SW Software

StateWORKS®

RTDB
Reference Manual for the Class Library

Release 5.0.6

Recent Revision History[1]

The state*WORKS*® Release 4.0 adds some useful features like trace to the database items. Some class interfaces have changed.

The state*WORKS*® Release 4.1 implements a new dependency mechanism among subclasses of CItem. The C_NO and CIO_Handler have got a new SetOutput concept. Some classes got additional interfaces. 4.0 is forward compatible.

The state*WORKS*® Release 4.2 has improved trace feature, added DDE items and attributes, new accesses to existing classes, various other changes and improvements.

The state*WORKS*® Release 4.3 supports internationalization, data objects can be enumerations, easier use of enumeration class, parameters with extended categories, digital In/Output have invert attribute, alarm text more complex, alarms sent to the event log, new methods for existing classes.

The state*WORKS*® Release 4.4 additionally to DDE support acts now as a pure TCP/IP server. The VFSM execution has now a single step mode to ease debugging of state machine systems.

The state*WORKS*® Release 5.0 is based on generic C++ and so eases porting to other OS like Linux or VxWORKS®. There is a new item type: STR that allows extensive string parsing.

[1] Revision code: r.r.x x is for formal changes only

Contents

Introduction

This *VFSM System Class Library Reference Manual* covers the classes, global functions and declarations used to build a full fledged control system on the surface of Windows XP/2000/NT (or generally WIN32) or other Operating Systems. The manual is divided into two parts:

Introduction to the VFSM System Class Library

The VFSM System Class Library Reference

Part 1 shows how to work with the VFSM System classes. Chapter 1 lists the classes in helpful categories. Chapter 2 shows how to integrate a VFSM System into a VisualC++ application framework. Chapter 3 describes with examples how to write IO-Handlers, the links to the input/output hardware. Chapter 4 shows how to write your own output functions to Vfsm's output actions. Chapter 5 explains function and details of the host interface to the real-time database. And at last chapter 6 shows the concept of internationalization of the message texts.

Part 2 contains the following components:

- A description of a selection of the classes used by a system integrator
- A section that explains global declarations

Be aware that the class documentation does not include repeated descriptions of inherited member functions. Overridden virtual member functions are included if their implementation is different from the superclass. If not included you must refer to the base classes depicted in the hierarchy diagram.

The class description is not complete. In principle, it includes classes and methods that are used to understand and integrate the VFSM System to an application and to write IO-Handlers and output functions. Not all described classes are really needed when coding but some of them are included in the manual for better system understanding (for instance the C_VFSM class). Several classes which are used only internally in the RTDB are completely omitted. The Manual documents ca. 30 classes from 70 classes which are used by the RTDB implementation.

Intended Audience

This manual is designed to be used as a guide for the state*WORKS* developer who has an understanding of object-oriented concepts, multitasking operating systems and C++ programming language. It is also assumed that the developer knows the state*WORKS* Development Tools.

Document Conventions

This manual uses the following typographic conventions.

Example of Convention	Description
CIO_Handler::SetOutput	C++ methods and references to classes, procedure declarations.
stAttrKey	In text and method/procedure declarations, italic letters indicate placeholders. In text italic words refer to titles, lists or examples.
SYSTYP.H	Words in capital letters indicate file names or constants.
`Void SetOutUnit (int n, WORD w) { int ret = Out(n, w); }`	This font is used for sample code or sample configuration text. Also for items typically used in program context, e.g. enumeration.
Associated Item List (AIL)	Acronyms are usually spelled out the first time they are used.
`VSFILE_AllWords`	Global procedures have a preamble consisting of the module file name (VSFILE.C/.H)
`<NL>`	Non printable ASCII characters (e.g. NL = new line = 0x0A)
IOD-File, *.IOD	The IOD-file, all files with the extension IOD
`CS_RESET, CC_Reset`	Enumeration representations for the states of the several item types are written in all capital letters. Commands are written with the first letter capitalised.
C_AL	The class of the alarm items. (This doesn't apply only to AL items but to all item types)
AL	The state*WORKS* Studio doesn't use the preamble C_. But it means the same as C_AL.
AL item or C_AL item	Used to point out that they belong to the database
C_AL item object or C_AL object	Used to point out that a certain instance of C_AL is meant.
LinkItem	Name of the item in a host conversation (comes from DDE/VisualBasic).

Registered Trade Marks: All trade marks acknowledged as such.

References to "WINDOWS", Win32, Window NT etc. are all intended to refer to the Microsoft products of those names, as is quite clear from the context, so ™ , ©and ® symbols have been used sparingly, so as to avoid an irritating appearance of the document.

References to "WindowsNT" imply various Microsoft products, including Windows NT4 and Windows NT5 – namely Windows 2000 and Windows XP – and future compatible products.

References to "UNIX" or "UNIX-like" operating systems acknowledge the rights of the UNIX trade-mark holder to that name.

P A R T 1

Introduction to the VFSM System Class Library

C H A P T E R 1

The VFSM System Class Library

This chapter categorizes the classes in the VFSM System Class Library version 5.0. These classes support control application development for Microsoft® WindowsNT/2000/XP (generally WIN32) or other Operating Systems, with or without disc storage. (Note that the software is not based on the Microsdoft Foundation Classes, and this implies compatibility to other systems such as UNIX™ and derivatives of its ideas such as Linux or VxWorks™.)

Because the class library is built around a Real-time Database (RTDB) containing objects used in control applications the database item classes are the largest and most important group of classes. Although the whole system consists of more than seventy classes this manual only describes the thirty or so that are interesting to system integrators and those users who write their own I/O handlers and output functions.

The following categories of classes from VFSM System library are presented here:

- RTDB Management
- RTDB General Items
- RTDB I/O-Items
- Accessories to the RTDB Items

At the end of this chapter there is a map showing all these classes and their hierarchical dependencies.

Real-time Database Management

Management of the databases means at System Startup to build-up the database with its items according to the Configuration File and the Description Files. At runtime clients have access to the database that acts as a data server. The following class represents the databases:

CItemList	Collection of all the item objects

CItemList is the heart of the Real-time Database. At System Startup it reads the Configuration File and creates, initializes and connects the database items. During runtime the entries are accessed directly and fast by pointers. The host interface accesses the items by their names. CItemList performs this access. Every item has itself a global access to CItemList. So it could talk to any other item if it knew its name.

RTDB General Items

The database items perform the control system's control flow. They are created and connected during System Startup according to the Configuration File. The general items have no direct connection to the system's environment (except to the file and timer system):

Citem	Superclass of all items, performing the systems control flow. This is a virtual class; there are no instances of this class.
C_AL	Alarm object with alarm-text.
C_CMD	Command to a Vfsm possibly with command text.
C_CNT	Counter with counter-register and constant defining overflow.
C_DAT	Holds and organizes a data.
C_ECNT	Subclass of C_CNT counting external events.
C_OFUN	Connection to the system environment of a user written output function.
C_PAR	Subclass of C_DAT holding system's parameters, possibly persistent.
C_SWIP	Supervises low and high limits for a C_DAT and C_DAT derived item.
C_STR	Extracts substrings from a C_DAT and C_DAT derived item (string) using regular expressions.
C_TAB	Multiplexes several C_DAT and C_DAT derived items to one output.
C_TI	Timer object with several timebase options, subclass of C_CNT.
C_UDC	Up/Down counter, subclass of C_DAT.
C_VFSM	Holds a Virtual Finite State Machine.
C_XDA	Holds a certain number of bytes.

The class CItem is the superclass of all item types. The class C_DAT is the superclass of all the items that additionally perform a data flow especially in a form of C_PAR and C_UDC classes. The class C_CNT is a superclass of items that perform counting functions, like C_ECNT and C_TI.

RTDB IO-Items

IO-Items represent the connection to or from the system's peripherals:

C_DI	Digital Input.
C_DO	Digital Output.
C_NI	Numerical Input.
C_NO	Numerical Output.
C_UNIT	Object collecting several IO-Items.

C_UNIT items have no data and no control flow. They just act as containers for the other IO-Items. C_NI and C_NO items are subclasses of C_DAT and perform this way a data flow.

Remark The distinction between General and IO-Items does not mean that the General items cannot connect to system peripherals. Some of them can do this using the dependency mechanism.

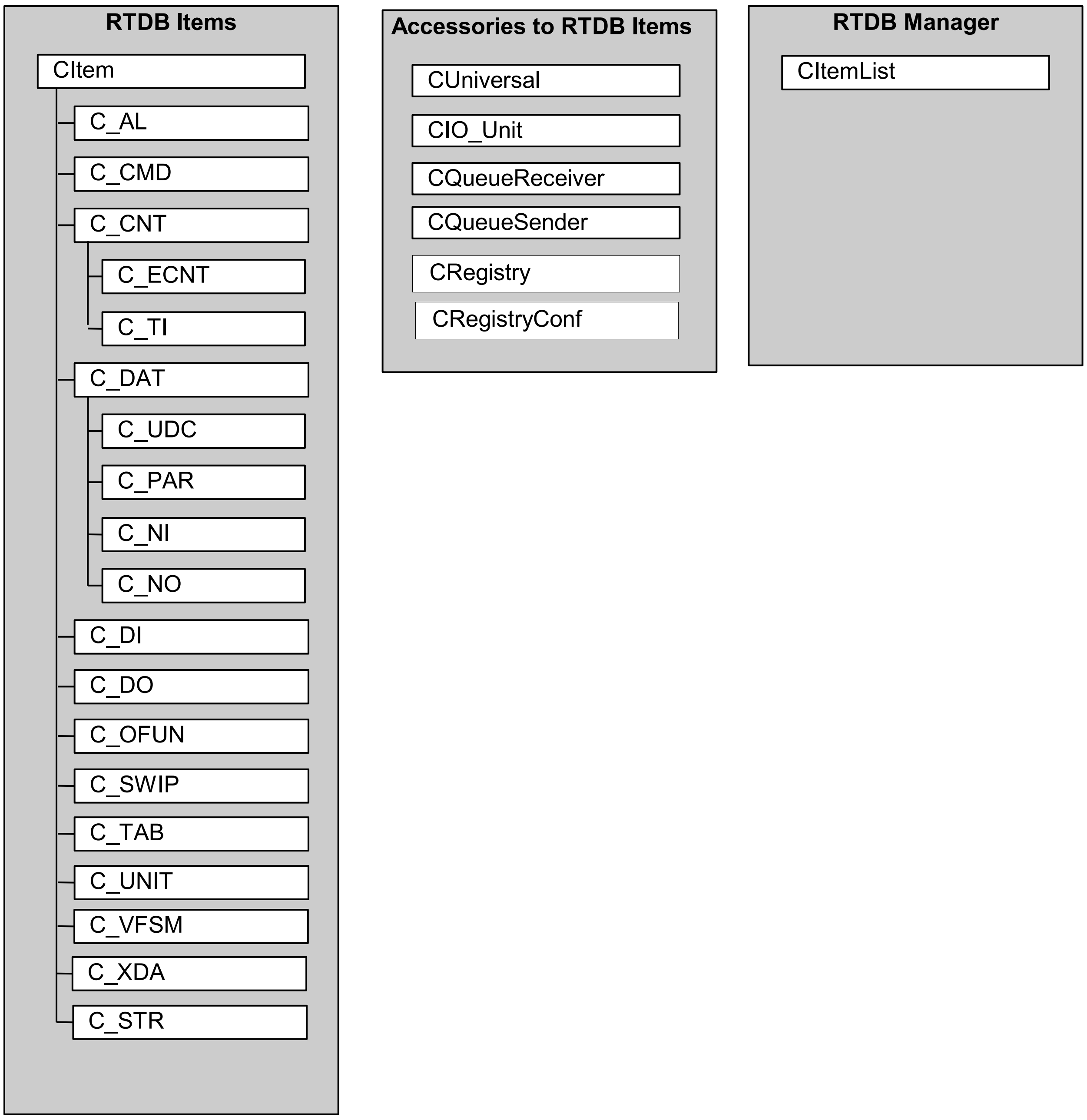
Accessories to the RTDB Items

The following accessory classes work together with the database items. They give them the specific item behavior:

CAssItemList	Collection of items working together with the owner item (C_VFSM, C_UNIT).
CUniversal	Data element in several formats and operators.
CIO_Handler	Virtual superclass of the user written IO-Handlers.
CQueueReceiver	Message queue receiver with process synchronisation.
CQueueSender	Counter part of CQueueReceiver.
CRegistry	Access to Registry or Registry (EP parameter persistency)
CRegistryConf	Access to Registry or Registry files (System configuration paths)

CAssItemList is used only by C_UNIT and C_VFSM items to store the pointers to the items they work together with. *CUniversal* is a universal data type with the according data manipulations. It is used by all C_DAT type items; it is "the data" in the system's data flow. *CIO_Handler* is the superclass of all (user written) IO-Handlers. It holds the connection to one C_UNIT item and passes this way the appropriate information from the Configuration File to the IO-Handler. *CQueueReceiver* and *CQueueSender* is the implementation of a fast message queue mechanism usable for instance between IO-Handlers and IO-Handler threads. The CRegistry and CRegistryConf are used to access Windows Registry or equivalent Registry files in a non-Windows environment.

Map of the Class Library



C H A P T E R 2

Integration of the VFSM System into a Win32 Application

This chapter describes how to integrate a VFSM System with the RTDB to a Win32 operating system like WindowsNT. The environment is assumed to be a Microsoft VisualC++/Foundation Class Document/View Framework application (MFC).

Remark The RTDB itself is not based on MFC. So it is possible to integrate a VFSM Sytem with any multi-tasking (real-time) OS.

In the following section we will integrate a VFSM System to a VisualC++ framework. Although the description is in a form of cook-book, a basic understanding of the VisualC++ workbench is recommended. Be aware that the result is a very simple application, just to show the hooks of the system.

First let the Application Wizard create a single document application (MFC in a static library). Rename the following classes:

Application class	to CvfsmApp
Document class	to CvfsmDoc
View class	to CvfsmView

Then let the Class Wizard generate the methods:

"OnIdle"	in CVfsmApp
"OnTimer"	in CVfsmView
"OnCreate"	in CVfsmView

Where to place the VFSM System

The integration is done in four steps: place the RTDB database, load the configuration file, attach to system timer, and attach to Windows queue. And sometimes after running the system a very last step is necessary: destruction of the VFSM System and the RTDB.

Place the database

The database is considered a kind of document, so it is placed in the document class. Enter for that the declaration of the VFSM System library to the CVfsmDoc's h-file and the database as a member variable to the class definition:

```
// Doc.h : interface of the CVfsmDoc class
#include "vswin.h"
class CVfsmDoc : public CDocument
{
// Attributes
public:
  CVfsmSystem  m_VfsmSystem;
// other object belongings
}
```

The CVfsmApp needs to know the database. So it gets a pointer to it. Don't forget to initialize the pointer to NULL in the class constructor. The CVfsmApp's h-file looks like this:

```
class CVfsmSystem;
class CVfsmApp : public CWinApp
{
public:
// Attributes
      CVfsmSystem* m_pVfsmSystem;
// other object belongings
}
```

The pointer is connected in the CVfsmDoc's constructor. Add for that the following lines:

```
CVfsmApp* pWinApp = (CVfsmApp*)AfxGetApp();
  pWinApp->m_pVfsmSystem = &m_VfsmSystem;
```

StateWORKS Studio generated files

The RTDB library is the heart of the VFSM System. The RTDB contains the VFSM Executor which carries out specifications of state machines, effectively controlling the application behavior. The structure of the RTDB and its behavior is specified by a set of files generated by state*WORKS* Studio:

ProjectName.swd	Configuration file.
StateMachineName.h	Contains enumerations which define: Items used by the state machine, virtual Input Names, virtual Output Names State Names and Command Names.
StateMachineName.iod	Contains string lists corresponding to enumerations in StateMachineName.h.
StateMachineName.str	Contains strings which define state machine behavior.

UnitName.h	Contains enumerations which define Items used by IO-Handler or Output Function and Command Names.
UnitName.iod	Contains string lists corresponding to enumerations in UnitName.h.

There is one configuration (SWD) file for an application. The configuration file defines all RTDB items.

Each state machine type and each unit type has its own H-, STR- and IOD-file. The H-files are used for writing IO-Handlers and Output Functions. The IOD- and STR-files are rather irrelevant for programming.

Load the Configuration File

Add the following lines. At last the CVfsmDoc's constructor will look something like this:

```
CVfsmDoc::CVfsmDoc()
{
CVfsmApp* pWinApp = (CVfsmApp*)AfxGetApp();
  pWinApp->m_pVfsmSystem = &m_VfsmSystem;

CString   stVfsmDir    = "C:\\VS\\VFSMEXA\\DivTests\\";
CString   stDataDir    = "C:\\VS\\VFSMEXA\\DivTests\\";
CString   stConfigName = "C:\\VS\\VFSMEXA\\DivTests\\udc.swd";
bool      bConfigOk;
bConfigOk = m_VfsmSystem.Create( ConvertString(stVfsmDir),
                                 ConvertString(stDataDir),
                                 ConvertString(stConfigName) );
}
```

The VFSM System needs to know two directories. One directory (stVfsmDir) where it expects the VFSM description files (the *.IOD and *.STR files) and a second directory (stDataDir) where it puts the startup log-file. Then it needs the configuration file (stConfigName). Usually all these files are in the same directory. Of course these lines have to be adapted to the directory where the files really are. *Create()* produces a file named SULOG.TXT[2] in the *stDataDir*-directory. It reports information (date of creation, statistics), warnings (if there are missing objects but the VFSM System is able to run with default values) and at last errors. Errors are for instance missing VFSM description files or no configuration file at all. If there are errors *Create()* returns false. Then it would be better not to continue. The VFSM System is most likely unable to run and can even produce an exception. If *Create()* returns true the database with all its objects and state machines is created but there is no access to them yet.

Note, that the example used fixed path and file names. A more realistic solution gets the start-up information from the Windows Registry (see CRegister and CRegisterConf classes) or from equivalent Registry files for a non-Windows environment.

Attach to System Timer

Add the statement *SetTimer()* to the CVfsmView's method *OnCreate()*. It then looks finally like this:

[2] Because there may be entries made later the file sulog.txt remains open until the system stops. It can be read while open, for instance with the Notebook editor.

```
int CVfsmView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
      if (CView::OnCreate(lpCreateStruct) == -1)
            return -1;

// TODO: Add your specialized creation code here
CVfsmDoc* pDoc = GetDocument ();
 pDoc->m_VfsmSystem.AttachToMainWindow (this);
 SetTimer (1, 100, NULL);
 return 0;
}
```

The timer started in the second last line is used as the VFSM System's timebase. CVfsmSystem class' method *TimeBaseTick()* is called in the View object's method *OnTime()* that is called every 100ms by the Windows event handler:

```
void CVfsmView::OnTimer(UINT nIDEvent)
{
CVfsmDoc* pDoc = GetDocument ();
 if(nIDEvent == 1) pDoc->m_VfsmSystem.TimeBaseTick();
}
```

Attach to Windows Queue

Complete the CVfsmApp class' method *OnIdle()* as follows:

```
bool CVfsmApp::OnIdle(LONG lCount)
{
 if(m_pVfsmSystem != NULL)
 {
  while ( m_pVfsmSystem->PollAdviseQueue() ) { };
 }
 return CWinApp::OnIdle(lCount);
}
```

Between the database and the Windows thread there is a message queue. It decouples the real-time database events from the Windows event queue. *OnIdle()* is called by the Windows event queue on its idle time. *PollAdviseQueue()* checks for entries in the database queue and puts the database events to the Windows queue in a form of Host (TCP/IP and DDE) events.

The VFSM System's real-time database is now fully functioning. It is able to answer the Host clients' requests. It handles timer events and it can send these events to its Host clients. What's missing now is the link to the hardware. There is no connection to input and output hardware yet.

Destruct the VFSM System

The VFSM System is able to destruct itself when the application stops existing. This is the case in the very simple example above. A more comfortable application should be able to select and change the database configuration via the framework's file handling. Then it is necessary to destruct the RTDB programmatically. Use for that the following statement:

```
m_VfsmSystem.RemoveAll();
```

This could be placed for instance in the CDocument class' method *OnNewDocument()*. *RemoveAll()* deletes all the objects from the real-time database. It is then ready for a next *Create()* with another configuration file.

The RTDB Layer Model

The database is located between the interfaces to the Host (which may be typically a User-Interface) and the IO- and Timer-Handler. It has no direct contact to user or peripherals:

Host (User Interface)	
Host Interface	
Real-time Database (RTDB)	
IO-Handler	Timer-Handler
Peripherals	

According to the three boxes around the database there are three sources of events:

- from the Host Interface
- from the Timer
- from the IO hardware

Input Events

From the Host Interface

The host interface is message oriented. For instance a GUI (graphical user interface) might reside in another program or even on a different computer. It has no direct access to the RTDB. It sends messages containing commands to the RTDB’s host interface. So the arrival of such a message is an input event.

From the Timer

The Real-time Database is assumed to be invoked every time interval (tick) - for instance every 100ms or every second, so that it can handle its own timer objects. Such a tick doesn't directly produce events. It rather triggers the timer objects to increment their counter registers. If a timer object expires, then an event is produced, so only then does the tick produce an event.

From the IO-Handler

In a real-time system we expect most of the events to come from the input hardware. In the VFSM System these events are typically not produced by the inputs themselves. This only can happen when there is an interrupt-driven input handler. More typical input handlers are so-called polling loops which are triggered (or invoked) periodically. Polling means that a regular time intervals all the inputs are read from the peripherals and checked whether they have changed since the previous reading. If so, then an input event is produced. But initially the event was triggered by a time interval just like the timer events. So in the case of input polling there are only two sources of events to the VFSM System: the User Interface and the Timer. Then the layer model can be altered to this:

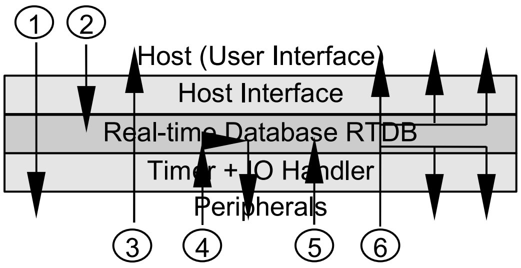
Host (User Interface)
Host Interface
Real-time Database (RTDB)
Timer- + IO-Handler
Peripherals

Output Events

Outputs are events too but only to the outside world, the Host Interface and the peripherals. They do not trigger the VFSM System; they are produced by it and then forgotten. Of course in a control system the outputs are the most important thing, but because of their straitforward behavior they cause us little trouble.

Event Control Flow

The following picture shows some scenarios of control flows. Control flows always start with an event. Sometimes they are without consequence but usually they end at an output.



1. Command from the User Interface (UI), an output to the peripherals is set.
2. Command from the UI, no output is set. This typically happens, when the UI sets some data to the database (this is rather a data flow than a control flow) or when the command is stored in the database for later use (later use means that the occurrence of this event is a prerequisite in a combination of several events, so the command will be effective at a later time).
3. An input from the peripherals produces an event. This causes a message to the User Interface. The UI for instance sets a certain display.
4. An input from the peripherals produces an event that sets an output to the peripherals.
5. An input from the peripherals produces an event. The event is either ignored or stored in the database for later use (see also 2.) but there is no immediate reaction.
6. One event (from peripherals or UI) can lead to the generation of several messages to the User Interface and/or several output settings.

VFSM Control Flow

The stateWORKS RTDB implements the virtual finite state machine (VFSM) concept which insists that data and control flow are separated in the software.

Software operates on data which are processed: transformed, stored or transferred. The task of the control flow is to decide how and when data are processed.

Data may be of different types: digital (two-valued), integer, float, or strings for example. Software is triggered by various signals, such as inputs from the external or internal devices, interrupts or timers. All triggers and data contain control information which we call control values. Examples:

– A digital value is pure control information having two[3] control values: On and Off. Note that the value On may be: true or false, high or low, enabled or disabled depending on the context.

– An integer value may represent commands (Cmd_Start, Cmd_Stop, Cmd_Go, Cmd_Continue, Cmd_On, Cmd_Off, etc.).

– Ranges of a float value may be represented by control values like: Level_Low, Level_High, Level_Ok, etc.

[3] Actually, in the VFSM concept any data type has also the control value *unknown*.

– Timer expiration may generate a control value: Timer_OVER. Note that in some applications other Timer states (Timer_Running, Timer_ Stopped, etc.) may have significant control meanings.

– States of a state machine represent control values: Motor_On, Motor_OnBusy, Motor_OffBusy, Motor_Off, etc. and are often used as inputs to higher level state machines.

Control values defined in an application should represent the entire control information and should be used to specify the behaviour of the application software. All control values defined for an application represent **the** control information relevant to that application and are called *virtual inputs*. For instance, values defined in the examples above may be virtual inputs of a certain application:

On, Off, Cmd_Start, Cmd_Stop, Cmd_Go, Cmd_Continue, Cmd_On, Cmd_Off, Level_Low, Level_High, Level_Ok, Timer_OVER, Timer_Running, Timer_ Stopped, Motor_On, otor_OnBusy, Motor_OffBusy, Motor_Off.

The RTDB filters the control values from external (digital inputs, analog (numerical) inputs, commands) and internal (timers, counters, parameters, state machines) objects. The VFSM state machine Executor uses these values, as virtual inputs to control the application.

The same concept applies to outputs. Outputs in the control flow are descriptions (names) of activities. In contrast to inputs the outputs are not used for control of the software, but represent only its actions; therefore we omit the word *control* while speaking about outputs (otherwise we should use *input control values* and *output control values* to distinguish the two). Examples:

– A digital value has two values[4] called: On and Off. Note that for instance the value On may be: true or false, high or low, enabled or disabled, etc. depending on the context.

– An integer value may represent commands (Cmd_Start, Cmd_Stop, Cmd_Go, Cmd_Continue, Cmd_On, Cmd_Off, etc.) sent typically to other state machines.

– Setting of a float value may be defined as values: No_Off, No_On specifying that a physical value is to be set, the true data value being irrelevant for control purposes in terms of software behaviour, even though it will be important for the application.

– Timer control requires values like: Timer_Start, Timer_Reset, Timer_Stop, etc.

