SIEMENS

SIMATIC

WinAC Open Development Kit (ODK)

User Manual

Edition: 2

Safety Guidelines

This manual contains notices which you should observe to ensure your own personal safety, as well as to protect the product and connected equipment. These notices are highlighted in the manual by a warning triangle and are marked as follows according to the level of danger:

Danger

indicates that death, severe personal injury or substantial property damage **will** result if proper precautions are not taken.

Warning

indicates that death, severe personal injury or substantial property damage **can** result if proper precautions are not taken.

Caution

indicates that minor personal injury or property damage can result if proper precautions are not taken.

Note

draws your attention to particularly important information on the product, handling the product, or to a particular part of the documentation.

Qualified Personnel

Only **qualified personnel** should be allowed to install and work on this equipment. Qualified persons are defined as persons who are authorized to commission, to ground, and to tag circuits, equipment, and systems in accordance with established safety practices and standards.

Correct Usage

Note the following:

Warning

This device and its components may only be used for the applications described in the catalog or the technical descriptions, and only in connection with devices or components from other manufacturers which have been approved or recommended by Siemens.

This product can only function correctly and safely if it is transported, stored, set up, and installed correctly, and operated and maintained as recommended.

Trademarks

SIMATIC®, SIMATIC HMI® and SIMATIC NET® are registered trademarks of SIEMENS AG.

Some of other designations used in these documents are also registered trademarks; the owner's rights may be violated if they are used by third parties for their own purposes.

Siemens AG
Automation and Drives (A&D)
Industrial Automation Systems (AS)
Postfach 4848, D- 90327 Nürnberg

© Siemens AG 1999
Technical data subject to change.

Siemens Aktiengesellschaft

Table of Contents

Preface

The Windows Automation Center (WinAC) Basis Open Development Kit (ODK) provides you with the tools for creating custom software that interacts directly with the Windows Logic Controller (WinLC) program scan cycle. This interaction uses Microsoft's COM (Component Object Model) technology: your custom software is a COM object that is loaded as a DLL within the WinLC process.

WinAC Basis ODK consists of the following elements:

- COM extension for WinLC version 3.0
- Application wizard for Visual C++ version 6.0
- STEP 7 library containing SFBs for interacting with COM objects
- WinAC Basis ODK User Manual (electronic)
- Sample program

Audience

This manual is intended for engineers, programmers, and maintenance personnel who have a general knowledge of programmable logic controllers and who are proficient in C or C++. Knowledge of STEP 7 programming and of WinLC is also required.

Scope of the Manual

This manual describes the features and the operation of version3.0 of the WinAC Open Development Kit (ODK).

How to Use This Manual

This manual provides the following information:

- Overview of WinAC Basis ODK
- Installing WinAC Basis ODK
- Getting started a sample application program
- Creating a COM object (including the WinAC Basis ODK application wizard and C/C++ Elements)
- Creating the STEP 7 program

Other Manuals

For additional information, refer to the following manuals:

Title	Content
Windows Logic Controller (WinLC) User Manual	This manual provides basic information about the performance characteristics and operation of the WinLC controller.

Additional Assistance

If you have any questions not answered in this or one of the other STEP 7 manuals, if you need information on ordering additional documentation or equipment, or if you need information on training, please contact your Siemens distributor or sales office.

Contacting Customer Support

You can find additional information about WinAC Basis ODK and updates to this user manual at the Siemens Energy & Automation web site:

- www.sea.siemens.com/industrialsoftware

This web site includes useful information, such as application notes, in the Technical Service area. This area also includes downloadable software, including sample programs for linking a Delta Tau motion control board with WinLC.

