDesc_t desc, uint8 *rcvData, uint16 *stat,
                               bool_t *stale)
```

Includes

```
#include "types.h"
#include "hart.h"
```

Parameters

IN `desc`	The descriptor of the repetitive command—returned by `nihSendCommand`.
OUT `rcvData`	The address of the buffer into which the last received reply data must be stored.
OUT `stat`	The address of buffer to receive the communication status.
OUT `stale`	TRUE if the data returned has already been retrieved at least once by `nihGetData`. FALSE otherwise.

Description

The last received reply data and the associated status is returned in `rcvData` and `stat`. You must ensure that the `rcvData` buffer is of sufficient size to accommodate the reply data of the command identified by `desc`. For HART protocol, `stat` is the HART reply status. Otherwise, a stat of 0 indicates success, and nonzero indicates an error.

If the last transaction resulted in a communication error, the communication error code is returned in `stat` and no data is returned in `rcvData`.

Possible Errors

`E_WRONG_ARGUMENT`	The descriptor is invalid or `rcvData` is NULL.

nihPutData

Purpose

Updates the data associated with a command that is part of a sequence.

Format

```
int16 nihPutData(nihDesc_t desc, uint8 *sendData, uint16 *stat)
```

Includes

```
#include "types.h"
#include "hart.h"
```

Parameters

IN `desc`	The descriptor of the repetitive command—returned by `nihSendCommand`.
IN `sendData`	The address of the buffer containing the data to be sent with the command.
OUT `stat`	The address of buffer to receive the communication status.

Description

This function updates the data associated with a command that is part of a sequence. The new command data `sendData` is sent to the device the next time the command identified by `desc` is executed. The status of the last transmission of the command `desc` is returned in `stat`.

For generic serial, the entire transmitted string is replaced by `sendData` when you call `nihPutData`. `sendData` will then be used in all subsequent iterations of the specified command until you call `nihPutData`.

For HART serial, the user data portion of the HART packet is replaced by `sendData` when you call `nihPutData`. The rest of the packet remains the same (except the `checksum`, which is recalculated for you automatically). The new packet will be used in all subsequent iterations of the specified command until you call `nihPutData` again.

In both generic serial and HART serial, the number of bytes in the command that is to be transmitted remains the same before and after the call to `nihPutData`.

Possible Errors

`E_WRONG_ARGUMENT`	The descriptor is invalid.

nihCancelSequence

Purpose

Cancels a previously defined sequence.

Format

```
int16 nihCancelSequence(nihDesc_t desc)
```

Includes

```
#include "types.h"
#include "hart.h"
```

Parameters

IN `desc`	Descriptor of the command sequence to be canceled.

Description

This function is used to terminate the command sequence `desc` defined previously. `desc` is not valid after this call. The sequence will stop at the end of the currently executing command.

Possible Errors

`E_WRONG_ARGUMENT`	The sequence descriptor is invalid.

nihSetParam

Purpose

Configure certain communication parameters for the network or device.

Format

```
int16 nihSetParam(nihDesc_t busno, int16 option, int16 value)
```

Includes

```
#include "types.h"
#include "hart.h"
```

Parameters

IN `busno`	The number of the bus whose communication parameter is to be changed.
IN `option`	A parameter that selects the communication parameter to be changed.
IN `value`	The value to which the selected communication parameter is to be set.

Description

This function alters the current value of the communication parameter to the specified value for the selected bus. `option` must be one of the constants defined in Table 8-1. `value` must be valid for the parameter configured. The parameters can only be changed if there are no opened devices for the bus given by `busno`.

nihSetParam

Continued

Table 8-1. Constants Available for the Option Parameter

Constant	Values	Default
RETRY_COUNT	0 to 255	1
NUM_PREAMBLES	3 to 10	3
TIME_OUT	0 to 64 k ms	500 ms
SERIAL_PROTOCOL	SERIAL_GENERIC, SERIAL_HART	SERIAL_HART
SERIAL_BAUD_RATE	BAUD_300, BAUD_1200, BAUD_2400, BAUD_4800, BAUD_9600, BAUD_14400, BAUD_19200	BAUD_1200
SERIAL_PARITY	ODD_PARITY, EVEN_PARITY, NO_PARITY	ODD_PARITY
SERIAL_STOP_BITS	ONE_STOP, ONE_FIVE_STOP, TWO_STOP	ONE_STOP
HART_LONG_FORM	TRUE, FALSE	TRUE

Option Parameter Constants

RETRY_COUNT	The number of times a command is to be re-transmitted on an error.
NUM_PREAMBLES	The number of HART protocol preambles to be transmitted.
TIME_OUT	The amount of time, in ms, to wait before concluding an error.
SERIAL_PROTOCOL	The serial protocol to be used for the bus.
SERIAL_BAUD_RATE	The baud rate of the serial bus.
SERIAL_PARITY	The parity of the serial bus.
SERIAL_STOP_BITS	The number of stop bits for the serial bus.
HART_LONG_FORM	Use long-form HART addresses if TRUE (applies only to SERIAL_HART protocol).

Possible Errors

E_WRONG_ARGUMENT	The busno specified is not valid, or the option is invalid, or the value specified is out of range.
E_BUS_ACTIVE	The busno specified already has active device descriptors opened.

Appendix A

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a Fax-on-Demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at (512) 418-1111.

E-Mail Support (currently U.S. only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	01 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (___)____________________ Phone (___) ____________________

Computer brand ________________ Model ________________ Processor ________________

Operating system (include version number) ____________________________

Clock speed ______MHz RAM _____MB Display adapter ____________________

Mouse ___yes ___no Other adapters installed ____________________

Hard disk capacity _____MB Brand ____________________________

Instruments used ____

____

National Instruments hardware product model __________ Revision ________________

Configuration ____

National Instruments software product ______________________ Version ____________

Configuration ____

The problem is: ____

____

____

____

____

List any error messages: ____

____

____

The following steps reproduce the problem: ______________________________

____

____

____

____

____

Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

Interrupt level of hardware ______________________________

DMA channels of hardware ______________________________

Base I/O address of hardware ______________________________

Other Products

Computer make and model ______________________________

Microprocessor ______________________________

Clock frequency or speed ______________________________

Type of video board installed ______________________________

Operating system version ______________________________

Operating system mode ______________________________

Programming language ______________________________

Programming language version ______________________________

Other boards in system ______________________________

Base I/O address of other boards ______________________________

DMA channels of other boards ______________________________

Interrupt level of other boards ______________________________

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *NI-FBUS™ Function Block Shell Reference Manual*

Edition Date: January 1998

Part Number: 321016C-01

Please comment on the completeness, clarity, and organization of the manual.

____

If you find errors in the manual, please record the page numbers and describe the errors.

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

Phone (___)______________________ Fax (___) ______________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

Prefix	Meanings	Value
m-	milli-	10^{-3}
M-	mega-	10^{6}

ASCII	American Standard Code for Information Interchange
DD	Device Description
DMA	Direct Memory Access
FB	Function Block
FMS	Fieldbus Messaging Specification
HART	HART Field Communications Protocol
Hz	Hertz
I/O	Input/output
MB	Megabytes of memory
OD	Object Dictionary
QUU	Queued User-triggered Unidirectional
RAM	Random-Access Memory
s	Seconds
snap	Read from the communications stack
VFD	Virtual Field Device