As for virtual inputs, virtual outputs are defined as a set of all output values defined for a given application. For instance, values defined in the examples above may be virtual outputs of a certain application:

On, Off, Cmd_Start, Cmd_Stop, Cmd_Go, Cmd_Continue, Cmd_On, Cmd_Off, No_Off, No_On, Timer_Start, Timer_Reset, Timer_Stop.

The VFSM state machine Executor produces these values. The RTDB deals with them, as actions, by passing them to I/O-Handlers, Output Functions or triggering internal devices.

[4] Also in this case outputs have the additional value which exists after the system start-up: the value "not set". Note also that for instance the value *On* once set cannot be removed but is rather replaced by *Off*.

Libraries

The VFSM System employs three libraries:

1. the Database Library
2. the IOH-Library and
3. the User Output Function Library.

The *Database Library* is hidden from the user. It only gives access to its classes and methods via the h-files. The implementation depends on the Operating System. For instance, for Win32 it is called "WinRTDB.lib" and the h-files are placed in a directory \INC.

The *IOH-Library* contains all the Input/Output-Handler dependent classes and methods. As these files have to be written by the user (or the distributor) of the system they are open to the user (h- and cpp-files).

The *User Output Function Library* has to be written by the user (or programmer) of the system. Here, there are the output functions that are not provided by the VFSM system (for instance arithmetic, data collection, file handling). Of course, like the IOH-Library it is open to the user.

C H A P T E R 3

Writing IO-Handlers

This chapter describes how to write IO-Handlers and how to attach them to the VFSM System. The same framework example as in the previous chapter is used to explain an example of an IO-Handler.

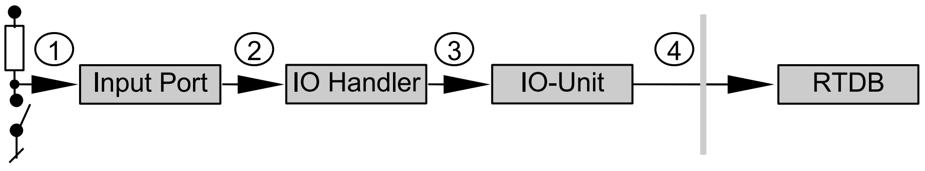
An IO-Handler is derived from the CIO_Handler class. CIO_Handler class implements connections to the database.

Connection to the IO-Hardware

The hardware could be attached directly to the computer bus or it could be connected via a serial line or a local area network. In any case, the IO-hardware is assumed to be divided into units with a certain physical address and/or communication port. A unit can have a number of inputs, outputs or both called the channels. A simple example is a digital input (DI) port. The 8 bits of its data byte are the 8 channels and its IO-address is the physical address. A hardware unit is represented (and controlled) by its software counterpart, the IO-Handler. An IO-Handler is a class derived from the superclass CIO_Handler. It holds all the data and methods needed to control the appropriate hardware. In the above DI example the IO-Handler would store the last read value to find out which bits have changed.

Connection to the Input Hardware

The input control flow goes from the electrical signal via input hardware, the IO-Handler to the Database.



The information whether the switch is open or closed is passed over four connections:

1. The information is produced at the physical interface.
2. The IO-Handler gets (or fetches) the information from the appropriate IO-hardware.
3. The IO-Handler processes the information and places it in the input items of the database. This is described later in the section "Connection to the Database".

The IO-Handler gets (or fetches) the information from the appropriate IO-hardware:

As mentioned earlier there are two ways to get information from the hardware to the IO-Handler:

– The IO-Handler is invoked by an interrupt from the input hardware itself.

– The IO-Handler is invoked periodically by a timer. Of course within the computer system this is an interrupt too.

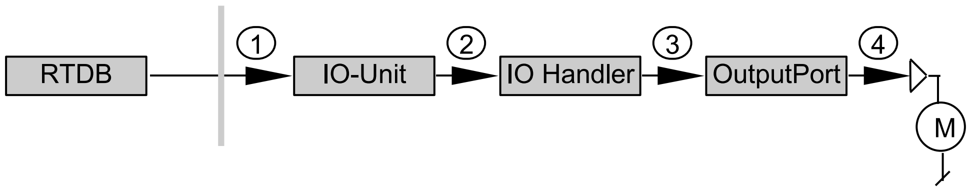
The direct input interrupt could be faster because of the immediate handling and because the interrupt itself points to the data that have changed. In the polling method some time passes between the polling cycles, and in every polling cycle all the inputs have to be read because nobody knows which of them have changed. Nevertheless, although the direct method is more efficient, in most cases the polling method is employed because of its simpler handling. In both cases the IO-Handler is invoked or called by an interrupt service routine. The IO-Handler has then to get the physical address of the data either via the interrupt vector or in the polling method by stepping through all its physical addresses.

The IO-Handler passes the raw information to the IO-Handler:

Let’s assume, the IO-Handler has just read the raw data from one physical address. It has now to pass this data exactly to the one IO-Handler object that belongs to this physical address. For every physical address there must be an IO-Handler of the type that fits to the appropriate hardware. There are two major ways of running an IO-Handler. An IO-Handler can be passive. It is assumed to be called periodically by a timer of the application framework. An active IO-Handler is able to run periodically by itself. It has for that purpose its own thread that wakes it up at certain time periods. If there are several IO-Handlers of that kind, it happens that two or more of them access a certain database item at the same time. What happens if one handler sets a database item to one value and another handler sets it to another value? The database is able to handle this conflict. It does this by allowing only one thread at the time to access its items. This is implemented by declaring the database as a Critical Region. Only one thread can enter that region; the others have to wait.

Connection to the Output Hardware

We look at the control flow between the database and the output hardware.



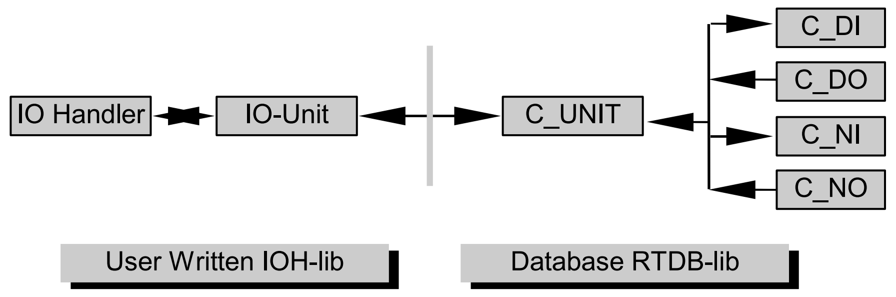
The output data passes in the same way as described in the input hardware, but in the opposite direction:

1. This step is described also in the section "Connection to the database". The database wants to set an output at a specified physical address and channel. To do that it calls a method of the IO-Handler object. This method is a virtual method of the IO-Handler object's superclass CIO_Handler. This is because the database doesn't know the IO-Handler derivation we use for this special output. The database only knows the superclass. At the moment there are five overlaid methods called ***SetOutput()*** to set data of the type bool, short integer (16Bit), long integer (32Bit) and float (32Bit) to a specified channel. The fifth produces triggers only the IO-Handler assuming that it is up to the IO-Handler to supply the data.One of the ***SetOutput()*** overlays (which one depends on the Output-Handler type, the others typically are dummies) calls a routine of the IO-Handler (with physical address and data) that copies the data to the specified output hardware.
2. The IO-Handler may be able to write e.g. 8 or 16 output channels at a time.

3. A digital output might, for example, reach an amplifier that drives a motor or a solenoid. This step is beyond the scope of this description.

Connection to the Database

So far we saw that for every piece of IO-hardware with a unique physical address region there is an IO-Handler object of appropriate type or class. Now it goes on this way: every IO-Handler has exactly one database item of type C_UNIT as a kind of partner or cooperative object:



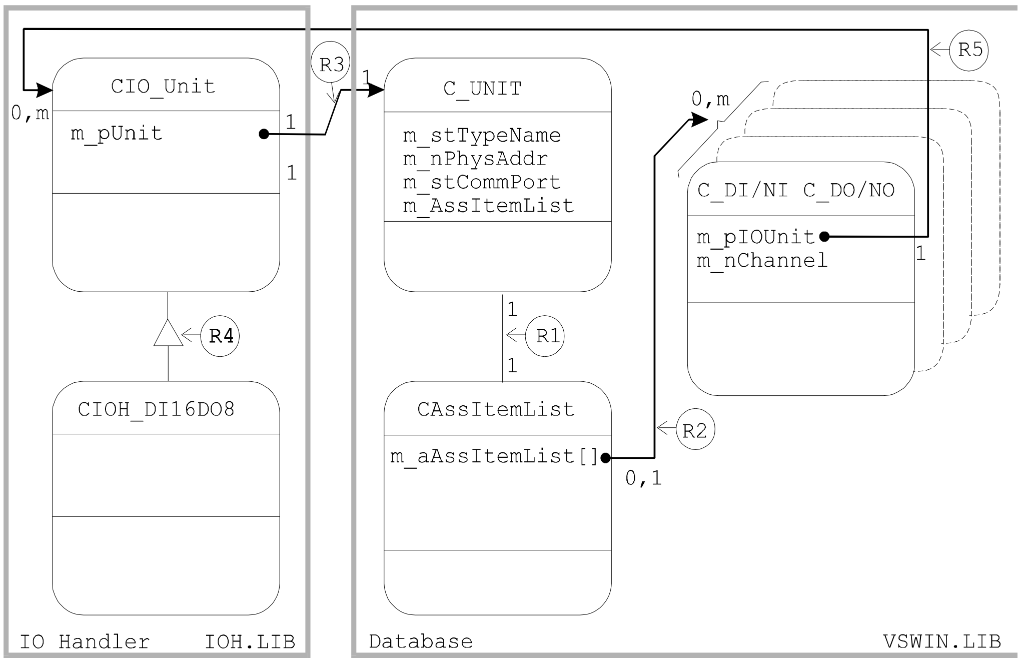
The reason for this is to keep things local:

- The IO-Handler objects deal with the hardware related things. But they have no knowledge about the configuration of the database and the IO-Items (these are the digital inputs/outputs, C_DI or C_DO and the numerical inputs/outputs C_NI or C_NO).
- The C_UNIT database items are created according to the database Configuration File. They get and keep all the information needed by the partner IO-Handler: the type of the IO-Handler, the physical address and the IO items that belong to this IO-Handler (for instance which 8 DI items belong to the 8 bits of the input port). The C_UNIT item has no knowledge how to map these IO items to the IO-hardware and how to access the hardware.

The cooperation between C_UNIT item object and an IO-Handler object is a typical feature of Object Oriented Programming (OOP). The connection exists between the class C_UNIT and the superclass CIO_Handler. The concrete CIO_Handler class inherits the connection to the C_UNIT items from their superclass. A C_UNIT item object doesn't know the physical IO-Handler object, only its superclass. Via the superclass's virtual methods it has access to the physical IO-Handler without knowing about its existence.

The Classes of IO-Handler and C_UNIT

The following section shows the cooperation between the three classes in a diagram. The IO items are the database items representing one physical in- or output. Their classes are the C_DI, C_DO, C_NI and C_NO. The C_UNIT items are database items that collect a number of IO items according to the set of physical IO objects on one physical address (the IO address). The IO-Handler derived classes represent the physical IO's individual behavior. Toward the database they are represented by their superclass CIO_Handler. The IO-Handler and the C_UNIT-items are members of the database. The specific IO-Handlers are members of the IO-Handler.



The picture shows two design components: the database and the IO-Handler. The database component is untouchable by the user; it is a part of the system software. The user (or the distributor of a VFSM System) can expand the IO-Handler component by writing derived classes of CIO_Handler. CIO_Handler transmits the instance connections to its derivations.

Remarks Connections are relations like inheritance and membership and so marked as Rx (e.g. R3). There is not a complete representation of the classes. Members, methods and relations are omitted if not interesting in this context.

R1, R2 C_UNIT objects have access to IO items via an object called Associated Item List. This list has an array of pointers that point to the associated IO items. The array is filled during startup according to the C_UNIT item's configuration. An element of the array can point to one or no IO item. IO items can be pointed by none to several C_UNIT item objects (typically by one).

R3 Access to the C_UNIT item's members like physical address and Associated Item List.

R4 CIO_Handler is a virtual class. So all IO-Handlers are objects of a class derived from CIO_Handler.

R5 Only the output items (C_DO and C_NO) have this connection. They have a member m_pIOHandler that points to the IO-Handler they belong to and a number m_nChannel that shows the position within the IO-Handler's array of output items.

How to write IO-Handlers

To write IO-Handlers is a programming task. It is done in the following three steps:

1. Write the IO-Handler’s code in the C++ language
2. Compile and build the library IOH.LIB
3. Link the whole application together with the VFSM System (VSWIN.LIB), the IO-Handler (IOH.LIB) and the output functions (OFU.LIB)

In the following sections there is a detailed description with an example of the first step. The last two steps depend on the programming environment and are beyond the scope of this manual.

The partnership between IO-Handler (with IO-Handler) and the database C_UNIT items (with IO-Items) has four aspects we now look at separately:

1. System Startup Phase

2. System Shutdown Phase
3. Inputs at Runtime (from input handler to input items)
4. Outputs at Runtime (from output item to the output handler)

System Startup Phase

From the IO-Handler's point of view startup is done in three steps:

1. Build up the RTDB
2. Initialize the IO-Handler
3. Run the IO-Handler

Build up the RTDB according to the Configuration File is done itself in several phases. This is beyond the scope of this section: When the IO-Handler steps into life it assumes that the database is completely built up and ready to run.

Initialize the IO-Handler. The IO-Handler has access to the database via *pDoc*. It uses this to get all the C_UNIT items of the system. To get the C_UNIT item objects means to get the pointers to the objects so that the object's methods are callable. The following code shows only the frame of the while-loop that asks for all C_UNIT items:

```
CVfsmDoc*      pDoc = GetDocument();
C_UNIT*        pUnit;
POSITION       pos;

 pUnit = (C_UNIT*)(pDoc->m_VfsmSystem.FirstItem(IT_UNIT, pos) );
 while ( pUnit != NULL )
 {

  // evaluate pUnit and create appropriate IO-Handler

  pUnit = (C_UNIT*)(pDoc->m_VfsmSystem.NextItem(IT_UNIT,pos));
 } /* End of while() */
```

pDoc is the access to the database engine that resides in the application's document. The two methods *FirstItem()* and *NextItem()* serve to step through the whole database and get all the items of a specified type (IT_UNIT is the element of an enumeration, see VSYSTYP.H). The variable *pos* serves as the loop counter. *First-* and *NextItem()* return a pointer to an object of the superclass CItem. Because we know, that the object is of type C_UNIT it is copied to the pointer *pUnit* of this type. So we have access to the C_UNIT specific methods. That's all we need from the database for the moment. Next we have to create the IO-Handler objects according the attributes of the C_UNIT object. This can differ according the type of the IO-Handler.

Let's assume that we have smart IO-Handlers that poll autonomously for their inputs. Then the application framework only has to create the IO-Handler units and keep the pointers in an array. We need these pointers to destroy the IO-Handlers at system shutdown.

The following piece of code shows the creation of some types of IO-Handlers. We have wrapped the code into a procedure *CreateIOH()* that could be called e.g. in the views method ***OnCreate()***.

```
CIO_Handler*  m_apIOHandler[BigEnough]; // keep this as ..
int        m_numbIOHandlers;         // ..class members

//------------------------------------------------------------
  void CVfsmView::CreateIOH(void)
//------------------------------------------------------------
CVfsmSystem* pVS = &(GetDocument()->m_VfsmSystem);
C_UNIT*      pUnit;
POSITION     pos;
bool         bOk;

{
 m_numbIOHandlers = 0;
 pUnit = (C_UNIT*)(pVS->FirstItem(IT_UNIT, pos));
 while ( pUnit != NULL )
 {
  if ( pUnit->GetUnitTypeName() == "DI16DO8" )
  {
   m_apIOHandler[m_nnumbIOHandler] = new CIO_HandlerDi16Do8;
   bOk = m_apIOHandler[m_nnumbIOHandler]->Create(pUnit);
  }
  if ( pUnit->GetUnitTypeName() == "AI8AO2" )
  {
   m_apIOHandler[m_nnumbIOHandler] = new CIO_HandlerAi8Ao2;
   bOk = m_apIOHandler[m_nnumbIOHandler]->Create(pUnit);
  }

  // possibly there are other IO-Handler types

  if(bOk)
  {
   m_apIOHandler[m_nnumbIOHandler]->Connect();
   m_nnumbIOHandler++;
  }
  else delete m_apIOHandler[m_nnumbIOHandler];
  pUnit = (C_UNIT*)(pVS->NextItem(IT_UNIT, pos) );
 } /* End of while() */
} /* End of CVfsmView::CreateIOH */
```

The two variables *m_apIOHandler* and *m_numbIOHandlers* have to be kept as member variables in the view class for later use. The while-loop fetches all the IO-Handlers from the database. This application knows two types of IO-Handlers: *"DI16DO8"* and *"AI8AO2"*. To know means that within the IOH-library there are the corresponding classes *CIO_HandlerDi16Do8* and *CIO_HandlerAi8Ao2*. For every C_UNIT object found in the database the cooperating CIO_Handler (the appropriate subclass resp.) is instanciated with *new*. The virtual method *Create()* makes the link between these two objects. There are other activities in *Create()* depending on the IO-Handler type. Typically input-type IO-Handlers grab for the pointers to all their input items for faster access at runtime. Then (if *Create()* was OK) the CIO_Handler's virtual method *Connect()* is called. The content of this method also depends on the IO-Handler type. Typically the output items (C_DO and C_NO) are connected to this IO-Handler object and told which IO channel number they are.

The VFSM Executor starts in the RTDB before the IO-Handlers are created. It means that the state machines are initialized without knowing the present inputs. This must be considered during the state machines specification by introducing an initialization state which assures that the true start of a system of state machines occurs when the IO-Handlers are operating. Note that delaying the start of the VFSM Executor is not a better alternative: in such a case the system will lose the initial events generated by IO-Handlers.

System Shutdown

A system shutdown always occurs when the application is closed (some applications are able to remove the current database and to build up a new one without dissolving themselves). The VFSM system shutdown is done in two stages:

1. Remove the IO-Handler
2. Remove the Real-time Database

Remove the Real-time Database can be done in two ways. The simple case is when the whole application disappears. Then the instance of CVfsmSystem disappears. Its destructor is able to remove the database with all its items. The second one is used, if the running application must remove the database. For that CVfsmSystem has the method *RemoveAll()*. It does the same as the destructor.

Remove the IO-Handler is usually too complex to be done in the CIO_Handler's destructors. Instead they have a virtual method *Destroy()*. This method can be used to destruct local data structures and to stop and remove the polling threads. The following example shows a procedure *DeleteIOH* that wraps the IO-Handler remove code.

```
//-------------------------------------------------------------
  void CVfsmView::DeleteIOH(void)
//-------------------------------------------------------------
{
bool        bOk;

 for (int i=0; i<m_nNumbIOHandler; i++)
 {
  bOk = m_apIOHandler[i]->Destroy();
  delete m_apIOHandler[i];
  m_apIOHandler[i] = NULL;
 } /* End of for(i) */
 m_nNumbIOHandler = 0;
}
```

We have kept the two variables *m_apIOHandler* and *m_numbIOHandlers* as member variables in the class. The for-loop goes through all existing IO-Handler objects and calls their method *Destroy()*, then the object itself is deleted.

Input-Type IO-Handler

In this section we will follow the path of data from the hardware to the input items in the database. A threaded input-type CIO_Handler subclass will be explained in detail.

Input Handlers

Input Handlers are classes derived from the superclass CIO_Handler. They inherit the methods ***GetUnitName()***, ***GetUnitPhysicalAddress()***, ***GetUnitCommPort()*** and ***GetUnitTypeName()***. Typically the virtual method ***Create()*** must be overwritten. In the following we look at an example of an Input Handler that takes 16 digital inputs packed in a 16 bit word. The example was already mentioned in the previous sections. This unit will be able to act as an input and also as an Output Handler. For the moment we just focus on its behavior as an Input Handler. First the declaration of the class:

```
#include "vsiou.h"
#include "vsdi.h"
#include "vsdo.h"

class CIO_HandlerDi16Do8 : public CIO_Handler
{
virtual bool Create (C_UNIT* pUnit);
virtual bool Destroy (void);
// ..
// Attributes
protected:
 HANDLE  m_hPollThread;
 DWORD   m_dwPollThreadID;
public:
 bool    m_bRunThread;         // if false stop the thread
}; /* End class CIO_HandlerDi16Do8 */
```

In the Create() method we just create the thread IOUnitPollThread. It gets a pointer to the IO-Handler class instance itself ((LPVOID) this). So it has the free access to the member variables. If the thread is correctly created the member variable m_bRunThread is set to true, so that the thread will keep on looping.

```
bool CIO_HandlerDi16Do8::Create (C_UNIT* pUnit)
{
bool bOk = CIO_Handler::Create (pUnit);
 if (!bOk) return false;
 m_bRunThread = true;
 m_hPollThread = CreateThread (
                     (LPSECURITY_ATTRIBUTES) NULL, 0,
                     (LPTHREAD_START_ROUTINE) IOUnitPollThread,
                     (LPVOID) this, 0, &m_dwPollThreadID );
 if (m_hPollThread == NULL)
 {
  m_bRunThread = false;
  return false;
 }
 return true;
} /* End of CIO_HandlerDi16Do8::Create */
```

Before looking to the thread itself we have a look at the method *Destroy()*. It is called by the application framework at system shutdown.

```
bool CIO_HandlerDi16Do8::Destroy (void)
{
DWORD  dwWaitResult;
 m_bRunThread = false;
 dwWaitResult = WaitForSingleObject(m_hPollThread, INFINITE);
 if (dwWaitResult != WAIT_OBJECT_0) return false;
 CloseHandle(m_hPollThread);
 return true;
} /* End of CIO_HandlerDi16Do8::Destroy */
```

First *m_bRunThread* is set to false. This will make the thread leaving the polling loop and finish. Because the thread runs asynchronously one has to wait until the thread is really finished. This is done by the kernel routine *WaitForSingleObject()*. It waits[5] until the thread is finished. Then the thread is killed.

The following lines show the code of the polling thread. It first grabs the polling time parameter and sets the variable *dwPollTime* that is used within the *Sleep()*. Then it fetches the pointers to its DI objects and stores them in an array for faster access at runtime. In the while loop the hardware digital inputs are read and compared with the old ones. The changed bits are set to the according DI items.

```
DWORD FAR PASCAL IOUnitPollThread (CIO_HandlerDi16Do8*
pIOHandler)
{
float  fPollTime;
DWORD  dwPollTime = 2000L; // Default Value 2 sec
CItem* pItem;
C_DI*  apDI[16];           // pointers to the unit's DIs
int    nPhyAdr = pIOHandler->GetUnitPhysicalAddress();
WORD   wDIs, wOldDIs, wChgDIs;

// Get the poll period time from the IO-Handler's C_UNIT item
 pItem = pIOHandler->GetAssItem(DIOB_PollTime);
 if (pItem != NULL)
 {
  fPollTime = pItem->fGetData();
  if (fPollTime > 0)          // fPollTim[sec], dwWaitTime[ms]
   dwPollTime = (DWORD)(1000*fPollTime);
 } /* if (pItem != NULL) */

// Get the pointers to the DIs for faster access
 for (int i=0; i<16; i++)
  apDI[i] = (C_DI*) pIOHandler->GetAssItem(DIOB_Di0+i);
 wChgDIs = 0;
 while(pIOHandler->m_bRunThread) // Endless polling loop
 {
 // fetch DI values at nPhyAdr and put to wDIs
  wChgDIs = wDIs ^ wOldDIs;
  if(wChgDIs)
  {
   // put the changed bits to the DI items
   wOldDIs = wDIs;
  } /* End of if(wChgDIs)
  Sleep(dwPollTime);
 } /* End of while() */
 return true;
} /* End of Thread IOUnitPollThread */
```

[5] To wait means that our thread is removed by the scheduler from the ready queue and put into the wait queue.

The symbols *DIOB_PollTime* and *DIOB_Di0* are delivered by the state*WORKS* Studio by defining a unit. This example uses the following unit declaration. It is an extract from the Configuration File delivered by the state*WORKS* System Configuration (*.swd file):

```
UNIT Name = "IOUnit1"
      Type = "DI16DO8"
      CommPort = "-"
      PhysAddr = 1
      PollTime = "IOUnit1PollTime"
      Di0  = "IOUnit1_Di0"
      Di1  = "IOUnit1_Di1"
      . .
 Di15 = "IOUnit1_Di15"
```

The name *IOUnit1* is typically not used by the IO-Handlers. It only serves as an access key in the VFSM System's database. The type is *DI16DO8*. It is used to create the according IO-Handler (see CreateIO()). *CommPort* is not used here. The *PhysAddr* is used to address the input register. The parameter item *PollTime* allows determining the poll frequency via the VFSM System's database. For this unit the state*WORKS* Studio produces the following include file DI16DO8.h. We can use this enumeration as index to the associated items:

```
typedef enum
{
      DIOB_PollTime = 1,
      DIOB_Di0,
      DIOB_Di1,
      . .
      DIOB_Di15
} ObjectID_DIO;
```

In this unit instance *IOUnit1* the by *DIOB_Di1* indexed associated item is the DI item with the name *IOUnit1_Di1*.

Now let's have a closer look at the poll thread's while-loop. First the 16 DI values are read from the hardware in a form of 16 bits in the word *wDI*. How this works in detail is not part of this manual. Then the read DI values are processed if they changed at all since the last poll cycle. To process the DIs means to send the bits that have changed to the corresponding DI items in the database. The code for that could look like this:

```
  for(int i=0; i<16; i++)
  {
   if ( (apDI[i]) && BitSet(wChgDIs,i) )
     apDI[i]->SetValue( BitValue(wDIs,i) );
  }
```

The for-loop goes through all the 16 bits. If one has changed and if it has a DI item in the database (the pointer to it not NULL) the DI item is updated. That's what DI item's method *SetValue()* is for.