Customer Support	
North America	
Telephone	(609) 734-6500 (609) 734-3530
E-mail	ISBU.Hotline@sea.siemens.com simatic.hotline@sea.siemens.com
Internet	http://www.aut.sea.siemens.com/winac/ http://www.aut.sea.siemens.com/simatic/support/index.htm http://www.ad.siemens.de/support/html_76/index.shtml
Europe	
Telephone	++49 (0) 911 895 7000
E-Mail	simatic.support@nbgm.siemens.de
Internet	http://www.ad.siemens.de/simatic-cs
Fax	++49 (0) 911 895 7001

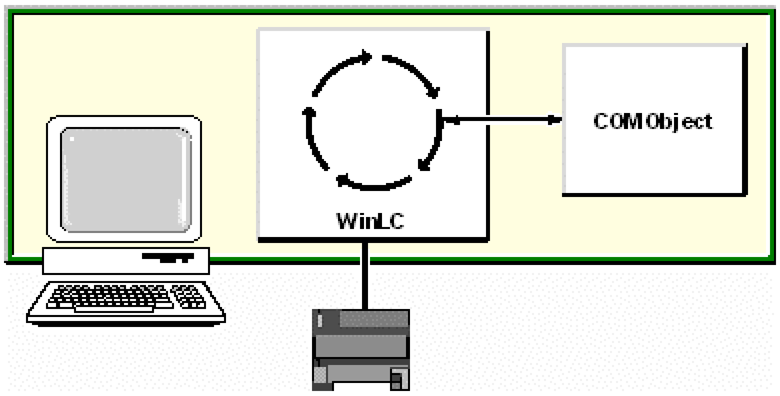
Product Overview

The Windows Automation Center (WinAC) Basis Open Development Kit (ODK) is an open interface to the Windows Logic controller (WinLC), providing a set of tools that enable you to implement a COM custom interface. This in-process COM object is implemented as a DLL. WinAC Basis ODK does not provide a dual interface: it does not directly support Visual Basic.

Special STEP 7 SFBs implement custom logic into the program logic that is being executed by the WinLC logic. The COM object that you create with the WinAC Basis ODK tools is an in-process COM object that executes within the WinLC process.

You can create more than one COM object, and each COM object can have more than one command.

Your COM object is invoked as an SFB, executed as part of the WinLC scan cycle. WinAC Basis ODK publishes the COM interface that it expects (IWinLCLogicExtension), and you implement your custom logic in an object with this interface. If the custom logic in the COM object requires too much time to be executed within the WinLC scan cycle, it must be executed on a separate thread of execution. WinAC Basis ODK includes a Visual C++ 6.0 application wizard which helps you set up this thread. Your COM object can also handle events by scheduling an asynchronous OB in the WinLC logic.



WinAC Basis ODK Expands the Capabilities of WinLC

Some of the benefits of using WinAC Basis ODK to expand the capabilities of WinLC include the following situations:

- Special control logic that was written in C or C++ needs to be incorporated into the S7 PLC.
- A complex (or proprietary) calculation (such as PID or gas flow) either has higher performance requirements or is more easily written and maintained in C or C++.
- The control solution requires connectivity with another application or hardware, such as for motion control or vision systems.
- The control solution needs to access features of the PC or the Windows NT operating system which are not accessible by standard S7 CPU mechanisms.

Tools Provided by WinAC Basis ODK

WinAC Basis ODK provides a set of software tools for developing custom logic that implements a COM "custom interface" with an interface published in the tool kit.:

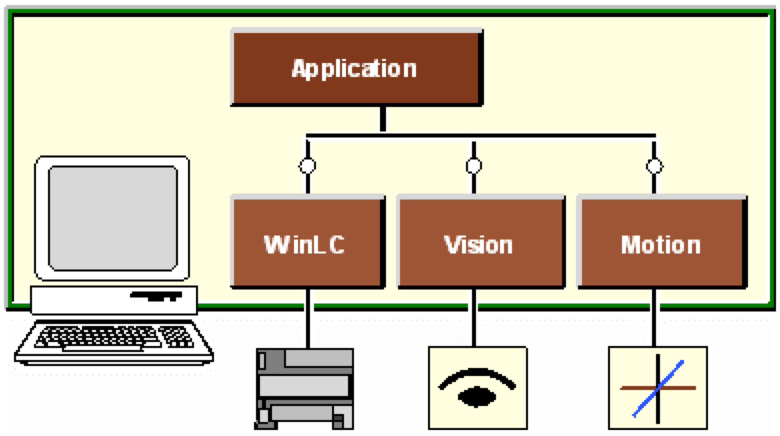
- Application wizard for generating a program shell with the correct interfaces and functions for interacting with WinLC
- STEP 7 library containing two SFBs that you can insert into the STEP 7 program:
  - ❑ SFB65001 ("CREA_COM") creates an instance of the COM object(s). You can have more than one object, with multiple actions in each object. SFB65001 returns the program handle for the COM object. This program handle can be stored in the WinLC memory to be used by SFB65002.
  - ❑ SFB65002 ("EXEC_COM") executes a specific function in the COM object created by SFB65001.
- COM extension for WinLC that enables the COM objects created with WinAC Basis ODK to communicate with WinLC
- Documentation: an electronic user manual
- Sample program: includes both a C++ project and a STEP 7 project