Output-Type IO-Handler

In this section we will follow the route of a control information item (an event) from an output item in the database to the output hardware. The Output Handlers are of special interest here.

Output Handlers

Output Handlers are also classes derived from the superclass CIO_Handler. Output Handlers must overwrite the virtual methods ***Connect()*** and ***SetOutput()***. In the following we look at an example that has both, digital inputs and outputs. We take the same example as before and change it so that it has an additional 8 digital outputs packed in 8 bits of a word. At the moment we just focus on the behavior as an Output Handler. First we extend the description of our IO-Handler so that it includes 8 digital outputs. With the state*WORKS* Studio the unit declaration is changed as follows:

```
typedef enum
{
      DIOB_PollTime = 1,
      DIOB_Di0,
      DIOB_Di1,
      . .
      DIOB_Di15
      DIOB_Do0,
      DIOB_Do1,
      . .
      DIOB_Do7
} ObjectID_DIO;

And the configuration could look like this:
UNIT Name = "IOUnit1"
      Type = "DI16DO8"
      CommPort = "-"
      PhysAddr = 1
      PollTime = "IOUnit1PollTime"
      Di0  = "IOUnit1_Di0"
      Di1  = "IOUnit1_Di1"
      . .
      Di15 = "IOUnit1_Di15"
      Do0  = "IOUnit1_D0"
      Do1  = "IOUnit1_D1"
      . .
      Do7  = "IOUnit1_D7"
```

For instance, the *DIOB_Do1* indexed associated item is the DO item with the name *IOUnit1_Do1*.

Connect the Output Items to an Output Handler

The IO-Handlers virtual method *Connect()* is called at startup. The units have here the opportunity to connect to their associated items. Typically this only makes sense for output-type items like digital or analog outputs. So only output type units have to implement the *Connect()* method. For our example it looks as follows:

```
void CIO_HandlerDi16Do8::Connect (void)
{
C_DO* pDo;
 for (int i=0; i<7; i++)
 {
  pDo = GetAssItem(DIOB_Do0+i);
  if (pDo) pDo->Connect(this, i)
 }
} /* End of CIO_HandlerDi16Do8::Connect */
```

The for-loop goes over the number of DOs of that unit. It fetches every pointer and calls the DO item's method *Connect()*. The enumeration *DIOB_Do0* from the DI16DO8.h-file is taken as a base of all DOs in this unit. *Connect()* takes the pointer to this IO-Handler and the index of the DO item within the IO-Handler. This way every DO item will know in case it changes its value at runtime which IO-Handler it has to inform and which index it is.

Output Item to Output-Handler

It starts when an output item (digital or numerical output, C_DO, C_NO) is called (by the VFSM System) to set an output value. Output items keep as a member variable a pointer to the IO-Handler they belong to and the channel number they have in that IO-Handler. The IO-Handlers have the following forms of the virtual method *SetOutput()*:

```
public:
  virtual void SetOutput (bool  bVal, int nChan);
  virtual void SetOutput (short nVal, int nChan);
  virtual void SetOutput (long  lVal, int nChan);
  virtual void SetOutput (float fVal, int nChan);
  virtual void SetOutput (int nChan);
```

This is a part of the CIO_Handler class declaration. It shows that output items have the choice to call one of these methods to set their value to the real output. With pointer (of the type CIO_Handler) and channel number that they have as a member variable they find the right IO-Handler and there the right output channel. There are four data formats (bool, integer, long, float) that cover all kinds of output data. The last *SetOutput()* method triggers only a channel – the data must be supplied in the IO-Handler (got from RTDB, calculated, etc., must be used with of *UseSimpleGetOutput()*, see the description of the C_NO class).

Output-Handler to IO-Hardware

Output-Handlers are like the Input-Handlers: user written derivations of the superclass CIO_Handler. Output-Handlers must contain the implementation of the methods ***Connect()*** and at least one of the ***SetOutput()***. The following code shows as an example a *SetOutput*() implementation. Our IO-Handler shall be able to handle 8 DOs. It maps them to a word and sends them in this form to the IO-Hardware:

```
void CDOUnit16Packed::SetOutput (bool bVal, int nChan)
{
 if(bVal) m_wAllDOs |=  (1<<nChan);
 else     m_wAllDOs &= ~(1<<nChan);

// put the DO values m_wAllDOs at nPhyAdr
} /* End of CDOUnit16Packed::SetOutput */
```

The IO-Handler has a member variable *m_wAllDOs* that stores the 8 DO's value it has sent the last time. The first two lines of the routine body set or reset the bit indexed by *nChan*. DOs are boolean values. At the end the new DO pattern is sent to the IO-Hardware. This depends on the kind of hardware and is not part of this manual.

C H A P T E R 4

User Written Output Functions

This chapter describes how to write output functions and how to integrate them to a VFSM System. We call them output functions because they are called by a Vfsm's virtual output. In the following we shall talk about output functions and we always mean user-written output functions.

Output functions are represented in the database by the class C_OFUN. C_OFUN item object contains a pointer to a user-written output function of the library OFU.LIB. A C_VFSM item object calls an output function by calling the C_OFUN item's virtual method ***SetValue()***. This method then calls via its function pointer the appropriate output function. We look at two aspects separately:

1. Output functions embedded in the VFSM System
2. Writing the output functions code

How to write Output Functions

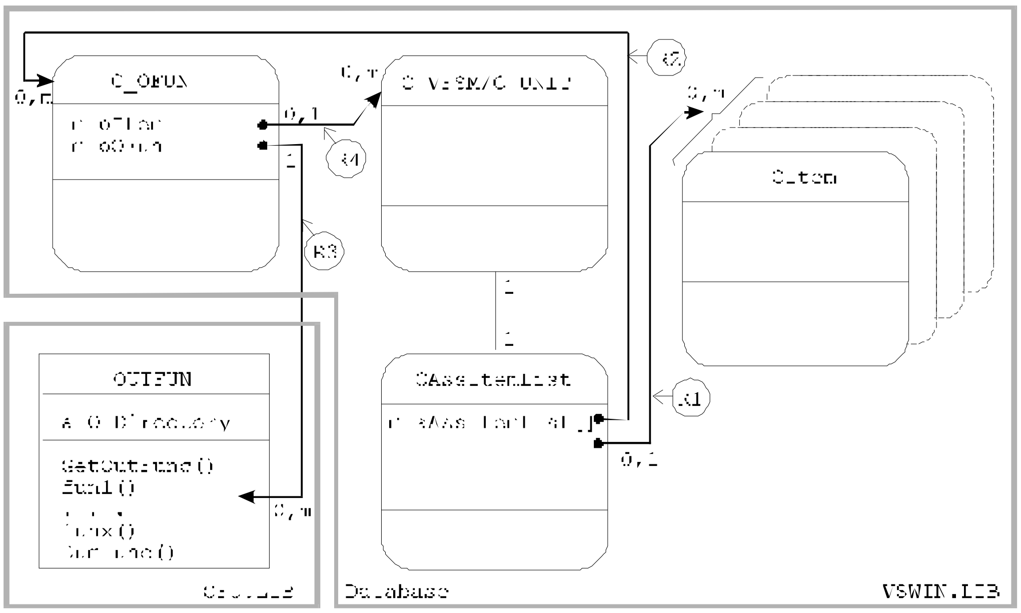
To write an output function is a programming task. It is done in the following four steps:

1. Write the output function's code in the C++ language.
2. Add the function with a name to the Output Function Directory.
3. Compile and build the library OFU.LIB
4. Link the whole application together with the VFSM System (VSWIN.LIB), the IO-Handler (IOH.LIB) and the output functions (OFU.LIB).

In the following sections there is a detailed description with an example of the first two steps. The last two steps depend on the programming environment and are beyond the scope of this manual.

Output Functions embedded in the VFSM System

The following picture shows the connection between the VFSM System's Real-time Database and the (user written) output function library OFU.LIB. OFUN is the module that contains the function directory; it is not a class in the C++ sense.



R1, R2	A C_VFSM or C_UNIT item object has (via a CAssItemList object) several item objects as its associated items. One or more of them is of type C_OFUN (R2).
R3	A C_OFUN item object has a pointer m_pOFun to a function in the OFU.LIB. An output function can be pointed by several C_OFUN item objects.
R4	A C_OFUN item object has a pointer m_pItem to a C_VFSM or C_UNIT item object (typically it's the one that owns the C_OFUN object). Via this pointer it has access to a (the) Vfsm’s or Unit's environment.

Important An output function can be referenced (and called) by several C_OFUN item objects. This is because the calls to the output function work with the individual environment represented by the pointer to the owner item. If a control system has several C_VFSM's of the same type that use a special output function, every instance of them must have its own C_OFUN item object. These C_OFUN item objects point all to the same output function.

System Startup Phase

The startup is divided itself into different phases. For the C_OFUN together with C_VFSM or C_UNIT items the phases Create and Connect are of special interest.

Create

During the create phase the items are configured according to the Configuration File produced by the state*WORKS* System Configuration. The following example shows a piece of configuration for a C_VFSM item that uses a C_OFUN and of a C_OFUN that uses that Vfsm's environment:

```
VFSM Name     = "ObjTest"
     Type     = "objtest"
     Cmd      = "ObjTest-Cmd"
     Tim      = "Tim1"
     No       = "Unit7-Ao4"
     Par      = "Par1"
     OFun     = "StepOFun"

OFUN Name     = "StepOFun"
     FuncName = "SetStep"
     UnitName = "ObjTest"
```

A C_VFSM item object with name *"ObjTest"* has among others the associated item *OFun* with the name *"StepOFun"*. The next block describes such an item: the name of the user written output function shall be *"SetStep"* and has access to the Unit or Vfsm with the name *"ObjTest"*, what is the above one. This means, the output function *"SetStep"* can work with its items, e.g. the C_NO item *No* with the name *"Unit7-Ao4"*.

Alternatively, a special C_UNIT items can be created for an output function. The C_UNIT item should contain all items which the output function is to use.

Connect

The C_OFUN item's method ***Connect()*** passes the output function name to the global procedure ***GetOutFunc()*** of OFU.LIB and gets the pointer to the appropriate function and stores it in *m_pOFun* for later fast access. Then it passes the unit name to the database manager (see CItemList) and gets the pointer to the C_VFSM or C_UNIT item object and stores it in *m_pItem* also for later fast access.

Output Functions at Runtime

At runtime a C_VFSM item object performs several actions (Entry-, Input- and Exit-Actions) according to its state table and virtual output. According to its IO description (see *.IOD) a virtual output (VO) is assigned to a database item type and to a value. When the Vfsm performs a VO that is assigned to a C_OFUN item object, then the specified output function is called. Output functions have the following form:

```
int PASCAL funx(CItem* pOwner, int nVOVal)
```

pOwner typically points to the owner Vfsm. So within this function we have access to that environment.

Virtual Input, Virtual Output

The state*WORKS* Studio lets you to define input and output names and values for an IO object of type C_OFUN. The output value is passed to the output function in the function parameter *nVOVal*. The return value of an output function goes back to the calling C_OFUN object and represents that item's control value. And this control value again can lead to a virtual input of the owner Vfsm. This way a user written output function is connected to the VFSM System in two directions: it gets control information from the owner Vfsm in a form of VOs and it can produce inputs to the owner in a form of VIs.

Output Functions at System Shutdown

When a C_OFUN item is destructed it calls its output function with *nVOVal* as -1. This gives the output function the chance to remove a thread or a dynamically allocated data structure.

Writing the Output Function Code

There is a file OFU.H that contains the declaration of the output functions. The parameter list and the body of an output function must look like this; otherwise it cannot be added to the Output Function Directory:

```
//-------------------------------------------------------------
  int Func1 (CItem* pOwner, int nVOVal)
//-------------------------------------------------------------
{
// Function body
 return nVIVal;
}
```

pOwner	Pointer to the C_VFSM (or C_UNIT) item object the function can work with. Typically this is the owner Vfsm, but it could be every other Vfsm or Unit.
nVOVal	Value the virtual output was assigned to. -1 means that the according C_OFUN item was destructed. pOwner is possibly not valid anymore.

The return value is copied to the caller C_OFUN item object's internal value and can this way produce a virtual input to the Vfsm that owns the C_OFUN item object.

Remarks The corresponding implementation file OFUN.CPP must not be changed

Adding the Output Function to the Directory

In the same file OFUN.H together with the output declaration there is the following data structure:

```
//---------------------------------------------------------------
// Output Function Directory

    r_OFDirectory a_OFDirectory[] =
    {
     "Func1",       Func1,
     "Func2",       Func2,
     "Func3",       Func3,
     "SetStep",     SetStep,
     "GahLeakRate", GahLeakRate
    };
//===============================================================
```

Every output function has to be entered here with a name in string representation and its function name (the pointer to it). The string is the same as entered in the C_OFUN item object's configuration:

```
OFUN    Name        = "StepOFun"
        FuncName    = "SetStep"
        UnitName    = "StepTest"
```

Typically you use the same string representation as the function name for better understandability, but this is not necessary; you could choose different names. At System Startup (Connect Phase) every C_OFUN item object calls the ***GetOutFunc()*** function (also placed in OFUN.H). This function goes through the data structure *a_OFDirectory* and compares the function names. If a function is found, it returns the pointer to the appropriate output function. If not it returns the pointer to a dummy function that just fulfills the call with doing nothing.

Example

The following example is one of the more complex ones, and interesting because output functions are used to cause inputs to be processed.. It is used to compute the leak rate in a vacuum chamber. This is done in three steps:

1. store the start pressure,
2. wait a specified time or a specified pressure increment (whatever happens first),
3. compute the difference between the current and the stored pressure and divide it by the elapsed time and by the vacuum chamber volume. Store the result in a C_DAT item for later use.

The output function shall work in the first and the last step. We assume that the Vfsm calls the output function once via a certain virtual output and a value at the leak rate start and once at the end via another virtual output and value. These values have to be assigned to the appropriate virtual outputs with the Vfsm's IO description (see the corresponding IOD-File produced by the state*WORKS* Studio).

Before starting with the output function we will have a look at the configuration of the C_VFSM item object and at its C_OFUN item object:

```
OFUN Name     = "PHGah-OFuLeakRate"
     FuncName = "GahLeakRate"
     UnitName = "PHGah"

VFSM Name           = "PHGah"
     Type           = "elegah"
     MyCmd          = "PHGah-Cmd"
     TiLeakCheck    = "PHGah-TiLeakCheck"
     TiLeakDelay    = "PHGah-TiLeakDelay"
     AlLeakRate     = "PHGah-AlLeakRate"
     AiIKR          = "PHGah-Ai"
     SwipIKR        = "PHGah-SwipIkr"
     SwipLeak       = "PHGah-SwipLeak"
     OFULeakRate    = "PHGah-OFuLeakRate"
     EPChambVol     = "PHGah-EPChambVol"
     EPLeakCheck    = "PHGah-EPLeakCheck"
     EPLeakDelay    = "PHGah-EPLeakDelay"
     EPLeakDiffPres = "PHGah-EPLeakDiffPres"
     EPMaxLeakRate  = "PHGah-EPMaxLeakRate"
     DatActLeakRate = "PHGah-DatActLeakRate"
     DatStartPres   = "PHGah-DatStartPres"
```

The C_OFUN item object has as *UnitName* the *"PHGah"*, what means, it works with that item objects. The stateWORKS Studio produces the declaration file ELEGAH.H with the typedef:

```
typedef enum
{
      GAHB_MyCmd = 1,
      GAHB_TiLeakCheck,
      GAHB_TiLeakDelay,
      GAHB_AlLeakRate,
      GAHB_AiIKR,
      GAHB_SwipIKR,
      GAHB_SwipLeak,
      GAHB_EPChambVol,
      GAHB_EPLeakCheck,
      GAHB_EPLeakDelay,
      GAHB_EPLeakDiffPres,
      GAHB_EPMaxLeakRate,
      GAHB_OFULeakRate,
      GAHB_DatActLeakRate,
      GAHB_DatStartPres
} ObjectID_GAH;
```

The file ELEGAH.H is included in the OFUN.H. So we can access the associated items via the enumeration.

Be aware that every time the IO dictionary of the Vfsm type "elegah" is changed the OFUN.CPP as well as the implementation file named for instance ELEGAH.CPP has to be compiled and linked again. Otherwise the enumeration could point to the wrong items.

Let's assume that the file ELEGAH.CPP contains the implementation. First we declare the pointers to the selection of item objects:

```
#include "elegah.h"
//-----------------------------------------------------
  int GahLeakRate (CItem* pOwner, int nVO)
//-----------------------------------------------------
{
 if (nVO == -1)  return 0; // need no destruction
}
```

This line of code checks if the according C_OFUN item is destructed. Here we have nothing to do in this case, so we leave the function

```
 if (pOwner->GetUnitTypeName() != "elegah") return 0;
```

This line of code is for careful programmers. It checks whether the output function works with the correct Vfsm type. If it is not the case it returns zero which could be analyzed by the calling Vfsm via its virtual input. In the correct case we let the output function return 1.

```
C_TIM*  pTiLeakCheck;
C_NI*   pAiIkr = (C_NI*)( pOwner->GetAssItem(GAHB_AiIKR));
C_SWIP* pSwipLeak;
C_PAR*  pEPChambVol;
C_PAR*  pEPLeakDiffPres;
CData*  pDatActLeakRate;
CData*  pDatStartPres =
          (CData*)( pOwner->GetAssItem(GAHB_DatStartPres));
float   f;
int     nLeakTime;
float   fDiffPres, fLeakRate, fChambVol;
```

The pointers to the seven item objects are declared here. The types are defined in the appropriate IOD-File. The *pAiIkr* and *pDatStartPres* pointers are fetched already here, because we need them anyway. This is done by calling the *pOwner* object's method *AssItem()* with the index of the appropriate item. The returned pointer is of type CItem; we have to cast it to the according item type.

```
 switch (nVO)
 {
  case 1: // Leak Start: code of leak rate start
  break;
  case 2: // Leak Calc: code of leak rate calculation
  break;
  case -1: // possibly destroy dynamic data
  default: return 0;
 } /* End of switch */
 return 1;
} /* End of GahLeakRate */
```

The switch statement selects one of the virtual output values, 1 for the leak rate start and 2 for the leak rate end respectively its calculation. For both of these values we return 1 which signals OK to the Vfsm. Other values of *nVO* are ignored and acknowledged with a zero return value.

A special *nVO* value is *-1*. The output functions are called with this value at system shutdown to give them the chance to remove possibly created dynamic data or threads. Our output function here does not need this feature.

Here is the complete code of the first case section:

```
case 1: // Leak Start
 pSwipLeak = (C_SWIP*)(pOwner->GetAssItem(GAHB_SwipLeak));
 pEPLeakDiffPres =
      (C_PAR*)(pOwner->GetAssItem(GAHB_EPLeakDiffPres));
 f = pAiIkr->fGetData();
 pDatStartPres->SetData(f);

 f = f + pEPLeakDiffPres->fGetData();
 pSwipLeak->SetLimits(f, f);
break;
```

The first two lines fetch the pointers to the rest of the used items, a switch point and a parameter specifying the maximum pressure difference. The next two lines fetch the current pressure and store it to a C_DAT item. The last two lines compute the absolute end pressure and set it to the switch point (limit low and high are equal because we only need the LOW and HIGH SWIP states).

The Vfsm will now wait until either the timer expires or the pressure limit is reached. Then it probably will change its state and call the output function again, but with a virtual output value of 2. Then the following case section will be executed:

```
case 2: // Leak Calc
 pTiLeakCheck =
       (C_TIM*)(pOwner->GetAssItem(GAHB_TiLeakCheck));
 pEPChambVol  =
       (C_PAR*)(pOwner->GetAssItem(GAHB_EPChambVol));
 pDatActLeakRate =
       (CData*)(pOwner->GetAssItem(GAHB_DatActLeakRate));
 fDiffPres = pAiIkr->fGetData() -
             pDatStartPres->fGetData();
 nLeakTime = pTiLeakCheck->GetCountRegister();
 fChambVol = pEPChambVol->fGetData();
 if( (nLeakTime != 0) && (fChambVol != 0.0) )
  fLeakRate = fDiffPres / nLeakTime / fChambVol;
 else
  fLeakRate = (float)0.0;
 pDatActLeakRate->SetData(fLeakRate);
break;
```

First the pointers to the items used here are fetched again. Then we fetch the current timer and current and start pressure values and compute the pressure difference (we need the current values because we don't know what the reason for the end leak check was: the timer, the pressure or something else). The leak rate is computed according to the leak rate formula. Of course we try to make everything as safe as possible, so we test the two divisors first for zero, to get no exception during runtime. At the end the value is stored to the appropriate C_DAT item object, either for display only or more probably to test the value by means of another switch point (SWIP).

C H A P T E R 5

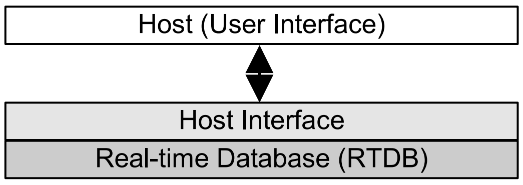
The Host Interface to the RTDB

This chapter describes the VFSM System as a server application. The interface to a client from the RTDB point of view is called the Host Interface.

Only the general behavior of the host interface is described in this chapter. The item specific behavior is described in part 2 for every item type separately.

Introduction

The host interface was mentioned the first time in the description of the RTDB layer model as something that can produce events to, or get events from the RTDB. Here we focus on what is exactly between the Host and the Host-Interface.



The interface between a host and the RTDB's Host-Interface is of client-server type. The RTDB is the server and a host is a client. The RTDB is able to serve an almost infinite number of clients at the same time with different protocols. At the time of writing a TCP/IP based message protocol and DDE interface (Direct Data Exchange, only for Windows based RTDB) are available. Others may follow in the future.

There are the following access types:

- Request/Reply: The client asks something and gets an answer from the RTDB.
- Poke: The client changes something in the RTDB.
- Event: The client gets informed that something changed in the RTDB. This of course only happens under the client's control with AdviseStart/Stop requests.

Accessible objects are all the Database Items, the RTDB items created during startup according the configuration file. Additionally, there is a Database Manager with general objects holding information regarding the behavior of the whole VFSM system.

Database access is organized as a key and an attribute. The key is the name of a certain database object. Depending on the object type various attributes are available. The representation of the key, attribute and value is protocol dependent. The TCP/IP message protocol represents the name as a string, the attribute as a number and the values again as strings. The DDE interface packs name and attribute into one string and the value into another string.

Attributes

The set of attributes is predefined. Not all attributes are applicable to all item types. The following list shows all the available attributes.

Attribute Name	**Number**	**Short**	**Comment**
None	0		State or value of an item as number
Value	1	Val	ditto
ServiceMode	2	SvM	Boolean value, service mode on/off
ServiceValue	3	SvV	Value, format depending on item type
PeripheralValue	4	PeV	ditto
VI	5	VI	Virtual input of a VFSM item
StateName	6	StN	Same as Value, as name
AssocItemList	7	AIL	Items which belong to a VFSM or UNIT
TypeName	8	Typ	E.g. the definition file name of a VFSM
CountConstant	9	CnC	E.g. time constant in a timer item
CountRegister	10	CnR	E.g. current time of timer item
Category	11	Cat	Additional information for alarm and parameter items
Format	12	Frm	The data format of a data item
PhysicalUnit	13	Uni	The unit (e.g. V for voltage)
LimitLow	14	LiL	Used in switchpoint and param items
LimitHigh	15	LiH	ditto
InitValue	16	IVa	Used for parameter items as start value
DataValue	17	Dat	The content of a data item
Text	18	Txt	Alarm text of a ALA item
Acknowledge	19	Ack	Acknowledge a pending alarm
Time	20	Tim	Time when an alarm occurred
ScaleFactor	21	ScF	Used in NI and NO items
Offset	22	Ofs	ditto
ScaleMode	23	ScM	ditto
List	24	Lst	For example a list of all VFSM state names
PhysAddr	25	PAd	Used in IO-Handlers, e.g. port number
CommPort	26	Com	ditto - could be a TCP/IP address, or a serial port name
Trace	27	Trc	Turn on/off tracing of an item
RunMode	28	Rmo	Put VFSM in Free/Step/Hold-Mode
NextStep	29	NSt	Name of the next possible transition

Remarks

The attribute number is used in the TCP/IP based messages.
The abbreviation, the short form is used in the Host interface as extention to the item name.
More detailed information will follow together with the description of the item types.

Database Manager

The database manager represents information about the RTDB itself. The objects available are independent of the RTDB configuration; their values are not.

Database Configuration

A client can request name and path of the current configuration file and the total number of all database items. The name is “IL”. No attribute means the config file and the CountRegister-attribute delivers the number of items.

Example:

Name	Attribute	Reply
“IL”	none	“C:\StateWORKS\Examples\Ex1\openclose.swd”
“IL”	CountRegister	“34” (a total of 34 items are in the RTDB)

Database Contents

A client application can request the statistics about certain database items. The name is a string with the abbreviation of the item type. The List-attribute delivers a string with the names of all items of the according type separated by <NL> characters (new line = 0x0A). The CountRegister-attribute delivers the number accordingly.

Example:

Name	Attribute	Reply
“AL”	List	“Alarm1<NL>Alarm2<NL>Alarm3”
“AL”	CountRegister	“3” (a total of 3 alarm items in the RTDB)

Alarm Handler

The database manager maintains an alarm queue. An alarm item object that gets into an active state is appended to this queue. The item with the name "AL" represents the first (and visible) alarm item of the alarm queue. To poke an Acknowledge to that alarm makes it vanish from the queue. The next alarm will be sent as an update. So the client (destination) application can step through all active alarms. The item "AL" has the same attributes as a normal alarm item:

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	Alarm’s state as number, see e_ALA_States
State Name	R	Alarm’s state as a text string
Type Name	R	Name of the alarm item represented by AL
Category	R	Text defined in state*WORKS* Studio/AL-Properties
Text	R	Alarm text defined in stateWORKS Studio/AL-Properties
Time	R	Time and date when alarm's state was entered
Acknowledge	W	Empty message, command to state machine

Example:

Name	Attribute	Reply
“AL”	TypeName	“Alarm2” if Alarm2 is on top of the alarm queue

Item Trace

The VFSM System is able to trace the database item's value changes and put them into a trace file together with a time stamp. For every item, tracing can be turned on and off individually by setting/resetting the trace flag. All item types have on their host interface the read/write attribute ".Trc". A value of "1" means that the trace is on for that item (or "0" for off). Trace can only take place when the trace file is open.

The trace file is opened automatically at system start up under the name "TRACE.TXT"[6] in the same directory as the configuration file. At system stop the file is closed again. To open and close the trace file during system run there is an item with the name "IL" and the attribute Trace:

Attribute	Access	Description
Trace	Write	Send a trace command to the trace manager
Trace	Read	Get the state of the trace file and traced items

The following trace command numbers are possible:

1. Open File. The trace file is opened (if it is not already opened). The activated items can trace their value changes from now on.
2. Close File. The trace file is closed. The activated item cannot trace (information is lost).
3. Reset Trace. Set the tracing of all items off.

Example:

Name	Attribute	Poke	Comment
"IL"	Trace	3	The trace flags of all RTDB items are reset, no trace

The state of trace manager can be requested (no automatic update). The following values are possible:

1. Trace file is open; there are one or more items with activated trace.
2. Trace file is closed; there are one or more items with activated trace.
3. Trace file is open; there is no item with activated trace.
4. Trace file is closed; there is no item with activated trace.

Example:

Name	Attribute	Reply	Comment
"IL"	Trace	"3"	Trace file is open but no trace flag set

[6] The trace file entries are immediately flushed. So the file can be read while the system runs, for instance with the Notebook editor. The resolution is milliseconds as in "10:36:20,009"

Database Items

The behavior of the database items in a host conversation is described in detail in part 2 of this manual. Here follows a table with all the item types and all the attributes they support.

R, W and *X* are the access modes:

R the attribute on this item type is read only (request only).

W is write (poke) only

X is read and write.

All the attributes support the automatic and manual link mode.

Attribute Name	**Short**	**VFSM**	**CMD**	**TI**	**AL**	**DI**	**DO**	**NO**	**NI**	**SWIP**	**XDA**	**PAR**	**OFUN**	**STR**	**CNT**	**DAT**	**UNIT**	**ECNT**	**UDC**	**TAB**
Value		R	R	X	R	R	R	R	R	X	X	R	X	X	X	R		X	X	X
Value	Val	R	R	X	R	R	R	R	R	X		R	X	X	X	R		X	X	
ServiceMode	SvM	X	X			X	X			X										
ServiceValue	SvV	X	X			X	X			X										
PeripheralValue	PeV	R	X			R	R			R										
VI	VI	R																		
StateName	StN	R		R	R			R	R	R		R		R	R	R		R	R	
AssocItemList	AIL	R			R												R			
TypeName	Typ	R	R														R			
CountConstant	CnC			X											X			X		
CountRegister	CnR			R											R			R		
Category	Cat				R							R								
Format	Frm							R	R			R				R			R	
PhysicalUnit	Uni			R				R	R			R				R			R	
LimitLow	LiL								R	X		R								
LimitHigh	LiH									X		R								
InitValue	IVa											R		X						
DataValue	Dat							X	R	R		X		X					R	
Text	Txt				R															
Acknowledge	Ack				W															
Time	Tim				R															
ScaleFactor	ScF							R	R											
Offset	Ofs							R	R											
ScaleMode	ScM							R	R											
List	Lst	R	R																	
PhysAddr	PAd																R			
CommPort	Com																R			
Trace	Trc	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
RunMode	RMo	X																		
NextStep	NSt	R																		

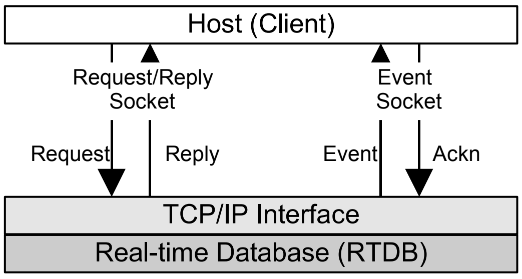
TCP/IP Interface

The design goal was a simple and fast interface, free of more or less platform specific standards like COM, CORBA or XML. The implementation of any sort of client is very easy. The only specific aspect is a message package that is used for all types of transactions. The rest is standard TCP/IP programming.

Architecture

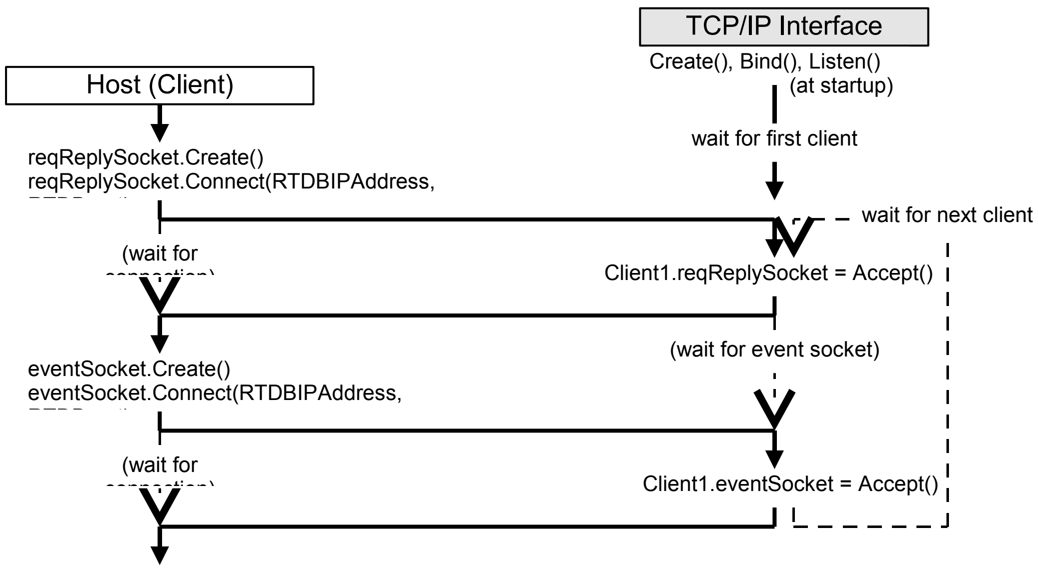
Per client there is one port with two sockets. Port and sockets get opened and closed on client's demand. Of course if the RTDB application exits it closes all open ports. Also if the RTDB recognizes that a client silently disappeared, it closes that particular port.

The sockets are called Request/Reply Socket (RRS) and Event Socket (ES). From the client application point of view the RRS is a synchronous connection. The client sends a request and immediately gets a reply. The ES on the other hand is asynchronous. The RTDB (server) can send a message to the client at any time (the client has to acknowledge it immediately). This type of messages is also called unsolicited messages. They are characteristic of event driven systems.



Establish Connection

The client creates and connects two sockets immediately after each other on the same port. The RTDB uses 9091 as the standard port number.



The previous pseudo code shows the procedure of connecting a port to the RTDB with two sockets. First the RTDB application has to finish initialization. During this it starts up the TCP/IP interface as a server with create, bind and listen. Finally it waits on accept for the first client. Now a client can begin with the connection. It first creates the Request/Reply Socket and connects it to the address and port number of the RTDB application. Then it waits for completion. In the meantime the server accepts and handles the first connection as the RRS. This makes the client go on for the second connection. Same procedure here: Create and connect the same address and port number.

Now the client server connection is established and ready to work. Of course this is quite a simplified description of the client actions. Typically a client application would wrap the event socket into an own thread so that the event mechanism becomes really asynchronous.

Message Packet

The client packs a request into a message packet and sends it to the RRS. It immediately receives the reply (packed into another message packet). If the RTDB has to advise the client about a change on a certain item it packs the according information into a message packet. The client receives it on the event socket and has to return an acknowledge; again in a message packet. The format of the message packet is declared in the file `vstypip.h`. The following excerpt from this file shows the declaration of the message packet itself:

```
struct r_MessagePack{
 int                  nHeader;
 e_MessageType        MessageType;
 e_Topic              Topic;
 e_ItemAttributes     IAtt;
 e_ItemSearchResults  MessageInfo;
 char                 cItemName[NAMESIZE];
 char                 cItemValue[VALUESIZE];
};
```

	Size [bytes]	Description
nHeader	4	This integer has no direct meaning. I can be set to any value in the request packet. The RTDB copies the value into the reply packet.
MessageType	4	The type of message (request, reply, advise..).
Topic	4	Until now the only valid value is IL (2).
IAtt	4	What attribute of the item we are talking about (see description in the previous section), declaration in the file vsystyp.h.
MessageInfo	4	Result of the request (Ok, no such request, no such item, no such attribute), declaration in the file vsystyp.h.
cItemName	64	Item name as array of characters, zero terminated
cItemValue	0..1000	Value of the item attribute; text representation as array of characters; zero (null character) terminated.

Message Type

Describes the kind of message packet. Depends on the kind of request and reply or event.

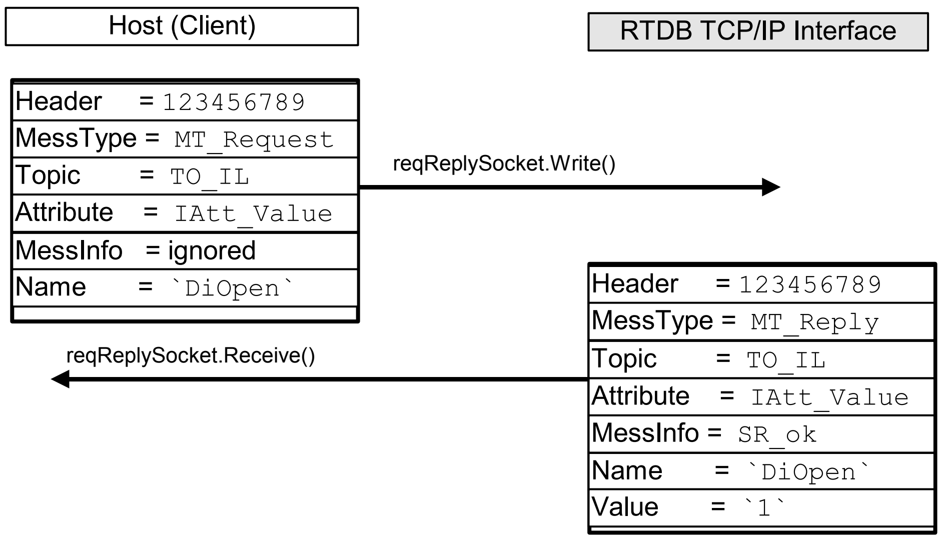
```
enum e_MessageType  { MT_none  = 0,
MT_Request, MT_Reply, MT_Poke, MT_AdvStart, MT_AdvStop,
MT_AdvData, MT_AdvAckn, MT_Disconnect, MT_Last };
```

Scenarios

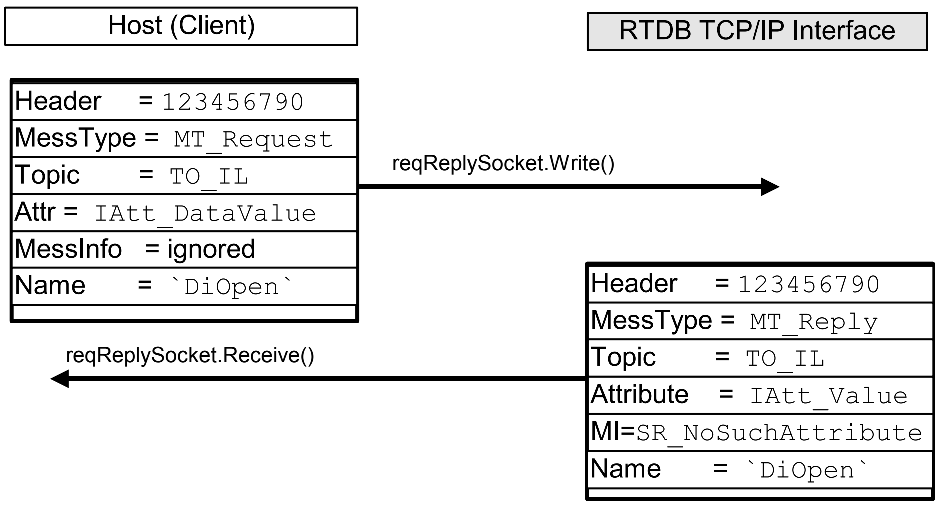
The following section illustrates all types of client/server and server/client transactions (item name, attribute and value are examples).

Request the Value of an Item Attribute

The client wants to know the value of a certain item attribute. It writes the according message packet to the RRS. It can immediately receive the reply from the same socket. The request was correct, so the message info is `SR_ok`.

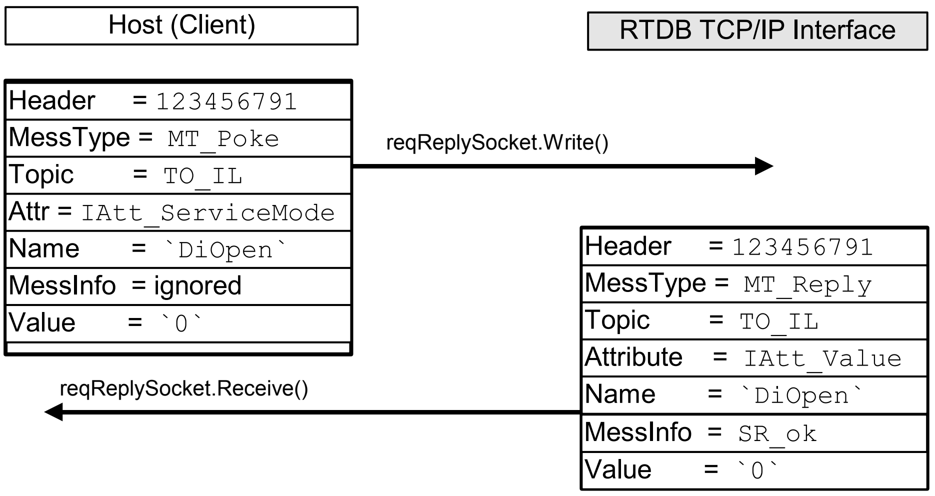


The following example shows a failed request. The particular item does not have the attribute. So the message info is `SR_NoSuchAttribute` and of course there is no value available.



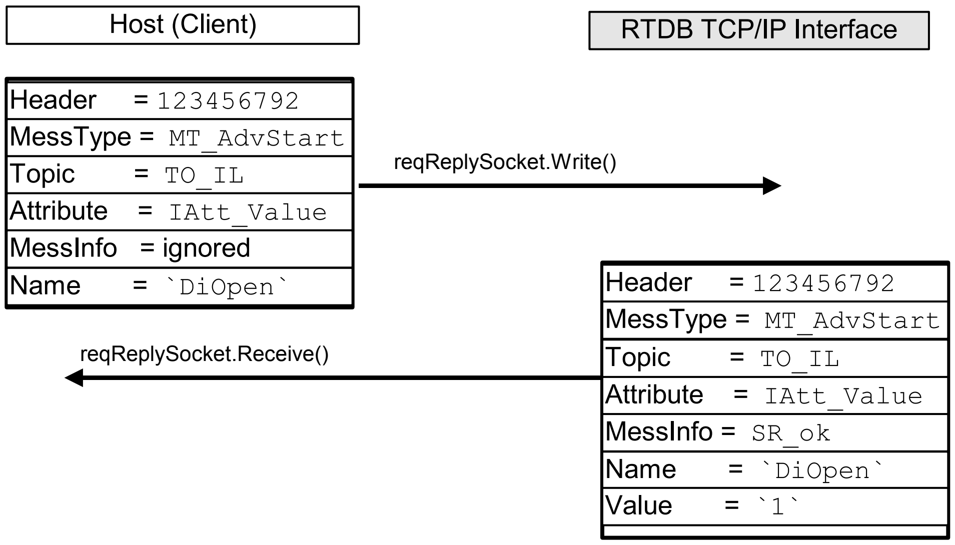
Poke the Value of an Item Attribute

The client wants to change the value of a certain item attribute. It writes a poke message packet to the RRS and can receive the answer that contains the changed item attribute.

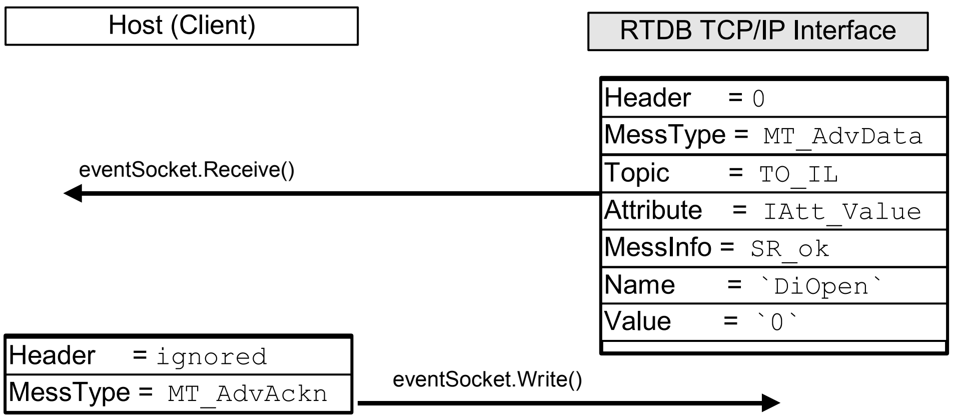


Advise, Event Data and Unadvise

The client wants to get advised when a certain item attribute changes. As a reply it gets the current value of the according item attribute.

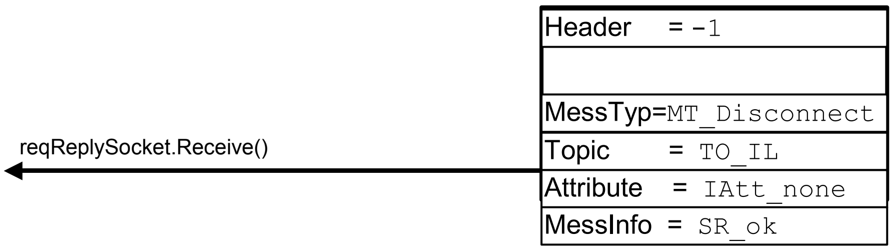


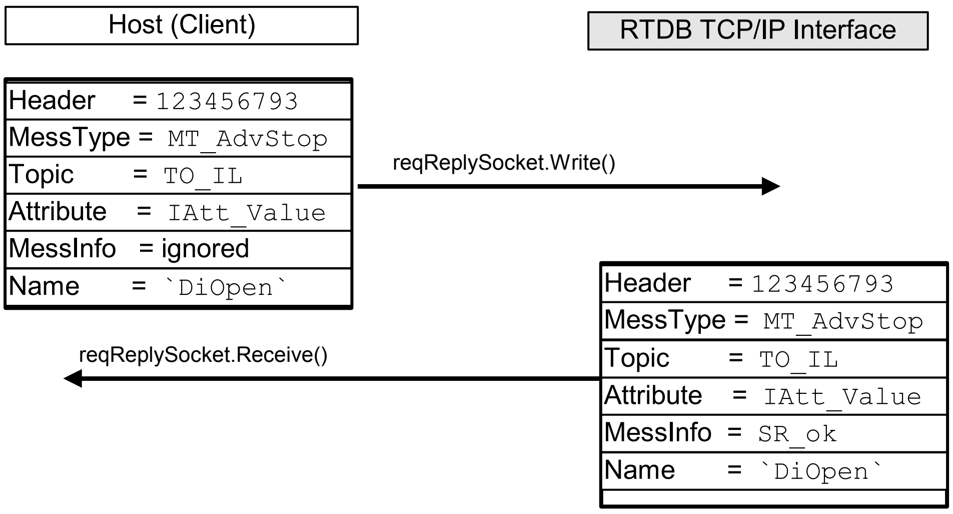
If the advised item attribute changes its value the RTDB sends the change as an event message packet over the event socket to the client.



Now the client wants to be no longer advised about the item attribute.

Disconnect Event





If the RTDB disappears for whatever reason it sends a disconnect event over the event socket to the client. No answer is expected.

Interplatform Connection

The design goal was to get an interface between all types of CPU, programming languages and operating systems. This requires some care in filling in the message packet. The first five fields are binary data that may be interpreted depending on the platform as little- or big-endian. To avoid getting into trouble with data formats TCP/IP socket implementations provide the programmer with a pair of functions like:

`u_long **ntohl** (u_long netlong);` and

`u_long **htonl** (u_long hostlong);`

The **`ntohl`** function converts a `u_long` from TCP/IP network order to host byte order (which is big- or little-endian) and **`htonl`** function converts a `u_long` from host to TCP/IP network byte order (which is always big-endian).

TCP/IP Client

There is a library and a DLL available which hide the complexity of the TCP/IP interface and make the design of clients easy. The following text describes the DLL interface. The library variant is similar and is defined in the *tcpip.h file*.

#include "tcpipclient_dll.h"

Dll functions

Initialize1	Stores the callback function and environment pointer.
Connect1	Creates sockets and connects via the according sockets to the addressed RTDB server port.
Connected1	Returns the value true or false signalling whether the client is connected or not.
Disconnect1	Disconnects the RTDB server port from the sockets.
Request1	Asks for the attribute value of a corresponding RTDB object.
Poke1	Sets the attribute value of a corresponding RTDB object.
AdviseStart1	Asks for advise of a corresponding RTDB object attribute.
AdviseStop1	Asks for unadvise of a corresponding RTDB object attribute.
UnAdviseAll1	Asks for unadvised of all RTDB object attributes.

Initialize1

void __stdcall Initialize1(void (*ReplyOrEvent)(int Rep, const string& stName, const string& pstVal, void* pOwner), void* pOwner);

Remarks Stores the callback function and environment pointer. Used in an object oriented environment to store the this pointer and the pointer to the callback function. The callback function is then called by events arriving from advised RTDB object attributes.

Connect1

bool __stdcall Connect1(char* pstIPAddress = HOST, int nPort = PORTNUMBER);

Remarks Creates sockets and connects via the according sockets to the addressed RTDB server port. The default TCP/IP address is the HOST=LOCALHOST=127.0.0.1, the default port PORTNUMBER = 9091.

Return Value true if connected is successful otherwise false.

Connected1

bool __stdcall Connected1( );

Remarks Returns the value true or false signalling whether the client is connected or not.

Return Value true if the client is connected otherwise false.

Disconnect1

void __stdcall Disconnect1(bool bWithUnAdvise = false);

Remarks Disconnects the RTDB server port from the sockets. If called with parameter bWithUnAdvise=true the function unadvises all RTDB objects attributes.

Request1

```
bool __stdcall Request1( char* pstItemName,
                         e_ItemAttributes eIAtt = IAtt_None,
                         char* pstValue = NULL);
```

Remarks Asks for the attribute value of a corresponding RTDB object.
Return Value true if the Request function is successful, otherwise false

Poke1

```
bool __stdcall Poke1( char* pstItemName,
                      char* pstValue,
                      e_ItemAttributes eIAtt = IAtt_None );
```

Remarks Sets the attribute value of a corresponding RTDB object.
Return Value true if the Poke function is successful, otherwise false

AdviseAtart1

```
bool __stdcall AdviseStart1( char* pstItemName,
                             e_ItemAttributes eIAtt = IAtt_None,
                             char* pstValue = NULL );
```

Remarks Asks for advise of a corresponding RTDB object attribute.
Return Value true if the AdviseStart function is successful, otherwise false

AdviseStop1

```
bool __stdcall AdviseStop1( char* pstItemName,
                            e_ItemAttributes eIAtt = IAtt_None );
```

Remarks Asks for unadvise of a corresponding RTDB object attribute.
Return Value true if the AdviseStop function is successful, otherwise false

UnAdviseAll1

```
void __stdcall UnAdviseAll1( );
```

Remarks Asks for unadvised of all RTDB object attributes.
Return Value true if the UnAdviseAll function is successful, otherwise false

DDE Interface

This interface type only works for Microsoft Windows applications and for the RTDB implemented on Microsoft Windows (NT4.0 and later or in general WIN32). The RTDB delivered in the library RTDB.LIB for Windows integrated on a VFSM System as a WIN32 application is able to work as a DDE source application. With DDE (Dynamic Data Exchange) Windows applications are linked, allowing the transfer of data. DDE establishes a conversation between two applications. The application that initiates the conversation is called the destination. The target of the DDE conversation is called the source.

Remark In this chapter RTDB for WIN32 is called RTDBWin if an issue is DDE specific.

The RTDBWin is the data server and so in a DDE conversation the source. The client or the destination is typically a user interface application for instance written in VisualBasic (VB). The following text assumes that the client has been written in VB.

Name and Topic

To establish a DDE conversion, the destination application must specify the name of the source application, a topic and the item. The application (the WIN32 VFSM System) has the name "VS" and the RTDBWin topic is "IL". So a VB source application would specify for the variables `vfsmList` and `vfsm1VI`:

```
vfsmList.LinkTopic = "VS|IL"
vfsm1VI.LinkTopic = "VS|IL"
```

Item

The DDE item specification for the RTDBWin is a string consisting of the name of a database item, a "." as separator and an extension that specifies a certain attribute of that item. If there were for instance a state machine with the name "Vfsm1" in the database, then the following first VB line would link the Virtual Input of that state machine to the variable `vfsm1VI`:

```
vfsm1VI.LinkItem = "Vfsm1.VI"
vfsmList.LinkItem = "VFSM.Lst"
```

The second line links the list of all vfsm instances to the variable vfsmList. Be aware that the two variables in the VB application have no values yet.

Link Types

The RTDBWin supports the link types manual and automatic. With an automatic link, the destination application is updated automatically whenever the item changes in the database. In manual link mode, the database only sends an update when requested by the destination application.

The first two lines set the link mode manual and force one update of the VB variable `vfsmList`:

```
vfsmList.LinkMode = VbLinkManual
vfsmList.LinkRequest
vfsm1VI.LinkMode = VbLinkAutomatic
```

The last line sets link mode automatic for `vfsm1VI`. From now on this variable is updated whenever the Virtual Input of the state machine “Vfsm1” changes in the RTDB.