With WinAC Basis ODK, you can define and implement one or more standard FBs or FCs that become the block library used throughout the rest of your STEP 7 program. The blocks in this user-defined library, in turn, access the WinAC Basis ODK SFBs.

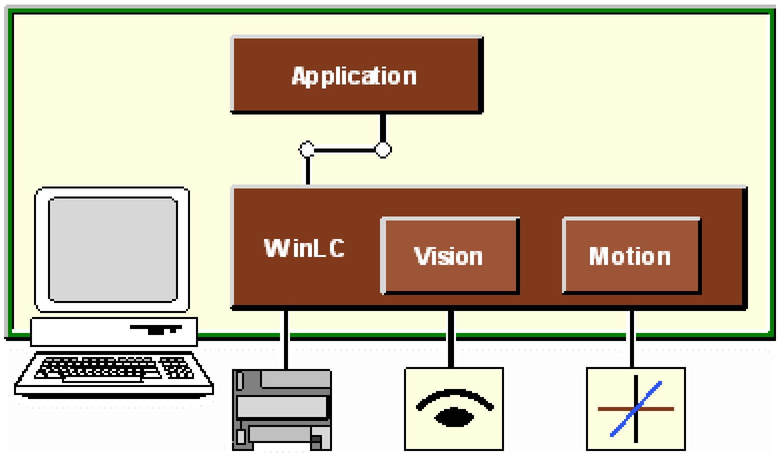
WinAC Basis ODK Provides a Mechanism for Defining Custom Logic

WinAC Basis ODK provides the mechanism for you to define one or more custom system functions that can be integrated into the STEP 7 program logic. Microsoft's COM technology connects the SFBs in the program logic to your custom logic extension.

For example: Without using WinAC Basis ODK, a process that uses a motion control board and a vision board might have a custom application that interacts between the two boards. This application interacts with WinLC by manipulating the I/O controlled by WinLC through asynchronous read/write functions provided by STEP 7 or by the Data control of WinAC Computing.



By using WinAC Basis ODK, this custom application could now interact directly with the WinLC program logic.



The WinAC Basis ODK software supports only in-process COM objects. An in-process COM object runs in the same process as WinLC and is loaded in the same address space. This allows for faster processing between the application and WinLC because there is no context switching between the applications. (If the application runs in a different process, the operating system must stop the current process and save its context before switching the execution focus to another process. Also, the function input/output data must be marshaled and moved between the processes.)