A destination application also can change the data of an item in the database with a so-called poke message. The following four VB lines connect the VB variable `vfsm1Cmd` to an RTDB item object “Vfsm1Cmd” and set it to the value “On”:

```
vfsm1Cmd.LinkTopic = "VS|IL"
vfsm1Cmd.LinkItem = "Vfsm1Cmd.Val"
vfsm1Cmd = "On"
vfsm1Cmd.LinkPoke
```

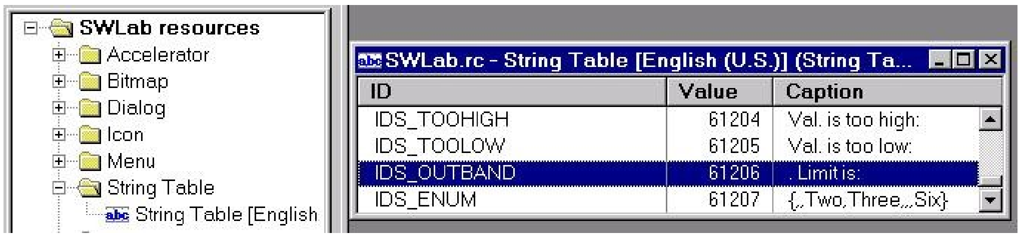
Data Types

The RTDBWin application supports only the text format (CF_TEXT). So, all RTDB formats are represented as strings. For instance, “1.23e-45” stands for a floating number.

CHAPTER 6

Internationalization Support

Text properties (Alarm Text and Data Units) can be set as plain text or as string constants. A string constant is a Windows concept supported in VisualC++[7]. In the project resource there may be a String Table like this:

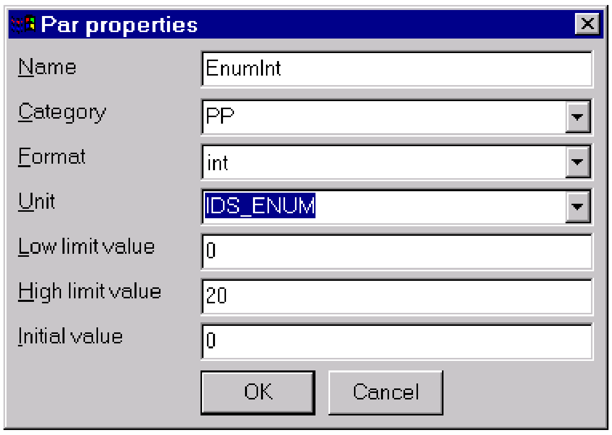


ID	Value	Caption
IDS_TOOHIGH	61204	Val. is too high:
IDS_TOOLOW	61205	Val. is too low:
IDS_OUTBAND	61206	. Limit is:
IDS_ENUM	61207	{,,Two,Three,,,Six}

In this string table all strings used in a project of the VFSM system with the RTDB are entered. Of course the VC++ project has to be rebuilt every time the String Table is changed. VC++ produces a file `resource.h.` This file contains the references to the particular strings like:

```
#define IDS_TOOHIGH                          61204
#define IDS_TOOLOW                           61205
#define IDS_OUTBAND                          61206
#define IDS_ENUM                             61207
```

The file `resource.h` has to be copied to the state*WORKS* application configuration directory (the same that contains the configuration file *.swd). The `IDS_xx` identifiers are used instead of the texts as for instance in the following state*WORKS* project:



Name	EnumInt
Category	PP
Format	int
Unit	IDS_ENUM
Low limit value	0
High limit value	20
Initial value	0

There can be several string tables for different languages. Changing the workstation's language changes the string table and so the strings used by state*WORKS*.

For the VFSM system with RTDB running under a non-Windows operating system (especially UNIX-like) the string constant are taken from a `stringres.src` file which has to be situated together with the `resource.h` file in the application configuration directory (with the *swd file). The `stringres.src` and `resource.h` files are prepared using a translation program *StringRes.exe* which can be found in the state*WORKS* Studio development environment. The translation program generates the `stringres.src` and `resource.h` files using as input strings defined in a text file `StringFile.txt`. For the above example, the content of the `StringFile.txt` file would be:

[7] RTDB for other applications or OS may not allow string constants.

```
IDS_TOOHIGH Val. Is too high
IDS_TOOLOW  Val. Is too low
IDS_OUTBAND Limits
IDS_ENUM    {,,Two,Three,,,Six}
```

P A R T 2

The VFSM System Class Library Reference

CHAPTER 7

VFSM System Class Library Reference

class Citem

The CItem class is the superclass of all database objects. It contains all the data and methods applying to all objects, such as object -name and -value with the methods for the client/server communication. The item value depends on the actual derived item type. For instance, for a DI the value represents the status of the signal line, for a Vfsm it is the state.

#include <vsitem.h>

Runtime Virtual Public Members

SetValue	Sets the CItem object's internal value.
ResetValue	Resets a CItem internal value (typically used for virtual inputs).
GetData	Returns the data value of CItem types that performs a data flow (e.g. C_NI).
GetUnitTypeName	Used by C_VFSM and C_UNIT item types.
GetValue	Returns the internal value of a CItem object.
GetName	Returns the CItem object's name.
GetType	Gets the type of the item as an enumeration (e.g. IT_VFSM).
GetTypeName	Gets the type of the item as a string (e.g. "VFSM").
AddDependent	Adds an item as dependent of another.

General Static Members

GetItemType	Gets the enumeration of an item type in string representation.
GetItemTypeName	Gets the string representation of an item type in enumeration.
GetAttrName	Gets the string representation of an attribute specification.
GetAttr	Gets the enumeration of an attribute specification in string representation.

Member Functions

CItem::SetValue

virtual void SetValue (int nVal);

	nVal Value to set to item internal value.
Remarks	This function is called by other items, by IO-Handlers and by user written output functions to set the item object's specific value. Remark: the kind of value depends on the type of item (see the virtual function of the derived classes). Here, the function sets the items internal value.
See Also	***C_xxx::SetValue***

CItem::GetValue

virtual int GetValue (void);

Remarks	This function is typically used internally to get the item objects internal value. The meaning of the value depends on the type of item (see the virtual function of the derived classes). In CItem types this value has no practical meaning.
Return Value	Item objects internal value (the items's state).
See Also	***C_xxx::GetValue***

CItem::GetName

CString* GetName (void);

Remarks	Gets the name (the key to) of the item object.
Return Value	Pointer to a string object with the item object's name.

CItem::GetData

virtual CUniversal* GetData (void);

Remarks	Dummy function. Gets a pointer to the universal data object which is always a long integer.
Return Value	Pointer to the universal data object with value (long integer).
See Also	***CUniversal, C_DAT:: GetData***

CItem::GetType

e_ItemTypes GetType (void);

Remarks Returns the type of the appropriate item in a form of an enumeration.

CItem::GetTypeName

CString GetTypeName (void);

Remarks Returns the type of the appropriate item in a form of a string.

CItem::GetItemType

e_ItemTypes GetItemType (CString &*stType*);

stType String object representation.

Remarks Is a static method of CItem. It returns the enumeration of the item types from the string representation.

Return Value Enumeration representation, or default value (IT_Item) if not available (see the enumeration in section on page).

CItem::GetItemTypeName

CString GetItemTypeName (e_ItemTypes *eType*);

eType Enumeration representation of item type.

Remarks Is a static method of CItem. It returns the string representation from the enumeration of the item types. When called the first time it initializes the appropriate text arrays.

Return Value String representation of the item type.

CItem::GetAttrNameGetAttrName

static CString GetAttrName (e_ItemAttributes *IAtt*);

IAtt Enumeration of the item objects attributes.

Remarks Is a static method of CItem. It returns the name of the appropriate attribute, e.g. "PeV".

Return Value String representation of the attribute.

CItem::GetAttr

e_ItemAttributes GetAttr (CString &*stAttrKey*);

stAttrKey	One of the allowed attributes (e.g. "PeV").

Remarks Is a static method of CItem. It returns the enumeration that matches with attribute key.

Return Value Enumeration of the attributes if allowed, else out range.

CItem::AddDependent

void AddDependent (CItem* *pItem*);

pItem	Pointer to the item that will be dependent.

Remarks Add pItem to the dependent list of the current item. If the current item changes the pItem's method ChangedOn().

See Also ***CIO_Handler::ChangedOn***

CItem::Disconnect

virtual void Disconnect (void);

Remarks Typically used by a IO-Handler object when it is destroyed.

class C_AL : public CItem

C_AL is the class of alarm objects. It contains all the data and methods to handle the alarm aspects of an application. The virtual method ***SetValue()***, typically called via a Vfsm's virtual output or a user written output action, sets one of the commands to the alarm object (see `e_ALA_Cmds` in VSYSTYP.H). The internal value of the item represents the state of the alarm (see `e_ALA_States` in VSYSTYP.H). An alarm object acts as a simple finite state machine. The state table is presented here in a form of a transition matrix (the preambles "AC_" and "AS_" are omitted):

from \ to	NONE	STAYING	COMING	ACKN.	COM_GO	GOING
NONE		Stay / I	Com / I			
STAYING	Ack / R					
COMING				Ack / R	Go	
ACKN.						Go / I
COM_GO			Com / RI			Ack / RI
GOING	Ack / R		Com / RI			

Commands: Com=Coming, Go=Going, Stay=Staying, Ack=Acknowledge

Actions: R=Remove from.., I=Insert to.., RI=Remain in Alarm Queue but move to the head

Alarm Text

The text property of the alarm objects can contain:

- A plain alarm text directly set to the alarm message.
- Several string constants (see the chapter Internationalization).
- References to data objects (Dat, Ni, No, Par, Udc).

When an alarm enters the state COMMING or STAYING the value of the referenced object is read and copied together with its unit (not if data object is an enumeration) into the actual alarm text. This can lead to more clear alarm texts like: "Val. Is too high: 6.83V. Limit is: 1V(AlarmHigh)"

Example

The AL property's entry Text (set via stateWORKS Studio) could contain:

```
IDS_TOOHIGH %NiVoltage IDS_OUTBAND %ParLimitHigh (AlarmHigh)
```

The above alarm text example shows several concepts:

– Text can contain several string constants: IDS_TOOHIGH and IDS_OUTBAND. If an IDS_xx identifier is not in the resource a warning is entered in the sulog.txt file and IDS_xx itself is copied to the alarm text.

– Text can contain several references to data objects: %NiVoltage and %ParLimitHigh.

– Data references start with the % character. Of course the data object with name NiVoltage has to be in the stateWORKS project. Otherwise the name itself is displayed in the alarm text and a warning is entered to the sulog.txt file.

– String constants and references can be mixed with plain text, see "(AlarmHigh)".

– String constants and references must not contain spaces. Spaces are used as separators.

Remark

Decoding of string constants: see the chapter *Internationalization Support*.

Alarm Queue

Active alarms are linked in a queue, the latest at the head. The head is represented by a pseudo alarm item "AL". If a client advises one of the attributes of "AL", it gets the data of the alarm at the head of the queue. If an alarm disappears (becomes inactive) it is removed from the queue. If this alarm was at the head, the data of the next one in the queue is sent under the name "AL" to the clients. So a client that requests to be advised of the item "AL" always gets the data of the latest active alarm. The alarm queue is a single linked list with the element "AL" as a root.

Alarm Event

According the alarms object's category property the alarm events can be sent to the AlarmLog.txt file in the following form (taken from an example):

```
G:\StateWORKS\Projects\Examples\RegulatorSH\Conf\RegulatorSH.swd
started at: 06-Aug-04 18:25:57
ERROR   06-Aug-04 18:26:39 - Reg2:Al:Pressure - STAYING - Reg2:
Pressure regulating error ( 247.061mBar)
WARNING 06-Aug-04 18:26:35 - Reg1:Al:Motor - STAYING - Reg1:
Motor too hot
INFO    06-Aug-04 18:51:54 – Reg2:Al:Pressure – NONE
Terminated at: 06-Aug-04 18:27:59
```

The AlarmLog.txt file is created in the Config directory and the alarms are appended to it at any time.

The alarm log knows three alarm severity levels: Error, Warning and Information.

The alarm system of the state*WORKS* realtime database looks for an EP type parameter `AL_CatKeyPar`. It can have a value from 0..7 with a binary meaning.

1	Error to AlarmLog
2	Warning to AlarmLog
4	Information to AlarmLog

For instance if `AL_CatKeyPar` is 7 all three types of alarms are written into the alarm log file. This applies to all alarms in the stateWORKS realtime database. If the realtime database does not find the `AL_CatKeyPar` all types of alarms are written into the alarm log file.

Alarm Category

Every individual alarm instance has the Category-Property that determines whether its events are sent to the alarm log.

Category	**COMMING, STAYING**	**GOING, NONE**
1	Error to AlarmLog	Information to AlarmLog
2	Warning to AlarmLog	Information to AlarmLog
4	Information to AlarmLog	Information to AlarmLog
Other	Not sent to AlarmLog	

#include "vsala.h"

Runtime Virtual Public Members

SetValue	Sets the C_AL object's command value.
GetValue	Returns the state of a C_AL object.

Member Functions

C_AL::SetValue

virtual void SetValue (int *nVal*);

nVal C_AL command casted to e_ALA_Cmds (see VSYSTYP.H).

Remarks Can cause a change of C_AL's state (internal value) and so the advise of other items or clients.

C_AL::GetValue

virtual int GetValue (void);

Remarks Returns the item object's internal value. Here this is the state of the C_AL object cast to e_ALA_States (see VSYSTYP.H).

C_ALA::GetAlarmText

CString* GetAlarmText (void);

Remarks Returns the current alarm text. String constants or references are connected if set via configuration file.

C_ALA::SetCategory

void SetCategory (CString *stCat*);

Remarks In a running system this has no influence whether the events are sent to the AlarmLog or not.

C_ALA::GetCategory

CString* GetCategory (void);

Remarks Returns the current category

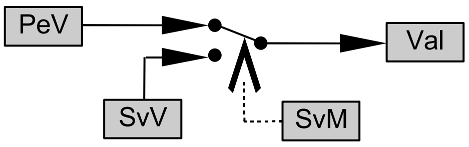
Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	Item's state as number, see e_ALA_States
State Name	R	Item's state as text string
Category	R	Text defined in state*WORKS* Studio/AL-Properties
Text	R	Alarmtext defined in state*WORKS* Studio/AL-Properties
Time	R	Time and date when alarm's state was entered
List	R	State, Time, category and text in one string, separated with <LF>
Acknowledge	W	Empty message, command to AL state machine

Remarks Category and Text come via the Configuration File from the stateWORKS Studio/AL-Properties to the alarm item object. The VFSM System does nothing with them.
They are just provided for display purposes.

class C_CMD : public CItem

Commands are objects holding typically the connection between a (master) Vfsm or a User Interface (Client) and another Vfsm (slave). Commands contain a feature called “service mode” that allows (via a client) for overriding the value set by the (master) Vfsm.



The Peripheral Value *PeV* is set typically by a (master) Vfsm via the method ***SetValue()*** or via a Client/Server access to the attribute "PeV". If the Service Mode is OFF (*SvM* = false), the Peripheral Value is passed to the C_CMD item object's internal value *Val* and from here typically to a (slave) Vfsm. If the Service Mode is ON, the Service Value *SvV* is passed to the internal value *Val*.

Command Names

A C_CMD item object can be connected to a Vfsm’s IO-Description File from where it takes the C-Block as its command names:

```
C # Name    - Cmd Names -         Value
  1 Off                             7
  2 CmdX                           10
  3 CmdY                           11
  4 CmdZ                           12
```

A C_CMD delivers then as the attribute "PeV" and "SvV" the corresponding command name (without command names they return the numbers). The attribute "Lst" delivers a list of the command names with the appropriate numbers, as shown in the following example (\n is the <NL> character 0x0A):

```
"7 Off\n10 CmdX\n11 CmdY\n12 nCmdZ\n"
```

#include "vscmd.h"

Runtime Virtual Public Members

SetValue	Sets the C_CMD object's Peripheral Value.
GetValue	Returns the internal Val value of a C_CMD item object.

Member Functions

C_CMD::SetValue

virtual void SetValue (int *nVal*);

nVal Numerical value of the command.

Remarks The Peripheral Value is set directly; the internal value depends on the Service Mode switch.

C_CMD::GetValue

virtual int GetValue (void);

Remarks Gets the internal value of the item. This means here the command value of the C_CMD as it is sent to the (slave) Vfsm.

Return Value The command as number.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	Internal value as number
Service Mode	X	0 -> serv.mode OFF, 1 -> ON
Service Value	X	As numbers or as text, see according IOD-file.
Peripheral Value	X	As numbers or as text, see according IOD-file.
Type Name	R	Name of the according IOD-file, or empty if none
List	R	Numbers and names as a list

Remarks The type name (the IOD-file name) comes via the Configuration File from the state*WORKS* Studio/CMD-Properties to the command item object.

class C_CNT : public CItem

C_CNT is the class of counter objects. It contains all the data and methods to handle the counter aspect of an application. The virtual method ***SetValue()*** sets one of the commands to the counter object (see e_CNT_Cmds in VSYSTYP.H). The C_CNT item's internal value represents the state of the counter (see e_CNT_States in VSYSTYP.H). A counter object acts as a simple finite state machine. The state table is presented here in a form of a transition matrix (the preambles "CC_" and "CS_" are omitted):

from \ to	RESET	STOP	RUN	OVER	OVERSTOP
RESET			Start, ResetStart		
STOP			Start, ResetStart		
RUN	Reset	Stop	ResetStart	“expiration”	
OVER	Reset		ResetStart		Stop
OVERSTOP	Reset		ResetStart	Start	

“Expiration” happens when the Counter Value becomes equal to the Counter Constant.

In any state the command CC_NewCountConst is allowed. It can lead in the state CS_RUN to an expiration of the counter. The commands CC_IncCounter and CC_DecCounter are possible in the states CS_RUN and CS_OVER. CC_IncCounter in CS_RUN can lead to an expiration of the counter. The state remains CS_OVER even if CC_DecCounter is issued. The commands CC_Reset and CC_ResetStart set the counter register to zero. Instances of C_CNT are used typically to count system internal events like for instance certain state changes of a Vfsm. C_CNT is also the superclass of the timer- and event-counter-items.

#include "vscnt.h"

Runtime Virtual Public Members

SetValue	Sets the C_CNT object's command.
GetValue	Returns the state of a C_CNT object.
SetCountConst	Sets the C_CNT's counter constant.
GetCountConst	Gets the C_CNT's counter constant.
GetCountRegister	Gets the C_CNT's counter register.

Member Functions

C_CNT::SetValue

virtual void SetValue (int *nVal*);

nVal C_CNT command casted to e_CNT_Cmds.

Remarks Sets the C_CNT object's command. This can cause a change of C_CNT's state and so the advise of other items or client/servers.

C_CNT::GetValue

virtual int GetValue (void);

Remarks Returns the item object's internal value. Here, this is the state of the C_CNT object casted to e_CNT_States (see VSYSTYP.H).

C_CNT::SetCountConst

virtual void SetCountConst (int *nCountConst*);

nCountConst Count Constant (or Limit).

Remarks Sets a C_CNT item object's Counter Constant value.

C_CNT::GetCountConst

virtual int GetCountConst(void);

Remarks Returns the C_CNT item object's Counter Constant value.

Return Value Counter Constant value.

C_CNT::GetCountRegister

virtual int GetCountRegister(void);

Remarks Returns the C_CNT item object's Counter Register value.

Return Value Counter Register value.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	Counter's state as number see e_CNT_States
Value	W	Counter's commands as number see e_CNT_Cmds
State Name	R	Counter's state as a text string
Count Constant	X	as an (integer) number
Count Register	R	as an (integer) number

class C_DAT : public CItem

C_DAT items are objects holding values typically of physical types like parameters or numerical values. A C_DAT item has the attributes: Data Value, Data Format and Physical Unit. Instances of C_DAT items can be used as objects too but typically C_DAT just serves as superclass of the PAR, NI, NO or UDC item objects.

A C_DAT item object acts as a simple finite state machine. The state table is presented here in a form of a transition matrix (the preambles "SC_" and "SS_" are omitted):

from \ to	OFF	INIT	UNDEF	CHANGED	DEF
OFF	-	on	-	-	-
INIT	off	-	-	“new data”	-
UNDEF	-	-	-	“new data”	-
CHANGED	off	-	-	”new data”	set
DEF	off	-	-	-	-

“New data” happens when the data changes.

Remarks The UNDEF state is used only for PAR items whose Category attribute equals PG_PP or PG_PP_Coded.

Data Value

The CItem's internal value represents the state of the data (see e_DATA_States in VSYSTYP.H). The state of a CItem (here a C_DAT) object generally represents the control flow (see the state machine transition table above) in the control system, the data value - the data flow.

Data Format

The format determines the numerical representation of the C_DAT's data value. Numerical values are always stored in 32Bit but nevertheless values are rounded or limited to the resolution of the appropriate format. The following list shows valid formats and their specifications (the syntax and meaning is almost C-like):

Format	Description
bool	two values: true and false
char	-128..+127 (8Bit signed)
unsigned char	0..255 (8Bit unsigned)
unsigned short int	0..65535 (16Bit unsigned)
long	-2G..+2G (32Bit signed)
float	32Bit floating point, representation most convenient (E or F)
%e0 .. %e8	32Bit floating point, representation exponent, e.g. "3.5e-12"
%f0 .. %f8	32Bit floating point, representation fixpoint, e.g. "1.2345678"
string	a (almost) unlimited text string

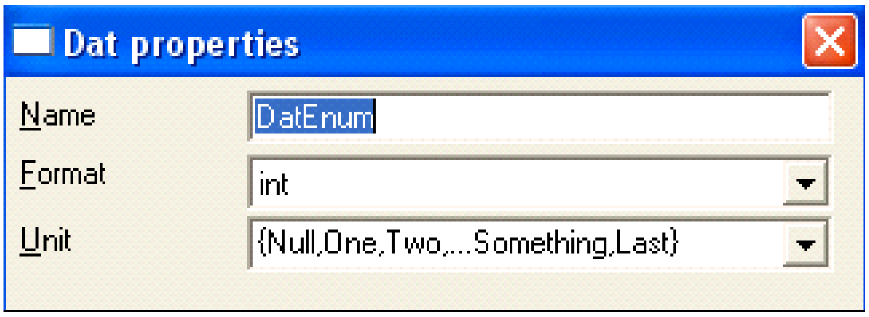
Remarks %En and %Fn are floating point values. The number n in 0..8 range specifies the number of digits when sent to a client. It is not the internal representation; this is always the full range.

Unit

Unit is a free selectable string representing the physical like "mA" or "sec". The string is not used internally. It just serves the client to display the data appropriately.

Enumeration

Data objects have a unit attribute intended to add the physical unit of the represented data. This unit can also be used to define the data as a set of enumeration. In the state*WORKS* Studio enter the following string to the Unit-Property of a data object (Dat, Ni, No, Par, Udc):



Name: DatEnum
Format: int
Unit: {Null,One,Two,...Something,Last}

The brackets are important; they declare the text inside as an enumeration definition instead as a real unit text.

Data Value	..-1	0	1	2	3	4	5	6	7	8..
Display	..-1	Null	One	only Two	3	4	5	Sex	Crime	8..

Values entered as strings are analyzed. If an enumeration text is recognized it is converted to the according numerical value and set to the internal data value. Valid numbers are also accepted.

Remarks

Units can be replaced by string constants like, e.g. IDS_ENUM.
Comma is used as separator. No text between commas means no text replacement for the number. Start with a comma {,One,Two} if zero is not used.
The value ranges are positive integers including zero.
The Format-Property may be anything (but a string). Float values are rounded down to integers.
Data Values without an assigned enumeration value are displayed in the original format:

Data Value	-1.5	0.1	1.1	2.99	3.01	1.23e45
Display	-1.5	Null	One	Two	3.01	1.23e45

Use of Enumeration Class

Enumerations are typically used in context with Cmd objects. Enumerations can also be used elsewhere, for instance in an IO-Handler.

Example

```
CString       stName("MyEnum");
CString       stEnum("{Null,One,only Two,,,,Sex,Crime}");
CEnumeration* pEnum;

pEnum   = m_pEnumList->Lookup(stName, stEnum);
```

Remark

Explanation of enumeration: see the above chapter.

#include "vsdata.h"

Runtime Public Members

GetValue	Returns the state of a C_DAT object.
GetFormat	Gets the format specification in enumeration representation.
GetUnit	Gets the string containing the C_DAT object unit.
SetData	Sets the data value of a C_DAT object.
GetData	Returns the universal data object.
bGetData	Returns the value of a C_DAT object as boolean.
chGetData	Returns the value of a C_DAT object as character.
uchGetData	Returns the value of a C_DAT object as unsigned char.
nGetData	Returns the value of a C_DAT object as short integer.
unGetData	Returns the value of a C_DAT object as unsigned short integer.
lGetData	Returns the value of a C_DAT object as long.
fGetData	Returns the value of a C_DAT object as float.
pstGetData	Returns the pointer of a C_DAT if it is a string.
stGetData	Returns the string of a C_DAT if it is a string.

Member Functions

C_DAT::GetValue

int GetValue (void);

Remarks Gets the internal value, the state of the C_DAT item. This values can be one of the enum e_DATA_States (DS_OFF, DS_UNDEF, DS_DEF, DS_CHANGED, DS_INIT, DS_SET).

Return Value C_DAT state as a number (see e_DATA_States is SYSTYP.H).

C_DAT::GetFormat

e_DATA_Formats GetFormat (void);

Remarks Gets the format of the data in enumeration representation (see VSYSTYP.H).

Return Value Data format as enumeration e_DATA_Formats.

C_DAT::GetUnit

CString* GetUnit (void);

Remarks Gets the physical unit of the C_DAT object as a string.

Return Value Pointer to a string object containing the C_DAT's Physical Unit.

C_DAT::SetData

void SetData (bool *bData*);

void SetData (char *nData*);

void SetData (unsigned char *nData*);

void SetData (short *nData*);

void SetData (unsigned short *nData*);

void SetData (long *lData*);

void SetData (float *fData*);

void SetData (CStdString& *stData*);

void SetData (CUniversal& *Data*);

	n,l,f,st,Data Input value in appropriate data format.
Remarks	The value is transformed and possibly limited to the C_DAT objects own format. Changes the C_DAT's status to DS_CHANGED and of course the data value to the specified value.

C_DAT::GetData

CUniversal* GetData (void);

Remarks	Gets a pointer to the universal data object. Sets the state of the C_DAT object to DS_DEF if it was DS_CHANGED before. This signals that the data was read (consumed) at least once after setting to a new value.
Return Value	A pointer to a universal data object CUniversal.
See Also	***CUniversal***

C_DAT::GetData

virtual bool GetData (CString* *pstVal*, r_Universal *data*);

pstVal	Pointer to the result string, at least "".
Data	Input value in appropriate data format.

Remarks	Represents in pstVal data value, Format is internally specified.
Return Value	true -> if ok, else false.

C_DAT::xGetData

bool bGetData (void);

char chGetData (void);

unsigned char uchGetData (void);

short int nGetData (void);

unsigned short unGetData (void);

long lGetData (void);

float fGetData (void);

CString* pstGetData (void);

CString stGetData (void);

Remarks Returns the C_DAT object's value in the appropriate format. The value is transformed and possibly limited. Sets the state of the C_DAT object to DS_DEF, if it was DS_CHANGED before. This signals that the data was read (consumed) at least once after setting to a new value.

Return Value A value in appropriate format.

C_DAT::Display

bool Display (CString* *pstVal*);

pstVal Output parameter; string representation of the data value.

Return Value true if value is available.

Remarks String representation is copied to pstVal. Possibly enumerations are displayed.

C_DAT::DisplayWithUnit

bool DisplayWithUnit (CString* *pstVal*);

pstVal String representation of the data value with unit attached, e.g. 6.83V

Return Value true if value is available.

Remarks String representation is copied to pstVal. Possibly enumerations are displayed, and in that case no unit is displayed.

C_DAT::Set

virtual bool Set(CString* *pstVal*);

pstVal String representation of the data value to be set.

Return Value true if data value has changed, else false.

Remarks Puts the string representation of a value to its appropriate format. Applies the limits format according to the value. Tests whether the value has changed. In case of a format error lets the data value unchanged, but sets changed true nevertheless. If it is an enumeration, tests for that.

C_DAT::SetUnit

void SetUnit (CString *stUnit*);

stUnit new unit.

Remarks Overwrites the unit of the data item. No enumeration possible.

C_DAT::SetUnitText

CString GetUnitText (CString *stUnit*);

stUnit Unit text or string constant.

Remarks If stUnit is a string constant IDS_xxx takes the corresponding contents from the resource file, or else takes the original.

Return Value The original stUnit or the content of the string constant.

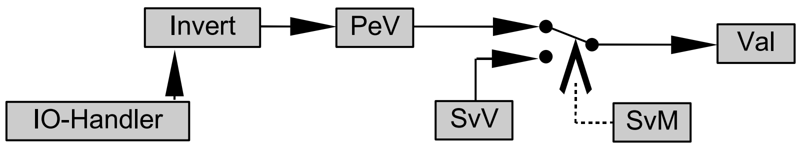
Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	Data object's state as number see e_DATA_States
State Name	R	Data object's state as text string
Format	R	Data object's format as string
Physical Unit	R	Data object's unit as string
Data Value	R	Data object's value as string according its own format
Data Value	W	Set value in any format as a string

Remarks Format and PhysicalUnit come via the Configuration File from the state*WORKS* Studio/DAT-Properties to the data item object. PhysicalUnit is not used by the VFSM System. Format is used to format the value string sent to the destination application.
Data set (poked) by a destination application can be in any valid format. If the format isn't valid the data value is not changed and the old value is sent as update.

class C_DI : public CItem

A C_DI item is an object holding the connection to a real digital input signal. It is set to 0 or 1 (false, true) with its own method ***SetValue()*** by the IO-Handler. C_DIs contain a feature called service mode that allows (via a client) the value set by the IO-Handler to be overridden:



If the Service Mode switch *SvM* is false, the Peripheral Value *PeV* is connected to the C_DI item's value *Val*. This value is also called the Control Value, because it is the value seen by the state machine (Vfsm). If the Service Mode switch is true, the service mode is ON and the Service Value *SvV* is connected to the C_DI item's value *Val* i.e. the DI is disconnected from its peripherals. The Peripheral Value is set by the method ***SetValue()*** that is called typically by an IO-Handler object. The method ***GetValue()*** returns the item's value *Val*. The Service Mode switch and the Service Value are set by the clients (see the C_DI item's attributes "SvV" and "SvM").

Invert

DI objects have the Invert property (set via the state*WORKS* Studio). This is typically used in case of reversed logic. For instance a DI `PressureTooHigh` may be wired as a low voltage level for active to detect a wire break. If the invert property is set then the signal appears in the control system with the correct logical state.

#include <vsdi.h>

Runtime Virtual Public Members

SetValue	Sets the C_DI object's peripheral value.
GetValue	Returns the internal value of the C_DI item.
GetInvert	Returns the value of the Invert property.

Member Functions

C_DI::SetValue

void SetValue (int *nValue*);

nValue State of the Digital Input. 0-> false=Low, not 0->true=High.

Remarks It is typically called by an IO-Handler object to set the state of the appropriate digital input. The Peripheral Value is set directly. The internal value depends on the Service Mode switch.

C_DI::GetValue

int GetValue (void);

Remarks Gets the C_DI item's internal value. Here, this means the Control Value of the C_DI.

Return Value C_DI Control Value. State of the digital input: false=Low=0, true=High=1.If the Service Mode switch is set the value of the Service Value is returned.

C_DI::GetInvert

bool GetInvert (void);

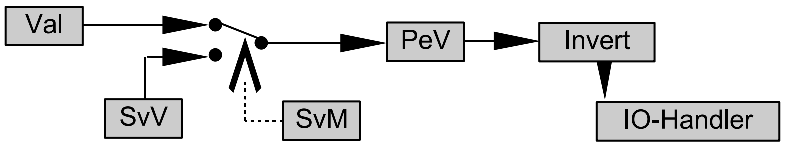
Return Value The member variable.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	DI object's state as number 0 or 1
Service Mode	X	0->service mode in OFF, 1->ON
Sevice Value	X	0->false/Low, 1->true/High
Peripheral Value	R	value set by the IO-Handler (0 or 1)

class C_DO : public CItem

A C_DO item is an object holding the connection to a real digital output signal. It is set to 0 or 1 (false, true) with its own method ***SetValue()*** by a Vfsm's output action or a user written output function. C_DOs contain a feature called service mode that allows (via a client) the value set by the control system to be overridden.



If the Service Mode switch *SvM* is false, the item's internal value *Val* is connected to the C_DO's Peripheral Value *PeV*. If the Service Mode switch is true, the service mode is ON and the Service Value *SvV* is connected to the Peripheral Value *PeV* i.e. the digital output is disconnected from its control (state machine). The internal value *Val* is set by the method ***SetValue()***. The method ***GetValue()*** returns the item's value *Val*. The Service Mode switch and the Service Value are set by the clients (see the C_DO item's attributes "SvV" and "SvM").

Invert

DIOobjects have the Invert property (set via state*WORKS* Studio). This is typically used in case of reversed logic.

If the Peripheral Value changes, the IO-Handler's method ***SetOutput()*** is called to set the value to the according hardware. The IO-Handler is connected to the C_DO item with its method ***Connect()***.

#include "vsdo.h"

Initialization - Virtual Public Members

Connect	Connects the C_DO item to an IO-Handler object.
Disconnect	Disconnects the C_DO item from an IO-Handler object.

Runtime Virtual Public Members

SetValue	Sets the C_DO objects internal value.
GetValue	Returns the internal value of the C_DO item.
GetInvert	Returns the value of the Invert property.

Member Functions

C_DO::Connect

void Connect (CIO_Handler* *pIOHandler*, int *nChannel*);

pIOHandler	Pointer to the IO-Handler object.
nChannel	Number within the array of DOs.

Remarks Typically used by an IO-Handler object to give the C_DO item access to itself. The C_DO item calls during runtime the IO-Handler objects method SetOutput() to set its value to the peripherals.

C_DO::Disconnect

virtual void Disconnect (void);

Remarks Typically used by the IO-Handler object when it is destroyed. Sets the pointer m_pIOHandler to NULL, so that a possible access to the IO-Handler can be avoided (m_pIOHandler -> (member) set to NULL)

C_DO::SetValue

void SetValue (int *nValue*);

nValue	State of the Digital Output false=Low=0, true=High=1.

Remarks Sets the internal value of the C_DO item (Val). If the Service Mode is off nValue is copied directly to the IO-Handler via the appropriate IO-Handler's method SetOutput() (if set with Connect()).

See Also ***CIO_Handler:: SetOutput***

C_DO::GetValue

int GetValue (void);

Remarks Gets the C_DO item's internal value.

Return Value C_DO Control Value. State of the digital output: false=Low=0, true=High=1.

C_DO::GetInvert

bool GetInvert (void);

Return Value The member variable.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	DO object's state as number 0 or 1
Service Mode	X	0->service mode in OFF, 1->ON
Sevice Value	X	0->false/Low, 1->true/High
Peripheral Value	R	value set to the IO-Handler (0 or 1)

class C_ECNT : public C_CNT

C_ECNT is the class of event counter item objects. It contains all the data and methods to handle this aspect of an application. C_ECNT is a subclass of the item type C_CNT (Counter). It inherits the counter's state machine and adds the aspect of the event count. It typically counts the events on an item object, for instance the C_DI. Every time the C_DI gets the configured trigger value (1 for true, 0 for false) it increments its counter register. If an event is not that concrete as a digital signal, it is possible to put a SWIP item between, for example, an analog input (NI) and the event counter. The trigger value is then set, for example, to HIGH (such as the value >4). So every time the NI passes the high limit, the counter is incremented.

#include "vsecnt.h"

Remarks This class has the same interface as C_CNT.

class CItemList

CItemList is the heart of the database. It contains all the items that are created according to the control system designer's configuration. It is a dynamic list with an access key, it grows during startup but it remains constant after that. The access key is the item's name. The name length is unlimited. Requests from the server always go to this list. The connections between the items are established once via this list during startup (afterwards communication goes directly via pointers). CItemList reads the Configuration File during the startup create phase.

#include "vsil.h"

Initialization - Public Members

Lookup	Gets the pointer to an item identified by an access key.
GetFirst	Looks for the first item of the specified item type.
GetNext	Looks for the next item of the specified item type.
GetItemNumb	Gets the number of specified items in the database.

Member Functions

CItemList::Lookup

CItem* Lookup (CString *stKey*);

stKey	Access key to the item object.

Remarks Gets a pointer to the item object addressed by key. If there is no item under the specified name, a NULL pointer is returned.

Return Value A pointer of the superclass type CItem.

CItemList::GetFirst

CItem* GetFirst (e_ItemTypes *ItemType*, POSITION& *pos*);

ItemType	Select special item type (see VSYSTYP.H).
pos	Iteration variable.

Remarks Looks for the first item of the specified item type. Used together with GetNext() to iterate through all of specified item types in the database.

Return Value Pointer to the item object, NULL if not available.

See Also CitemList::GetNext

Example

```
CItemList*    pIL = &(pDoc->m_VfsmSystem.m_ItemList);
CItem*        pItem;
C_UNIT*       pUnit;
POSITION      pos;

 pItem = pIL->GetFirst (IT_UNIT, pos);
 while ( pItem != NULL )
 {
  pUnit = (C_UNIT*)pItem;
  // do something with the pUnit object
  pItem = pIL->GetNext (IT_UNIT, pos);
 }
```

CItemList::GetNext

CItem* GetNext (e_ItemTypes *ItemType*, POSITION& *pos*);

ItemType	Select special item type (see VSYSTYP.H).
pos	Iteration variable.

Remarks Looks for the next item of the specified item type ItemType. Used together with GetFirst() to iterate through all of a specified item types in the database.

Return Value Pointer to the item object, NULL if there are no more items.

See Also CItemList::GetFirst

Example see GetFirst

CItemList::GetItemNumb

int GetItemNumb (e_ItemTypes *ItemType*);

ItemType	Select special item type (see VSYSTYP.H).

Remarks Returns the number of the specified item in the database.

Return Value Number of items in the database.

class CIO_Handler

The class CIO_Handler is a virtual class that is just used as superclass for the user written IO-Handlers. It holds the connection to the database C_UNIT item object that holds the connection to the physical database items like C_DI, C_DO, C_NI or C_NO. User written IO-Handlers inherit the connection to the C_UNIT items and to the methods used by output type database items like C_DO and C_NO.

#include "vsiou.h"

Construction/Destruction - Public Members

CIO_Handler	Creates a CIO_Handler object.
~CIO_Handler	Destroys a CIO_Handler object.

Initialization - Virtual Public Members

Create	Installs the connection to its C_UNIT item.
Connect	Connects the associated output items to its output function.

Initialization - Public Members

GetUnitName	Gets the name of the cooperating C_UNIT item.
GetUnitTypeName	Gets the type of the cooperating C_UNIT item.
GetPhysicalAddress	Gets the physical address of the cooperating C_UNIT item.
GetCommPort	Gets the communication port of the cooperating C_UNIT item.

Runtime Virtual Public Members

SetOutput	Output function used by the associated output items.
GetNumbAssItem	Gets the number of associated items.
GetAssItem	Gets the indexed associated item.
pUnit	Gets a pointer to the cooperating UNIT item.
pItemList	Gets a pointer to the (global) item list.
ChangedOn	Item of dependency has changed

Member Functions

CIO_Handler::CIO_Handler

CIO_Handler (void);

Remarks Constructor.

CIO_Handler::~CIO_Handler;

~CIO_Handler (void);

Remarks Destructor.

Remarks The constructor and destructor of this class are listed here as users have to use them while writing IO-Handlers.

CIO_Handler::Create

virtual bool CIO_Handler::Create (C_UNIT* *pUnit*);

	pUnit Pointer to the associated C_UNIT item object.
Remarks	Stores the pointer pUnit for later use and gets and stores the pointer to the associated item list of this C_UNIT item.
Return Value	true if succeeded.

CIO_Handler::Connect

virtual void Connect(void);

Remarks Typically used by the subclasses to connect the associated output items like C_DOs or C_NOs to themselves, so that they can set their output values to appropriate output hardware.

CIO_Handler::GetUnitName

CString GetUnitName (void);

Remarks Returns the cooperating C_UNIT item's name if it exists, otherwise "".
Return Value String with name of C_UNIT item.

CIO_Handler::GetUnitTypeName

CString GetUnitTypeName (void);

Remarks Returns a string object with the cooperating C_UNIT item's type if it exists, otherwise "".
Return Value String with name of the C_UNIT item's type.

CIO_Handler::GetPhysicalAddress

int GetPhysicalAddress (void);

Remarks Returns the physical address of this IO-Handler. It comes from the Configuration File via the cooperating C_UNIT item.
Return Value Number with the physical address.

CIO_Handler::GetCommPort

int GetCommPort (void);

Remarks Returns the communication port number of this IO-Handler. It comes from the Configuration File via the cooperating C_UNIT item.
Return Value String with the communication port name.

CIO_Handler::SetOutput

virtual void SetOutput (bool *bVal*, int *nChan*);

virtual void SetOutput (short int *nVal*, int *nChan*);

virtual void SetOutput (long *lVal*, int *nChan*);

virtual void SetOutput (float *fVal*, int *nChan*);

virtual void SetOutput (int *nChan*);

b/n/l/fVal Value of the output item to set to the output hardware in appropriate format.
nChan Sets the output value to this channel number.

Remarks Typically used by output items like C_DO and C_NO to set their values via this IO-Handler to a physical output.
The last ***SetOutput()*** method triggers only a channel – the data must be supplied in the IO-Handler (got from RTDB, calculated, etc., must be used with of ***UseSimpleGetOutput()***, see the description of the C_NO class).

See Also C_DO::SetValue, C_NO::SetData

CIO_Handler::GetNumbAssItem

int GetNumbAssItem (void);

Remarks Returns the number of associated items of the cooperating C_UNIT item. It is the maximum possible number. It does not mean that these items are really existent but there are slots for them.

Return Value Possible number of associated items of the cooperating UNIT item.

CIO_Handler::GetAssItem

CItem* GetAssItem (int *nObjID*);

nObjID Index to the associated item.

Remarks Returns the pointer to the indexed associated item. This pointer can be NULL if there is no item or if the index is outside the number of items of that UNIT.

Return Value Pointer to a CItem object.

CIO_Handler::pUnit

C_UNIT* pUnit (void);

Remarks Returns the pointer to the cooperating C_UNIT item.

Return Value Pointer to the cooperating UNIT item.

CIO_Handler::pItemList

CItemList* pItemList (void);

Return Value Returns the pointer to the (global) item list.

CIO_Handler::ChangedOn

void ChangedOn(CItem* *pItem,* e_ItemAttributes *attr*);

pItem Pointer to item that has changed
attr This attribute of pItem has changed (see VSYSTYP.H)

Remarks Is called by the item from which the current item is dependent, to say that it has changed.

See Also ***CItem::AddDependent***

Example

```
void CIO_HandlerExa::Connect(void)
{
CItem*   pItem;
pItem = GetAssItem(EXA_B_Cmd);
 if(pItem != NULL)
 {
  if(pItem->GetType() == IT_CMD)
  {
   m_pCmd = (C_CMD*)pItem;
   m_pCmd->AddDependent(m_pUnit);
  }
 }
} /* End of CIO_HandlerExa::Connect */

void CIO_HandlerExa::ChangedOn(CItem* pItem, e_ItemAttributes
attr)
{
 if (pItem == m_pCmd)
  if (attr == IAtt_Value)
  {
   switch (m_pCmd->GetValue())
   {
    case 1: // Load
     Load(m_pPar->stGetData());
    return;
    case 2: // Save
     Save(m_pPar->stGetData());
    return;
   }
  }
} /* End of CIO_HandlerExa::ChangedOn */
```

class C_NI : public C_DAT

C_NIs are objects derived from the class C_DAT. They are typically used as representation of hardware devices such as ADCs, Coders or (Hardware) Counters. As the C_Dis, C_NIs are organized in appropriate IO-Handlers. The state of a C_NI item object corresponds to one of the C_DAT states such as DS_CHANGED or DS_DEF to signal for instance to a SWIP item object to check its data value. The method ***SetInput()*** is characteristic for C_NIs. It takes a value (e.g. the output of a 12Bit ADC 0..4091) to transform it, e.g. to a current value (e.g. -200… +200A). It can scale the input values with several methods that all take the parameters Scale Factor, Offset and Scale Mode (linear, exp). C_NI adds to the inherited attributes of C_DAT (Data Value, Physical Unit and Format) the attributes: Scale Factor, Offset, Scale Mode and Limit Low.

Scale Algorithms

C_NI items are capable to transform the raw information from the peripherals to scaled values of appropriate physical units. The following scaling methods are available (where *x* -> Input Value delivered by ***SetInput()***, *y* -> Data Value, *a* -> Scale Factor, *b* -> Offset):

none	y = x	(the Data Value is passed to the Output Value as is)
Lin	y = a*x + b	
Exp	y = e^(a*x + b)	

#include "vsni.h"

Runtime Virtual Public Members

SetInput	Sets the C_NI item object's data value via scaling.
GetThreshold	Gets threshold value

Member Functions

C_NI::SetInput

void SetInput (int *nInp*);

void SetInput (unsigned integer *nInp*);

void SetInput (long *lInp*);

void SetInput (float *fInp*);

void SetInput (CString& *stInp*);

n,w,l,fInp, stInp Input value in appropriate data format.

Remarks Is typically called by an IO-Handler object to pass a value from the peripherals to the C_NI item. The value is transformed and possibly limited to the appropriate format and scaled using the scaling method configured. SetInput() changes the C_NI's status to DS_CHANGED and of course the data value to the specified value.

Warning Do not mix-up with the C_DAT item's virtual method SetData() that goes directly to the data value without scaling.

C_NI::GetThreshold

Cuniversal* GetThreshold(void);

Remarks Get threshold as a universal data.

Rteurn Value Pointer to the threshold data in the appropriate format.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	NI object's state as number see e_DATA_States
State Name	R	NI object's state as text string
Format	R	NI object's format as string
Physical Unit	R	NI object's unit as string
Data Value	R	NI object's value as string according its own format
Scale Factor	R	Scale factor
Offset	R	Offset
Scale Mode	R	Scale mode as string
Limit Low	R	Threshold value, means: don't update if difference is lower

class C_NO : public C_DAT

C_NOs are objects derived from the class C_DAT. They are typically used as representations of hardware devices like DACs or numeric output registers. As for the C_DOs, C_NOs are organized in appropriate IO-Handlers. The state of a C_NO is the one of C_DAT's states like DS_CHANGED, DS_DEF, DEF_SET or DS_OFF. A C_NO object accepts four commands via the method ***SetValue()***: DC_Off, DC_On, DC_Set and DC_NewData. These commands are set by a Vfsm's virtual output or a user written output function. There are two ways to set a C_NO item's output value:

- A C_NO item object can be plugged to a parameter (C_PAR item) or a table (C_TAB item) object containing the data value. With the commands DC_Off and DC_On the output value of the C_NO item is switched to zero respectively to the PAR or TAB's data value. With this method a C_NO item can be switched on and off (and via TAB to several data values) just by a Vfsm's virtual output.
- The virtual C_DAT method ***SetData()*** sets the C_NO item's Data Value directly. This method is typically used by user written output functions, if the previous method is too limited.

In both cases the data value, e.g. a voltage of -20V..+20V, is transformed to a code value which could be something like 0..4091 as output to a 12Bit DAC. This output value is passed to the peripherals via the method ***SetOutput()*** of the appropriate IO-Handler object. The IO-Handler is connected to the C_NO item with its method ***Connect()***. It can scale the data value with several methods that all take the parameters Scale Factor, Offset and Scale Mode (none, linear, exp). C_NO is like C_NI a subclass of C_DAT. It adds to its attributes (Data Value, Physical Unit and Format) the attributes Scale Factor, Offset and Scale Mode. In addition, C_NO has also the Out_Data attribute to define the source of the output value (C_PAR or C_TAB).

Scale Algorithms
C_NO items are able to transform the data value of a physical unit into the information of the peripherals. The following scaling methods are available and can be defined in the Configuration (*x* -> Data Value delivered by ***SetData()***, *y* -> Output Value, *a* -> Scale Factor, *b* -> Offset):

none	y = x	(the Data Value is passed to the Output Value as is)
Lin	y = a*x + b	
Exp	y = e^(a*x + b)	

#include "vsno.h"

Runtime Virtual Public Members

SetValue	Sets a command to the C_NO item.
SetData	Sets the C_NO item the object's data value.
n/l/fGetOutput	Call for the scaled output value in according format.
Disconnect	Sets the pointer to m_pIOHandler to NULL.

Runtime Public Members

UseSimpleGetOutput	Forces the NO item to use the SetOutput(nChan) method instead of SetOutput(xVal, nChan).

Member Functions

C_NO::SetValue

virtual void SetValue (int *nVal*);

nVal	Number of a valid C_NO command (see e_DATA_Cmds in VSYSTYP.H).

Remarks Sets the command to the C_NO. It accepts DC_Off, DC_On, DC_Set and DC_NewData:

`DC_Off`	State goes to DS_OFF and the data and output are set to zero.
`DC_On`	State goes to DS_CHANGED the output is set to the data of the attached parameter or table.
`DC_NewData`	State goes to DS_CHANGED the new data is copied to the internal data and output.
`DC_Set`	State goes to DS_SET. Value is set to the output. It is not copied from the attached parameter or table.

C_NO::SetData

virtual void SetData (short *nData*);

virtual void SetData (unsigned short *nData*);

virtual void SetData (long *lData*);

virtual void SetData (float *fData*);

virtual void SetData (CUniversal& *Data*);

l,n,f,Data	Output value in appropriate format.

Remarks Is typically called by user written output functions. The value is transformed to the appropriate format. SetData() changes the C_NOs status to DS_CHANGED (and of course the data value) and sends the scaled value to the attached output handler via the IO-Handler's method SetOutput().

See Also ***CIO_Handler:: SetOutput***

Example

```
int Func1 (CItem* pOwner, int nVO)
{
C_NO* pAO;
short   i = 2047;

 pAO = (C_NO*)GetAssItem(5);
 pAO->SetData(i);
 return 1;
}
```

C_NO::xGetOutput

short int nGetOutput (void);

long lGetOutput (void);

float fGetOutput (void);

Remarks Gets the scaled output value in according format.

Return Value Output value of the NO is scaled, formatted and possibly limited.

Example

```
void CAOUnit::SetOutput (int nChan)
{
long lVal;

 if(m_pSetNOs)
 {
  lVal = m_paNO[nChan]->lGetOutput();
  // .. put lVal to DAC hardware
 }
} /* End of CAOUnit::SetOutput */
```

C_NO::Disconnect

virtual void Disconnect (void);

Remarks Typically used by the IO-Handler object when it is destroyed. Sets the pointer m_pIOHandler to NULL, so that a possible access to the IO-Handler can be avoided (m_pIOHandler -> (member) set to NULL)

C_NO::UseSimpleGetOutput

void UseSimpleGetOutput (void);

Remarks Typically used by IO handles during system initialization. Forces the NO to use the CIO_Handler::SetOutput (int nChan) virtual method with the intension to use one of the xGetOutput() methods.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	NO object's state as number, see e_DATA_States
Value	W	NO object’s command number, see e_DATA_Cmds
State Name	R	NO object's state as text string
Format	R	NO object's format as string
Physical Unit	R	NO object's unit as string
Data Value	R	NO object's value as string according its own format
Data Value	W	Set NO object's value as string representation
Scale Factor	R	Scale factor
Offset	R	Offset
Scale Mode	R	Scale mode as string

class C_OFUN : public CItem

Output Function items are objects holding a connection to a user written procedure of a well defined type. A C_OFUN item object has the attributes: the pointer to the C_UNIT or C_VFSM item it belongs to and a pointer to the appropriate user written output function. The CItem's internal value represents the return value of the Output Function that is a user defined integer. The virtual method ***SetValue()*** calls the output function and passes to it the set value and the pointer to the item that owns the C_OFUN object.

#include "vsofu.h"

Remarks The C_OFUN class has no public programming interface.

Host Interface

Attribute	Ext.	Acc.	Description (enumeration see VSYSTYP.H)
Value		X	OFUN object's value (the OFun's input or return value)

class C_PAR : public C_DAT

C_PAR items are objects holding values used as time constants, output values or switchpoint values. A C_PAR item is a subclass of C_DAT. It adds to its attributes Data Value, Physical Unit and Format the attributes Category (describes the persistence aspect), LimitLow/High and Init Value. The state of a C_PAR item object corresponds to one of the C_DAT states such as DS_CHANGED or DS_DEF.

Category

There are two main parameter category classes: `PP` (process parameters) and `EP` (equipment parameters). Equipment parameters typically define a machine's general behavior, changed from time to time, but valid over system shutdowns. Process parameters are typically used as a kind of recipe. The management of the PPs is the user interface's responsibility. The EPs (in a Windows NT environment) are automatically stored in the Registry. With three EP categories it is possible to store EPs workstation wide:

PP	Don't store to Registry, managed otherwise e.g. by the user interface	
EP	HKEY_CURRENT_USER	path ../EP
EP_LM_ADMIN	HKEY_LOCAL_MACHINE	path ../EP_ADMIN
EP_LM_USERS	HKEY_LOCAL_ MACHINE	path ../EP_USERS

Remarks

The category EP allows parameter to be stored and used once individually per user, EP_LM_USERS common to all users. The EPs of category EP_LM_ADMIN are write protected from other users for Administrators.
The Registry is used under Windows. In UNIX-like operating systems the Registry is replaced by a file system (see the CRegistry class). Handling of that is transparent for the user: RTDB built for Windows stores the EP parameters in Registry, RTDB built for UNIX stores the parameters in appropriate files.

#include "vspar.h"

Runtime Public Members

GetInitValue	Returns a pointer to a CUniversal object with the initialization value.
GetLimitLow	Returns a pointer to a CUniversal object with the low limit.
GetLimitHigh	Returns a pointer to a CUniversal object with the high limit.
SetLimitLow	Sets the value of the limit low attribute.
SetLimitHigh	Sets the value of the limit high attribute.

Member Functions

C_PAR::GetInitValue

CUniversal* GetInitValue (void);

Remarks

Returns a pointer to a CUniversal object containing the initialization value in the PAR object's own format.

C_PAR::GetLimitLow

CUniversal* GetLimitLow (void);

Remarks

Returns a pointer to a CUniversal object containing the low limit value in the PAR object's own format.

C_PAR::GetLimitHigh

CUniversal* GetLimitHigh (void);

Remarks Returns a pointer to a CUniversal object containing the high limit value in the PAR object's own format.

C_PAR::SetLimitLow

void GetLimitLow (float *dLL*);

dLL limit low value as float.

Remarks Sets the parameter's low limit value.

C_PAR::SetLimitHigh

void GetLimitHigh (float *dLH*);

DLH limit high value as float.

Remarks Sets the parameter's high limit value.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	PAR object's state as number see e_DATA_States
State Name	R	PAR object's state as text string
Format	R	PAR object's format as string
Physical Unit	R	PAR object's unit as string
Category	R	PAR object's category (EP, PP, …)
Data Value	R	PAR object's value as string according its own format
Data Value	W	Set PAR object's value as string representation
Limit Low	R	PAR object's low limit according its own format
Limit High	R	PAR object's high limit according its own format
Initial Value	R	PAR object's init value according its own format

Class CRegistry

An application built on RTDB has to store some information to be used by the next start-up. The CRegistry class is used for storing EP parameters (see the description in C_PAR class).

The rules for storing the parameters under Windows operating systems are:

– EP are stored in HKEY_CURRENT_USER as EP,

– EP_LM_ADMIN are stored in HKEY_LOCAL_MACHINE as EP_ADMIN,

– EP_LM_USERS are stored in HKEY_LOCAL_MACHINE as EP_USERS.

The default full Registry "path" is SOFTWARE\SW Software\RTDB\Settings\.

For a non-Windows operating system (in principle, UNIX like) special Registry files are used for this purpose:

– EP in .SWdb.reg file in a HOME directory

– EP_LM_ADMIN in .SWdb_ADMIN.reg file in the /root directory

– EP_LM_USERS in .SWdb_USERS.reg file in the /root directory.

– Be aware of a dot at the beginning of the file names (UNIX convention for hidden files).

The max length of strings written into the files is 256 characters.

Remarks The CRegistry class is here documented for supplying the information about storing the EP parameters in the Registry. By programming a run-time system based on RTDB library, especially IO-Handlers or Output Functions there is no direct demand for accessing the EP parameters. It is possible to use the Registry files under Windows. In such a case the environment variable HOME must define the directory path which contains the (user) Registry file and the /root directory must exist. The following similar CRegistryConfig class is more interesting for a user.
It is difficult to find a standard solution for storing EP parameters in embedded systems. In fact, each system has a specific solution.
Note the naming convention used: for a non-Windows environment the Registry is replaced by Registry files.

#include "registry.h"

Runtime Public members

Init	Opens the Registry keys.
Query	Queries a string value from a Registry key.
SetValue	Sets or changes string value in a Registry key.

Member functions

CRegistry::Init

Init(const char* AdminDataPath= NULL);

Remarks Opens three Registry keys: EP, EP_LM_ADMIN or EP_LM_USERS, respectively opens/creates three Registry files in a non Win32 enviroment.

CRegistry::Query

Query(const char* ValueName, CStdString &sData, e_PAR_Categories Category= PG_EP);

Remarks Queries a string value from one of three Registry keys: EP, EP_LM_ADMIN or EP_LM_USERS, respectively gets a string value from Registry files. The default Registry key is the current user (EP).

CRegistry::Set

SetValue(const char* ValueName, CStdString* sData, e_PAR_Categories Category= PG_EP);

Remarks Sets or changes a string value in one of three Registry keys: EP, EP_LM_ADMIN or EP_LM_USERS, respectively sets/changes string values in Registry files. The default Registry key is the current user (EP).

Class CRegistryConf

An application built on RTDB has to store some information to be used by the next start-up. The CRegistryConf class is used for storing configuration file paths.

By start-up the RTDB needs the configuration file ('swd) and two directories: a directory which contains the VFSM specification files (*.iod and *.str) and a directory for SULOG.TXT and TRACE.TXT files. This information is stored under Windows operating system under Registry key SOFTWARE\SW Software\RTDB\Settings\Conf as (for istance): *ConfigFile, DataFilePath and VFSMTypePath*.

For a non-Windows operating system (in principle, UNIX-like) a special Registry file is used for this purpose: .SWdb_Conf.reg.

Be aware of a dot at the beginning of the file name (UNIX convention for hidden files).

The max length of strings written into the files is 256 characters.

Remarks It is possible to build an application which does not use the Registry under Windows. In such a case the environment variable HOME must define the directory path which will contain the Registry file .SWdb_Conf.reg, for instance "c:\Home". The directory must exist before the run-time application starts.
It is difficult to find a standard solution for storing the Configuration data in embedded systems. In fact, each system has a specific solution.

#include "registryconf.h"

Runtime Public members

Init	Opens the Registry keys.
Query	Queries a string value from a Registry key.
SetValue	Sets or changes string value in a Registry key.

Member functions

CRegistryConf::Init

Init();

Remarks Opens the Registry key Conf, respectively opens/creates the Registry files in a non Win32 enviroment.

CRegistryConf::Query

Query(const const char* ValueName, CStdString &sData);

Remarks Queries a string value from the Conf Registry key: ConfFile, DataFilePath or VFSMTypePath, respectively gets a string value from the Registry file.

CRegistryConf::Set

SetValue(const char* ValueName, CStdString* sData,);

Remarks Sets or changes a string value in the Conf Registry key: ConfFile, DataFilePath or VFSMTypePath, respectively sets a string value from the Registry file.

class CQueueReceiver

Instances of the class CQueueReceiver and CQueueSender can only exist in a multithreading environment such as WIN32. They are typically used to connect I/O unit threads with the real-time data base. They are designed for high performance in speed and reliability. A queue receiver can have one or several queue senders. A queue receiver is created with the method ***Create()*** by specifying the number and the size of the queue entries. The entries remain allocated for the lifetime of the queue. After that the senders are connected with the method ***CQueueSender::Connect()***. A queue has two semaphores: one to signal the receiver thread when the queue is not empty anymore and one to signal the sender thread(s) when the queue is not full anymore. A mutex prevents the queue data structure from simultaneous access by sender(s) and receiver.

#include "vsmequ.h"

Initialization - Public Members

Create	Allocates the needed number and size of entries.
GetMessageSize	Gets the size of the queue entries.

Runtime Public Members

Receive	Puts a free packet to the queue and gets a message packet back.
GetBody	Gets a pointer to the data within the message packet.
operator void*	Pointer to the message body.

Member Functions

CQueueReceiver::Create

bool Create(int *nBodySize,* int *nEntries*);

nBodySize	Size of the queue entries in bytes.
nEntries	Number of queue entries.

Remarks Allocates nEntries of message entries with the size of nBodySize and one entry for the receiver itself. The size has to fit to the data that is intended to be sent via the queue.

Return Value true if OK.

CQueueReceiver::GetMessageSize

int GetMessageSize (void);

Remarks A message packet in fact consists of the message body for the user data and additional data for the queue management. This method returns the size of only the message body in bytes.

Return Value Message body size in bytes.

CQueueReceiver::Receive

bool Receive(void);

bool Receive(unsigned long *dwTimeout*);

dwTimeout	Timeout interval in milliseconds.

Remarks The caller thread waits first on the queue mutex for the access to the queue. If the queue is empty it then waits on the receive semaphore until a sender puts a message to the queue or until the dwTimeout expires. If the queue is (or becomes) not empty the callers packet is put to the empty packets and it gets the first or only message packet from the queue. In case of a timeout the caller keeps his packet and the method returns false. Receive() without timeout waits infinitely.

Return Value true if message received, else false when timeout expired.

CQueueReceiver::GetBody

char* GetBody (void);

Remarks Uses the queue as a C string representation of the received message body.

Return Value Pointer to the message body.

CQueueReceiver::operator void*

operator void*(void);

Remarks Same as ***GetBody()*** as operator.

Return Value Pointer to the message body.

class CQueueSender

The description - see CQueueReceiver.

#include "vsmequ.h"

Initialization - Public Members

Connect	Connects itself to a receiver queue.
GetMessageSize	Gets the size of the message body.

Runtime Public Members

Send	Puts a message packet to the queue and gets a free one back.
SendUnic	Sends only if message packet is not there already.
SendLdt	Sends; if queue is full remove the oldest packet.
pstBody	Returns a pointer to the data within the message packet.

Member Functions

CQueueSender::Connect

void Connect(CQueueReceiver* *pQueue*);

pQueue Pointer to the queue receiver to connect to.

Remarks Connects itself to the specified queue. Creates the message packet according to the queue receiver's own message size.

CQueueSender::GetMessageSize

int GetMessageSize (void);

Remarks A message packet in fact consists of the message body for the user data and additional data for the queue management. This method returns the size of only the message body in bytes.

Return Value Message body size in bytes.

CQueueSender::Send

bool Send (void);

Remarks The caller thread waits first on the queue mutex for the access to the queue. If the queue is full it then waits on the send semaphore until the receiver puts a free message packet to the queue. If the queue is (or becomes) not full the caller's packet is put onto the queue and it gets the first or only free message packet from the queue.

Return Value Always true.

CQueueSender::SendUnic

bool SendUnic (void);

Remarks Same as ***Send()*** but send only if packet isn't already in the queue.

Return Value true if entered, false if message packet was already there.

CQueueSender::SendLdt

bool SendLdt (void);

Remarks Same as ***Send()*** but limited. If queue is full (no free packets) remove the oldest done packet. The oldest packet gets lost but it never has to wait.
Return Value true if OK else false.

CQueueSender::pstBody

void* pstBody (void);

Remarks Uses a C anytype pointer representation of the send message body.
Return Value Pointer to the message body.

class C_STR: public CItem

The STR VFSM is used to control a data object used to evaluate strings. In detail, it compares the received string with a regular expression (RE). The result is a “match”, “no-match” or “error”. The regular expression itself can be a DAT, PAR or a hard coded string. The regular expression allows all special characters as known in UNIX tools like sed, awk. This means that also multiple matches are possible, i.e. the compare result “match” can deliver more then one resulting string. The resulting (sub-) string(s) can be stored in other objects such as STR, DAT, PAR or NI. Dependant on the data type of the destination object, the resulting (sub-) string will be converted. In case the conversion is not possible the destination object will be not changed.

A C_STR item object acts as a simple finite state machine. The state table is presented here in a form of a transition matrix (the preambles "SC_" and "SS_" are omitted):

from \ to	OFF	INIT	DEF	ERROR	MATCH	NOMATCH
OFF	-	on	-	-	-	-
INIT	off	-	-	error	match	nomatch
DEF	off	-	-	error	match	nomatch
ERROR	off	-	set	-	-	-
MATCH	off	-	set	-	-	-
NOMATCH	off	-	set	-	-	-

Supported Regular Expressions

RE	Meaning	Example
.	Matches one arbitrary character	a.c matches ‘abc’ but not ‘abbc’
^	Matches the beginning of a string	^ab matches ‘abcd’ but not ‘cdab’
$	Matches the end of a string	ab$ matches ‘cdab’ but not ‘abcd’
\n	n=1..9, matches the same string of characters as was matched by a sub expression enclosed between () preceding the \n. n specifies the n-th sub expression	(ab(cd)ef)A\2 matches ‘abcdefAcd’
( )	sub expression	(\d)A(\d) matches 1A2, 0A4 ...
[ ]	Defines a set of characters to be matched	[a-z] matches ‘s’, ‘w’… but not ‘S’, ‘W’…
[^]	Defines all characters except the characters in the set	[^1-9] matches ‘s’, ‘W’ … but not ‘1’, ‘2’…
(| |)	Matches one of the alternatives	(ab|cd) matches ‘ab’ and ‘cd’
RE+	Matches one or more times the RE	[^1-9]+ matches ‘state*WORKS*’ but not ‘Obj5’
RE?	Matches one or zero times the RE	abc? matches ‘ab’ and ‘abc’
RE*	Matches zero or more times the RE	ab* matches ‘a’, ‘ab’, ‘abb’ …
RE{n}	Matches exactly n times the RE	ab{2} matches ‘abb’ only
RE{n,}	Matches at least n times the RE	ab{2,} matches ‘abb’, ‘abbb’ but not ‘ab’
RE{n,m}	Matches any number of occurrences between n and m inclusive	ab{1,2} matches ‘ab’ and ‘abb’ only

#include "vsstr.h"

Runtime - Virtual Public Members

SetValue	Sets a command to the C_STR object.
GetValue	Gets the state of the C_STR (Control Value).

Member Functions

C_STR::SetValue

virtual void SetValue (int *nCmd*);

nCmd Value of the C_STR Command (casted to e_STR_Cmds, see VSYSTYP.H)

Remarks Is typically called by an item's output action to set one of the C_STR commands. C_STR commands can change the C_STR's status either directly by the command or if the command is a SC_NewDataCmd via the changed input or changed limits.

C_STR::GetValue

virtual int GetValue (void);

Remarks Gets the internal item value, the state of the C_STR. Here, this means the Control Value of the C_STR.

Example The state of the C_STR as a number, to be cast to e_STR_States (see VSYSTYP.H).

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	SWIP object's state as number, see e_SWIP_States
Value	W	SWIP object's command, see e_SWIP_Cmds
State Name	R	SWIP object's state name
Service Mode	X	0->service mode in OFF, 1->ON
Sevice Value	X	SWIP object's service value, see e_SWIP_States
Peripheral Value	R	SWIP's state according input and limits
Limit Low	X	SWIP object's low limit according its own format
Limit High	X	SWIP object's high limit according its own format
Data Value	R	SWIP object's input value

class C_SWIP : public CItem

C_SWIP item objects are objects that divide an arbitrary data value range into three parts and reflect the presence in one of these of the data value by setting the item's internal value. A C_SWIP works together with at least one item of type C_DAT (and of course with all derivations of it like C_NI, C_UDC or C_PAR) as input data. The low- and high-limit values can be constants, or set via configuration or they can also be items of type C_DAT (typically of type C_PAR).

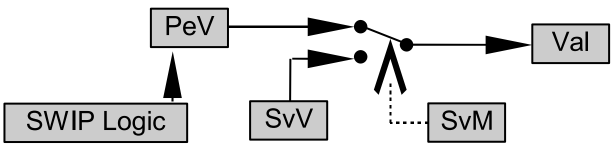
A C_SWIP item object acts as a simple finite state machine. The state table is presented here in a form of a transition matrix (the preambles "SC_" and "SS_" are omitted):

from / to	OFF	LOW	IN	HIGH
OFF	-	On & “low”	On & “in”	On & “high”
LOW	Off	-	“in”	“high”
IN	Off	“low”	-	“high”
HIGH	Off	“low”	“in”	-

“Low”: Input<LimitLow, “in”: LimitLow<Input<LimitHigh, “high”: Input>LimitHigh

In any of the states the commands SC_NewLimitLow, SC_NewLimitHigh and SC_NewInput are allowed. They can lead in the states SS_LOW, SS_IN and SS_HIGH to a change to another of these states.

C_SWIP items contain a feature called service mode that allows (via a client) the item's internal value to be overridden.



If the Service Mode switch *SvM* is false, the Peripheral Value *PeV* is connected to the C_SWIP item's value *Val*. This value is also called the Control Value, because it is the value seen by the control (Vfsm). If the Service Mode switch is true, the service mode is ON and the Service Value *SvV* is connected to the C_SWIP item's value *Val* i.e. the SWIP is disconnected from its peripherals. The Peripheral Value is set by the SWIP's logic according to the input value and the limits. The Service Mode switch and the Service Value are set by the clients (see the C_SWIP item's attributes "SvV" and "SvM").

#include "vsswip.h"

Runtime - Virtual Public Members

SetValue	Sets a command to the C_SWIP object.
GetValue	Gets the state of the C_SWIP (Control Value).

Runtime - Public Members

SetLimits	Sets the C_SWIP item object's low limit and high limit.
SetLimitLow	Sets the C_SWIP item object's low limit.
SetLimitHigh	Sets the C_SWIP item object's high limit.
GetLimitLow	Returns a pointer to a CUniversal object with the low limit.
GetLimitHigh	Returns a pointer to a CUniversal object with the high limit.

Member Functions

C_SWIP::SetValue

virtual void SetValue (int *nCmd*);

nCmd	Value of the C_SWIP Command (casted to e_SWIP_Cmds, see VSYSTYP.H)

Remarks
Is typically called by an item's output action to set one of the C_SWIP commands. C_SWIP commands can change the C_SWIP's status either directly by the command or if the command is a SC_NewDataCmd via the changed input or changed limits.

C_SWIP::GetValue

virtual int GetValue (void);

Remarks
Gets the internal item value, the state of the C_SWIP. This means here the Control Value of the C_SWIP.

Example
The state of the C_SWIP as a number, to be cast to e_SWIP_States (see VSYSTYP.H).

C_SWIP::SetLimits

void SetLimits (float *fLimLow*, float *fLimHigh*);

fLimLow/High	Low and High Limit Values.

Remarks
Typically used by a user written output function to set the C_SWIP's low and high limits. Possibly causes a state change. The limit values are transformed from the float of the parameter to the same internal representation as the C_SWIP's input value.

Example
```
C_PAR*  pPar  = (C_PAR*) (pOwner->GetAssItem(cPLim) );
C_SWIP* pSwip = (C_SWIP*)( pOwner->GetAssItem(cSwip) );
C_NO*   pAo   = (C_NO*)  (pOwner->GetAssItem(cAo) );
float   fLimH = pPar->fGetData();
float   fLimL = -fLimH;
float   fOVal;

  pPar  = (C_PAR*)(pOwner->GetAssItem(cPVal) );
  fOVal = pPar->fGetData();
  fLimH += fOVal;
  fLimL += fOVal;
  pSwip->SetLimits (fLimL, fLimH);
  pAo->SetData (fOVal);
```

C_SWIP::SetLimitLow

void SetLimitLow(float *fLimLow*);

fLimLow	Low limit values.

Remarks
Typically used by user written output functions to set the C_SWIP's low limit. Possibly causes a state change. The limit value is transformed from the float of the parameter to the same internal representation as the C_SWIP's input value.

C_SWIP::SetLimitHigh

void SetLimitHigh(float *fLimHigh*);

fLimHigh	High limit values.

Remarks
Typically used by user written output functions to set the C_SWIP's high limit. Possibly causes a state change. The limit value is transformed from the float of the parameter to the same internal representation as the C_SWIP's input value.

C_SWIP::GetLimitLow

CUniversal* GetLimitLow (void);

Remarks Returns a pointer to a CUniversal object containing the low limit value in the SWIP object's own format.

C_SWIP::GetLimitHigh

CUniversal* GetLimitHigh (void);

Remarks Returns a pointer to a CUniversal object containing the high limit value in the SWIP object's own format.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	SWIP object's state as number, see e_SWIP_States
Value	W	SWIP object's command, see e_SWIP_Cmds
State Name	R	SWIP object's state name
Service Mode	X	0->service mode in OFF, 1->ON
Sevice Value	X	SWIP object's service value, see e_SWIP_States
Peripheral Value	R	SWIP's state according input and limits
Limit Low	X	SWIP object's low limit according its own format
Limit High	X	SWIP object's high limit according its own format
Data Value	R	SWIP object's input value

class C_TAB : public CItem

A C_TAB is a table item. It has no data value of its own. It has an array with pointers to data items (typically parameters). Its state (the item's internal value) is the index to this array. The state is set directly by the virtual method ***SetValue().*** So it switches between the several data items. Thus, it acts as a multiplexer that maps several C_DAT items (or derived) to one. To an item that uses it (typically C_NO, C_CNT, C_SWIP) it looks like a C_DAT item.

Although for a C_TAB item only derivations of the item type C_DAT (typically parameters) make any sense, it accepts every item type. In the worst case it gets a value of zero from them.

#include "vstab.h"

Runtime Virtual Public Members

SetValue	Sets the C_TAB object's internal value, the index.
GetData	Gets the indexed data as a universal data object.

Member Functions

C_TAB::SetValue

virtual void SetValue (int *nVal*);

nVal	Index to the item array (0 based), typically a virtual output value.

Remarks Is typically called by a Vfsm's output function or by a user written output function. It sets the index to the according data item. Items that use the TAB (e.g. a NO) are advised at the change and can get the new data with GetData().

C_TAB::GetData

virtual CUniversal* GetData (void);

Remarks Gets a pointer to the universal data object value that is indexed by the internal value. If out of range or not available it returns a dummy data with long integer.

Return Value A pointer to the CUniversal object.

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	X	TAB object's value

class C_TI : public C_CNT

C_TI is the class of timer objects. It contains all the data and methods to handle the timer aspect of an application. C_TI is a subclass of the item type Counter (C_CNT). It inherits the counter's state machine and adds the aspect of the timebase. It has no other public interface.

#include "vstim.h"

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	Timer's state as number see e_CNT_States
Value	W	Timer's commands as number; see e_CNT_Cmds
State Name	R	Timer's state as a text string
Count Constant	X	as an integer number
Count Register	R	as an integer number
Physical Unit	R	The timer's timebase: "100ms", "sec" or "min"

class C_UDC : public C_DAT

The Up/Down Counter item objects C_UDC are objects derived from the class C_DAT. They hold a data value of type long (Sign plus 31 bits) used as a counter value. Different from other counter objects (C_CNT, C_TI or C_ECNT) they have no counter constant and do not return states as OVER or RESET. Their state is derived from those of C_DAT such as DS_CHANGED or DS_DEF. The counter can be monitored with a C_SWIP object and has a range from negative to positive values. C_UDC adds to C_DAT attributes (Data Value, Physical Unit and Format) the references to up to three item objects that can deliver the Clear-, Up- or Down-triggers by their internal values. The method ***SetValue()*** sets a command (UC_Clear, UC_Up and UC_Down, see e_UDC_Cmds in VSYSTYP.H). There are two ways to control a C_UDC item: via a Vfsm's virtual output and via an item's ItemAdviseList. In both cases the C_UDC's method ***SetValue()*** executes an appropriate command.

#include "vsudc.h"

Runtime - Virtual Public Members

SetValue	Sets a command to the C_UDC object.
GetValue	Returns the state of a C_UCD object.

Member Functions

C_UDC::GetValue

virtual int GetValue (void);

Remarks Gets the internal value, the state of the C_UDC item. This values can be one of the enum e_DATA_States (DS_OFF = 0, DS_DEF, DS_CHANGED, DS_INIT).

Return Value C_UDC state as number (see e_DATA_States is SYSTYP.H).

C_UDC::SetValue

virtual void SetValue (int *nCmd*);

nCmd	Value of the C_UDC command as number (cast to e_UDC_Cmds).

Remarks

Is typically called by item's ItemAdviseList, by Vfsm's virtual output or by user written output functions to set one of the C_UDC commands (e_UDC_Cmds see VSYSTYP.H). UDC commands change the C_UDC item's status to DS_CHANGED and of course the value of the counter (the data value).

Host Interface

Attribute	Acc.	Description (enumeration see VSYSTYP.H)
Value	R	UDC object's state as number see e_DATA_States
Value	W	UDC object's command see e_UDC_Cmds
State Name	R	UDC object's state as text string
Format	R	UDC object's format as string (always "long")
Physical Unit	R	UDC object's unit as string
Data Value	R	UDC object's value as string in long format

Remarks

PhysicalUnit come via the Configuration File from the state*WORKS* Studio/DAT-Properties to the data item object. PhysicalUnit is not used by the VFSM System. Format is always long.
Data set (poked) by a destination application can be in any valid format. If the format isn't valid the data value is not changed and the old value is sent as update.

class C_UNIT : public CItem

The C_UNIT item is an object that collects the I/O objects for a physical IO-Handler. The C_UNIT has an AssItemList to hold the connection to the I/O objects, a unit type, a physical address and a communication port name. It gets these things from the Configuration File during the System Startup Phase Create, and it makes it accessible to the IO-Handlers of the IO-Handler. It is the link between the configuration and the IO-Handler.

IO objects are basically the item types C_DI, C_DO, C_NI and C_NO. A C_UNIT item typically comprises items of these types because they hold connections to real physical I/Os. But there is no limitation to that. A C_UNIT item can hold every item type. In practice, for instance C_CMD C_XDA and C_PAR item types are often used in the C_UNIT.

The C_UNIT class is a derivation of CItem, but it makes no use of its data.

#include "vsunit.h"

Public Members

GetUnitTypeName	Returns the name of the unit type (the IOD-File name).
GetPhysicalAddress	Returns the configured physical address.
GetCommPort	Returns the configured communication port name.
GetNumbAssItem	Returns the number of associated item.
GetAssItem	Returns a pointer to the indexed associated item.

Member Functions

C_UNIT::GetUnitTypeName

CString GetUnitTypeName (void);

Remarks Returns the name of the IO-Handler type that is intended to cooperate.
Return Value IO-Handler type as string.

C_UNIT::GetPhysicalAddress

int GetPhysicalAddress (void);

Remarks This function is typically used by IO-Handlers to appropriately address the IO-hardware.
Return Value Returns the C_UNIT object's physical address.

C_UNIT::GetCommPort

CString GetCommPort (void);

Remarks Returns the communication port as string representation.
Return Value Communication port, e.g. "COM1".

C_UNIT::GetNumbAssItem

int GetNumbAssItem (void);

Remarks Returns the maximum number of associated items of that C_UNIT object.
Return Value Number of associated items.

C_UNIT::GetAssItem

CItem* GetAssItem (int *nObjID*);

nObjID Index to an associated item.

Remarks Returns the indexed associated item. Index is the number of the item within the UNIT. Returns NULL if number is out of bounds.

Return Value Pointer to the indexed associated item or NULL if none.

Host Interface

Attribute	Acc.	Description
Assoc Item List	R	Associated item list
Type Name	R	Type of the cooperating IO-Handler as string
Physical Addr	R	Physical address of the IO-Handler's hardware as integer
CommPort	R	Communication port as string

Remarks AssocItemList is a list of the items that are collected by the UNIT. The list has the form:
`"PAR PollTime\nDI IOUnit1_Di0\nDI IOUnit1_Di1\n <and so on>\n"`
PAR is the item type and PollTime is the item name. Type and name are separated by a blank. The items are separated by the <NL>characters (new line = 0x0A).
Type Name, Physical Addr and CommPort come via the Configuration File from the state*WORKS* Studio/UNIT-Properties to the unit item object. They are not used by the VFSM System.

class CUniversal

CUniversal objects are objects holding values in various formats. This data type is typically used by C_DAT item objects (and the derivations C_PAR, C_NI, C_NO). CUniversal objects have as attributes the enumeration *Format* and the union *Data* that stores the data in the appropriate format at the same physical memory address. This union has a size of 4 bytes. Of course when the format is char, three Bytes are wasted. There are operators and methods to copy objects with different formats. Copying from bigger formats to smaller (e.g. long to char) is done by limiting to the smaller object's value range (e.g. 900 to a char is 127, -10e37 to a boolean is 0).

#include "vsuni.h"

Initialization - Virtual Public Members

Create	Sets the CUniversal object's format.

Runtime - Public Members

bGetData	Returns the data value converted and limited to 1Bit.
chGetData	Returns the data value converted and limited to signed 8Bit.
uchGetData	Returns the data value converted and limited to unsigned 8Bit.
nGetData	Returns the data value converted and limited to signed 16Bit.
unGetData	Returns the data value converted and limited to unsigned 16Bit.
lGetData	Returns the data value converted and limited to signed 32Bit.
fGetData	Returns the data value converted to 32Bit float.
CopyConvert	Copies one object's data to another. Convert and limit to appropriate format.
Display	Makes a string representation of the data in appropriate format.
Set	Sets the object's data from a string representation.
SetData	Sets the object's data when string format.
GetFormat	Returns the object's data format as enumeration.
stGetFormat	Returns the object's data format as string representation.
operator=	Overloads, copies data of several formats to the object's data
IsGreaterThan	Compares two object's data independent of their format.
IsSmallerThan	Compares two object's data independent of their format.

Data - Public Members

m_Data	Data structure direct access in r_Universal (see VSYSTYP.H).

Member Functions

CUniversal::Create

void Create (e_DATA_Formats *Format*);

void Create (CString *stFrm*);

Format	Format specification as enumeration (see e_DATA_Formats in VSYSTYP.H).
stFrm	Format specification as string representation.

Remarks Sets the format of the object's data either directly or via a string. It can be applied after instanciation to set the format the first time, or any time to change the format of the object's data. The data value remains if the value fits to the format; otherwise it is limited to the new format's limits.

CUniversal::xGetData

bool bGetData (void);

char chGetData (void);

unsigned char uchGetData (void);

short int nGetData (void);

unsigned short unGetData (void);

long lGetData (void);

float fGetData (void);

Remarks Returns the data value in the appropriate format independent of the object's data format. Possibly limited to the return value's limits.

Return Value CUniversal object's data value in the appropriate format.

CUniversal::CopyConvert

void CopyConvert (CUniversal* *pSrc*);

pSrc Pointer to the source object.

Remarks Copies the value of the source object's data (in the format for that object) into the data of the destination object, in the format for the destination. The value is possibly limited to this format's limits, so data loss could occur.

CUniversal::Display

bool Display(CString* *pstVal*);

pstVal Pointer to the string object to copy the data to.

Remarks Makes a string representation of the object's data according to its format.
true if OK, false if format not supported.

CUniversal::Set

bool Set(CString* *pstVal*);

pstVal Pointer to the string object to take the data from.

Remarks Puts the string representation of a value to its appropriate format. Applies the limits and format according to the value. Tests whether the value has changed. In case of a format error leaves the data value unchanged, but returns true (use to fake changed data).

Return value Returns true if the data has changed.

Example
```
CUniversal data;
CString    st = "1.2345";
 data.Create(DF_FLOAT);
 data.Set(&st);
```

CUniversal::SetData

void SetData (CString& *stData*);

stData String object to copy to CUniversal object's string object.

Remarks Copies the string object to the object's data if its format is DF_STRING, else do nothing

CUniversal::GetFormat

e_DATA_Formats GetFormat (void);

Remarks Returns the CUniversal object's data format as enumeration (see VSYSTYP.H).

Return value Data format.

CUniversal::stGetFormat

CString* stGetFormat (void);

Remarks Returns the CUniversal object's data format as string representation.
Return value Data format as string.

CUniversal::operator=

CUniversal operator= (bool *b*);

CUniversal operator= (char *ch*);

CUniversal operator= (unsigned char *uch*);

CUniversal operator= (short int *n*);

CUniversal operator= (unsigned short *un*);

CUniversal operator= (long *l*);

CUniversal operator= (float *f*);

CUniversal operator= (CUniversal& *Uni*);

b, ch, uch, n, un, l, f, Uni CUniversal object's data new value in appropriate format.

Remarks Overloads the = Operator, so that several formats can be copied to a CUniversal object's data. Too large values are limited to the limits of the destination's format.

CUniversal::IsGreaterThan

CUniversal::IsSmallerThan

bool IsGreaterThan(CUniversal& *Uni*);

bool IsSmallerThan(CUniversal& *Uni*);

Uni Object to compare.

Remarks Compares the current CUniversal object's data value with the specified one. Takes the current format.
Return value According to the appropriate fact.

CUniversal::m_Data

r_Universal *m_Data*;

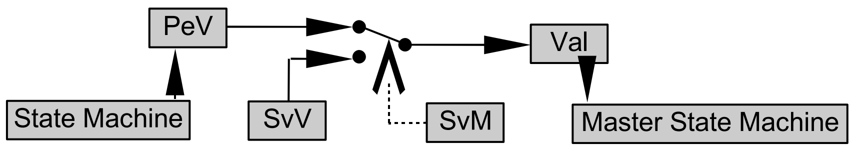
m_Data CUniversal object's data structure.

Remarks Direct access to the object's data structure.

class C_VFSM : public CItem

A C_VFSM item object is an object that stores the state of the Vfsm. It is the incarnation of the Vfsm. The state is simply the item's internal value. Additional data held here are the Virtual Input (VI), a reference to the Vfsm-Type object (state table, IO objects) and others.

C_VFSM items contain a service mode that allows (via a client) the state to be overridden:



If the Service Mode switch *SvM* is false, the real state of the Vfsm (here called the Peripheral Value) *PeV* is connected to the C_VFSM item's value *Val*. This value is also called the Control Value, because it is the value seen by the control (the master Vfsm). If the Service Mode switch is true, the service mode is ON and the Service Value *SvV* is connected to the C_VFSM item's value *Val* i.e. the VFSM item is disconnected from its state machine. The Peripheral Value is set by the Vfsm-Executor. The method ***GetValue()*** returns the item's value *Val*. The method ***GetState()*** always gets the state machine's state *PeV*. The Service Mode switch and the Service Value are set by the clients (see the C_DI item's attributes "SvV" and "SvM").

Note, that the user will never use the methods of the C_VFSM class because there is no application for them in programming IO-Handlers or Output Functions. The C_VFSM class is documented here only for training purposes to explain the functioning of the VFSM Executor.

#include "vsvfsm.h"

Runtime Virtual Public Members

SetValue	Enters the specified VI name, clears first the whole class.
ResetValue	Removes the specified VI name.
GetValue	Returns the state of the Vfsm including the service mode.
GetState	Returns the state of the Vfsm, no service mode.
GetUnitTypeName	Returns the name of the C_VFSM's type (specification, state table).
GetNumbAssItem	Returns the number of associated item
GetAssItem	Returns a pointer to the indexed associated item.

Member Functions

C_VFSM::SetValue

virtual void SetValue (int *nVI*);

nVI The VI name as a number.

Remarks Is typically called by other item objects to set a VI name to the VI-set of this C_VFSM item. It removes the class belonging to the nVI from the C_VFSM's VI set and then it enters the nVI and calls the executor.

C_VFSM::ResetValue

virtual void ResetValue (int *nVI*);

nVI The VI name as number.

Remarks Is typically called by other items to remove the VI class of the appropriate VI name from this C_VFSM's VI-set. Although the VI-set changes, the executor is not called.

C_VFSM::GetValue

virtual int GetValue (void);

Remarks Returns the item's internal value. This means here the state of the Vfsm. It depends on Service Mode switch, Service Value or the Vfsm's state machine.

C_VFSM::GetState

int GetState (void);

Remarks Is typically called by the Vfsm-Executor to look for the state of the Vfsm it is executing. The state machine's state returned here is the real state, independent of the service mode.

C_VFSM::GetUnitTypeName

virtual CString GetUnitTypeName (void);

Remarks Returns the name of the C_VFSM’s type as a string object. The name is the one of the appropriate state table specification and IO description file (*.STR, *.IOD).

C_VFSM::GetNumbAssItem

int GetNumbAssItem (void);

Remarks Returns the maximum number of associated items of that C_VFSM object.
Return Value Number of associated items

C_VFSM::GetAssItem

CItem* GetAssItem (int *nObjID*);

nObjID Index to an associated item.

Remarks Returns the indexed associated item. Index is the number of the item within the VFSM. Returns NULL if number is out of bounds.
Return Value Pointer to the idexed associated item or NULL if none.

Host Interface

Attribute	Acc.	Description
Value	R	State or ServiceValue as numb, see List or IOD-file.
State Name	R	State machine's state name as string
Virtual Input	R	Virtual Input, a set
Service Mode	X	0 -> service mode OFF, 1 -> ON
ServiceValue	X	As numbers, see List or IOD-file.
Peripheral Value	R	State machine's state as number
Assoc Item List	R	Associated item list
TypeName	R	Type of the Vfsm as string
List	R	State names as a list
Run Mode	X	0->FreeRun, 1->Hold, 2->Step
Next Step	R	In Hold-Mode next possible transition else “none”

Remarks

Virtual Input is displayed as a set of VI names. A VI string looks like "{1,4,5}". The VI names are separated by commas. The numbers refer to the I-Block in the according IOD-file.
Assoc Item List is a list of the items that are owned by the VFSM. The list has the form:
"CMD Stepper1_MyCmd\nTI Stepper1_Tim\n DI Stepper1_DiStart\nDO Stepper1_Do1\n"
CMD is the item type and Stepper1_MyCmd is the name. Type and name are separated by a blank. The items are separated by <NL>characters (new line = 0x0A).
List delivers a string containing names of states ordered by state numbers:
"Init\nOFF\nStep1Busy\nSTEP1\nStep2Busy\nSTEP2\n"
Init is state number 1, OFF is 2 and so on. \n is the <NL>character (new line = 0x0A).
Run Mode = Step means that the state machine performs one state change or one set of due input actions or both.

class C_XDA : public CItem

C_XDA item objects hold a certain amount of memory used typically by user written output functions or I/O-Units. They hold as normal items an internal value. The internal value is of integer type. It is set by the Vfsm's output function or by user written output functions. On the other hand the internal value can become a virtual input of the same or another Vfsm.

Warning: The C_XDA item object must not be used as an input and as an output in the same state machine, especially in the same state.

#include "vsxda.h"

Runtime - Virtual Public Members

SetValue	Sets the internal value to the C_XDA object.
GetValue	Returns the internal value of a C_XDA object.
pMemory	Returns the pointer to the auxiliary memory.
nSize	Returns the size of the auxiliary memory.

Member Functions

C_XDA::SetValue

virtual void SetValue (int *nVal*);

nVal Value of the C_XDA.

Remarks Is typically called by Vfsm's virtual output or by user written output functions.

C_XDA::GetValue

int GetValue (void);

Remarks Gets the internal value of the C_XDA item.

Return Value C_XDA value as integer number.

C_XDA::pMemory

void* pMemory (void);

Remarks Gets the pointer to the C_XDA item's auxilary memory block.

Return Value (Any type) pointer to a memory block.

C_XDA::nSize

int nSize (void);

Remarks Gets the size to the C_XDA item's auxilary memory block. This size is set via the Configuration File from the state*WORKS* Studio/XDA-Properties.

Return Value Size of memory block in bytes.

Host Interface

Attribute	Acc.	Description
Value	X	Value of the XDA.

class CVfsmSystem

CVfsmSystem is the class that contains the whole Vfsm system. It acts mainly during the Startup Phase of the system. It can be adapted to a WindowsNT environment or to any other OS.

#include <vswin.h>

Initialization - Public Members

Create	Builds up the whole system.
RemoveAll	Removes all dynamic data, system remains.
AttachToMainWindow	Installs the connection to the container window object.
TimeBaseTick	Updates the VFSM System's time base.
ItemNumb	Gets the number of a specified item type in the VFSM System.
FirstItem	Looks for the first item of a specified type in the VFSM System.
NextItem	Iterates through all items of a specified type
PollAdviseQueue	Empties the VFSM System's event advise queue.

Member Functions

CVfsmSystem::Create

bool Create (CString *stVfsmTypePath*, CString *stDataFilePath*, CString *stConfigName*);

stVfsmTypePath	Directory with the *.STR and *.IOD files.
stDataFilePath	Directory for the log files.
stConfigName	Directory and name of the Configuration File.

Remarks This method builds up the whole VFSM System. It installs the global database and system resources access. It performs the System Startup Phase Create, Connect and at last Initialize.

Return Value false if the system isn't able to work: true, there can be errors in configuration but the system can work.

CVfsmSystem::RemoveAll

void RemoveAll(void);

Remarks Deletes the database and all dynamic data by calling the appropriate object's method delete. After that, the system is ready to perform a new ***Create()***.

CVfsmSystem::AttachToMainWindow

void AttachToMainWindow (CWnd* *pPar*);

pPar	Pointer to the parent (main) window.

Remarks Sets the Parent Window to the VFSM System in case the OS is WindowsNT. This is the environment for the Host Client/Server connection.

CVfsmSystem::ItemNumb

int ItemNumb (e_ItemTypes *eItemType*) ;

eItemType	Specification of the item type (see VSYSTYP.H).

Remarks Returns the number of a specified item type currently available in the database.

Return Value Number of instances of the specified item type.

CVfsmSystem::FirstItem

CItem* FirstItem (e_ItemTypes *eItemType*, POSITION& *pos*);

eItemType	Specification of the item type (see VSYSTYP.H).
pos	Iterator

Remarks Returns the first item of a specified item type currently instanciated in the database.

Return Value Pointer to the first instances of the specified item type. NULL if not there.

CVfsmSystem::NextItem

CItem* NextItem (e_ItemTypes *eItemType*, POSITION& *pos*)

eItemType	Specification of the item type (see VSYSTYP.H).
pos	Iterator

Remarks Returns the next item of a specified item type currently instanciated in the database.

Return Value Pointer to the next instances of the specified item type. NULL if at the end of them.

See Also ***CVfsmSystem::FirstItem***

Example

```
CVfsmSystem*  pVS = &(pDoc->m_VfsmSystem);
C_PAR*        pPar;
POSITION      pos;

  pPar = (C_PAR*)pVS->FirstItem (IT_PAR, pos);
  while ( pPar != NULL )
  {
   // do something with the pPar object
   pPar = (C_PAR*)pVS->GetNextItem (IT_PAR, pos);
  }
```

CVfsmSystem::PollAdviseQueue

bool PollAdviseQueue(void);

Remarks This method has to be called at the Window's message loop idle time. It gets the advise requests one by one from the advise queue and passes them to the Host PostAdvise() method. If the queue is empty it does nothing.

Return Value true if an element was removed from the queue. false when the queue is empty.

C H A P T E R 8

Declarations

Declarations
All generally relevant declarations are in the file VSYSTYP.H.

VSYSTYP.H

This file is the declaration of all globally used enumeration and data structures in the VFSM System. Many of them belong to a specific class. There are some rules to know:

- The tables showing states or commands of a certain item types have a row called Name. It shows the string representation of the attribute ".StN" (state name).
- Enumeration values with _LAST or _Last are not really used, they serve only as limits in the program code.

enum e_ItemTypes

The enumeration represents the Real-time Database's item types. *Type Name* is the string representation used, e.g. in the Configuration File, as an item type identifier.

Enumeration	**Num**	**Type Name**
IT_Item	0	Item
IT_VFSM	1	VFSM
IT_CMD	2	CMD
IT_TI	3	TI
IT_AL	4	AL
IT_DI	5	DI
IT_DO	6	DO
IT_NO	7	NO
IT_NI	8	NI
IT_SWIP	9	SWIP
IT_XDA	10	XDA
IT_PAR	11	PAR
IT_OFUN	12	OFUN
IT_STR	13	STR
IT_CNT	14	CNT
IT_DAT	15	DAT
IT_UNIT	16	UNIT
IT_ECNT	17	ECNT
IT_UDC	18	UDC
IT_TAB	19	TAB
IT_last	20	END

enum e_ItemAttributes

This enumeration represents the summary of available attributes. A particular item type typically uses only a subset of it. The following table shows all the attributes. *Enumeration* is the text used in a C++ program, *Number* is the numeric representation for instance used in the TCP/IP host communication, *Attribute Name* is the way attributes are mentioned in text and the abreviation *Short* is the text used in the Host interface (TCP/IP or DDE) as extension to the item name (e.g. "DI_DoorOPEN.Val).

Enumeration	Number	Attribute Name	Short
IAtt_None	0	Value	
IAtt_Value	1	Value	Val
IAtt_ServiceMode	2	Service Mode	SvM
IAtt_ServiceValue	3	Service Value	SvV
IAtt_PeripheralValue	4	Peripheral Value	PeV
IAtt_VI	5	Virtual Input	VI
IAtt_StateName	6	State Name	StN
IAtt_AssocItemList	7	Associated Item List	AIL
IAtt_TypeName	8	Type Name	Typ
IAtt_CountConstant	9	Count Constant	CnC
IAtt_CountRegister	10	Count Register	CnR
IAtt_Category	11	Category	Cat
IAtt_Format	12	Format	Frm
IAtt_PhysicalUnit	13	Physical Unit	Uni
IAtt_LimitLow	14	Limit Low	LiL
IAtt_LimitHigh	15	Limit High	LiH
IAtt_InitValue	16	Initial Value	IVa
IAtt_DataValue	17	Data Value	Dat
IAtt_Text	18	Text	Txt
IAtt_Acknowledge	19	Acknowledge	Ack
IAtt_Time	20	Time	Tim
IAtt_ScaleFactor	21	Scale Factor	ScF
IAtt_Offset	22	Offset	Ofs
IAtt_ScaleMode	23	Scale Mode	ScM
IAtt_List	24	List	Lst
IAtt_PhysAddr	25	Physical Addr	PAd
IAtt_CommPort	26	CommPort	Com
IAtt_Trace	27	Trace	Trc
IAtt_RunMode	28	Rune Mode	RMo
IAtt_NextStep	29	Next Step	NSt

enum e_DATA_Formats

The enumeration represents the C_DAT item's data value format. *Format Name* is the string representation in the C_DAT item's attribute "Frm".

e_DATA_Formats	**Num**	**Format Name**
DF_none	0	None
DF_BOOL	1	bool
DF_CHAR	2	char
DF_UCHAR	3	uchar
DF_INT	4	int
DF_UINT	5	uint
DF_LONG	6	long
DF_FLOAT	7	float
DF_STRING	8	string
DF_PTR	9	ptr
DF_PITEM	10	pItem
DF_LAST		

union r_Universal

This data structure stores several data formats in the same 32Bit. It is used typically when working with C_DAT item and CUniversal objects.

```
union r_Universal
{
 bool           b;
 char           ch;
 unsigned char  uch;
 unsigned short un;
 short int      n;
 long           l;
 float          f;
 void*          ptr;
 CItem*         pItem;
 CString*       pst;
}; /* End of r_Universal */
```

enum e_PAR_Categories

The enumeration represents the C_PAR item's category. *Cat Name* is the string representation in the C_PAR item's attribute "Cat".

e_PAR_Categories	Num	Remark (stored in Registry)	Category Name
PG_none	0		None
PG_EP	1	HKEY_CURRENT_USER as EP	EP
PG_PP	2	not stored in Registry	PP
PG_EP_LM_ADMIN	3	HKEY_LOCAL_MACHINE as EP_ADMIN	EP_LM_ADMIN
PG_EP_LM_USERS	4	HKEY_LOCAL_MACHINE as EP_USERS	EP_LM_USERS
PG_PP_Coded	5	not stored in Registry; value set by IO-Handler Code	PP_Coded
PG_Last	6		

enum e_NIO_ScaleModes

The enumeration represents the C_NI and C_NO item's Scale Mode. *Scale Name* is the string representation in the C_NI/O item's attribute "ScM".

e_NIO_ScaleModes	Num	Scale Name
SM_None	0	None
SM_Lin	1	Lin
SM_Log	2	Log
SM_Exp	3	Exp
SM_Sin	4	Sin (not used)
SM_ASin	5	ASin (not used)
SM_Last		

enum e_UDC_Cmds

The enumeration represents the C_UDC item's command.

e_UDC_Cmds	Num
UC_none	0
UC_Clear	1
UC_Up	2
UC_Down	3
UC_Last	

enum e_CNT_States, e_CNT_Cmds

The two enumerations represent the C_CNT and the derived item's state (internal value) and command.

e_CNT_States	**Num**	**Name**
CS_none	0	none
CS_RESET	1	RESET
CS_STOP	2	STOP
CS_RUN	3	RUN
CS_OVER	4	OVER
CS_OVERSTOP	5	OVERSTOP
CS_LAST		

e_CNT_Cmds	**Num**
CC_none	0
CC_Reset	1
CC_Stop	2
CC_Start	3
CC_ResetStart	4
CC_IncCounter	5
CC_DecCounter	6
CC_NewCountConst	7
CC_Last	

enum e_STR_States, e_STR_Cmds

The two enumerations represent the C_STR item's state (internal value) and command.

e_CNT_States	Num	Name
STS_none	0	none
STS_OFF	1	OFF
STS_INIT	2	INIT
STS_MATCH	3	MATCH
STS_NOMATCH	4	NOMATCH
STS_DEF	5	DEF
STS_ERROR		ERROR
STS_LAST		

e_CNT_Cmds	Num
STC_none	0
STC_Off	1
STC_On	2
STC_Set	3
STC_NewInput	4
STC_NewRegExp	5
STC_NewSubStr	6
STC_Last	

enum e_SWIP_States, e_SWIP_Cmds

The two enumerations represent the C_SWIP item's state (internal value) and command.

e_SWIP_States	Num	Name
SS_none	0	none
SS_OFF	1	OFF
SS_LOW	2	LOW
SS_IN	3	IN
SS_HIGH	4	HIGH
SS_LAST		

e_SWIP_Cmds	Num
SC_none	0
SC_Off	1
SC_On	2
SC_NewLimitLow	3
SC_NewLimitHigh	4
SC_NewInput	5
SC_Last	

enum e_ALA_States, e_ALA_Cmds

The two enumerations represent the C_AL item's state (internal value) and command.

e_ALA_States	Num	Name
AS_NONE	0	none
AS_COMING	1	COMING
AS_GOING	2	GOING
AS_STAYING	3	STAYING
AS_ACKNOWLEDGE	4	ACK
AS_COM_GO	5	COM_GO
AS_LAST		

e_ALA_Cmds	Num
AC_None	0
AC_Coming	1
AC_Going	2
AC_Staying	3
AC_Acknowledge	4
AC_Cancel	5
AC_Last	

enum e_DATA_States, e_DATA_Cmds

The two enumerations represent the C_DAT item's state (internal value) and command.

e_DATA_States	Num	Name
DS_OFF	0	OFF
DS_UNDEF	1	UNDEF
DS_DEF	2	DEF
DS_CHANGED	3	CHANGED
DS_INIT	4	INIT
DS_SET	5	SET
DS_LAST		

e_DATA_Cmds	Num
DC_none	0
DC_Off	1
DC_On	2
DC_Set	3
DC_NewData	4
DC_Last	5

Index