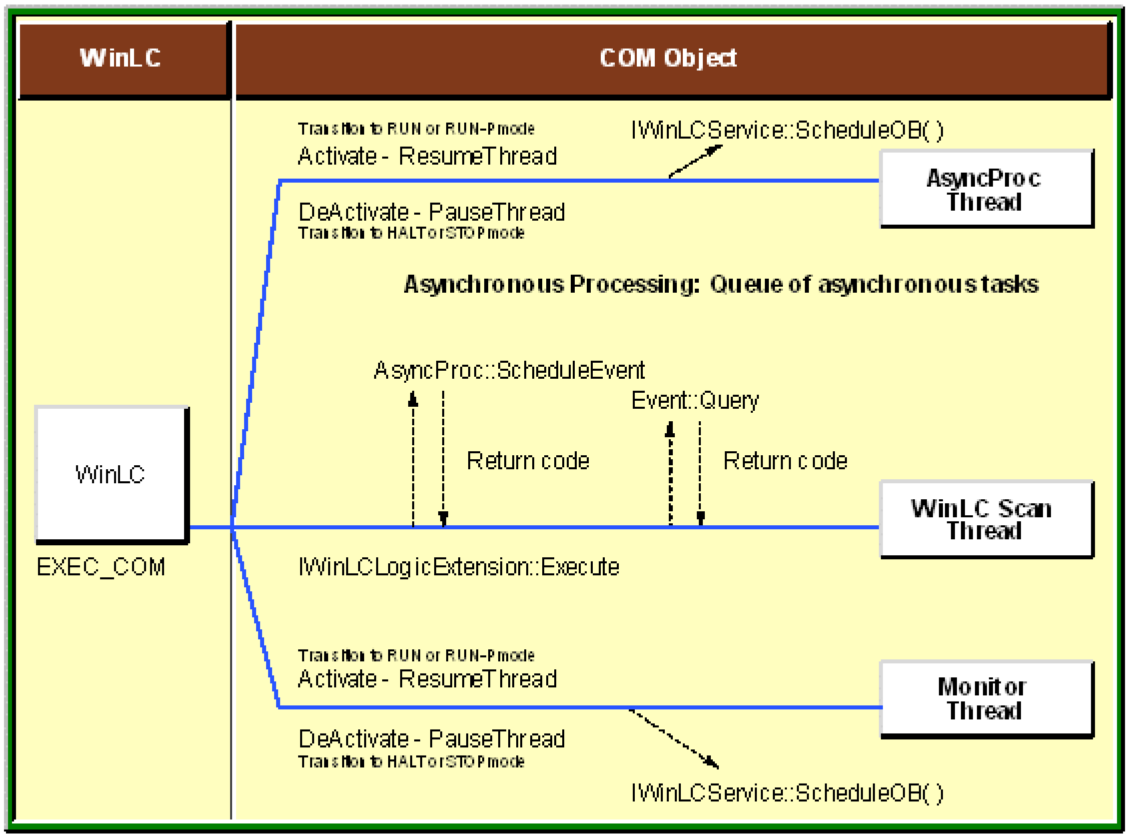
Note

Because WinLC may be started as a service, you must be aware that there are some restrictions on what your custom COM object can do. For example, if your COM object activates a graphical user interface (GUI), that interface would not be visible. The interface (GUI) needs to be a separate process started in a user context with communication to your COM object, possibly through shared memory.

WinAC Basis ODK handles the synchronization between the calls to the interface. However, if your COM object contains other threads or handles other asynchronous events, then your application must be responsible for coordinating internally between these events and the functions in the interface.

Note

Because the COM object executes as part of the WinLC scan cycle, it must finish within the scan requirements. If processing the COM object requires too much of the scan cycle, use the asynchronous processor (AsyncProc) to handle this processing outside of the WinLC scan.


WinLC
COM Object
Transition to RUN or RUN-P mode
Activate - ResumeThread
IWinLCService::ScheduleOB( )
AsyncProc Thread
DeActivate - PauseThread
Transition to HALT or STOP mode
Asynchronous Processing: Queue of asynchronous tasks
AsyncProc::ScheduleEvent
Event::Query
Return code
Return code
WinLC
WinLC Scan Thread
EXEC_COM
IWinLCLogicExtension::Execute
Transition to RUN or RUN-P mode
Activate - ResumeThread
Monitor Thread
DeActivate - PauseThread
Transition to HALT or STOP mode
IWinLCService::ScheduleOB( )

WinAC Basis ODK provides the tools for performing the following tasks:

- Initializing (or activating) your COM object which implements IWinLCLogicExtension at some user-defined time, such as startup (by calling SFB65001 in OB100)
- Releasing the the COM object
- Calling the methods of the COM object
- Notifying (or activating) when the COM object is created and before WinLC makes the transition from STOP to STARTUP or from HALT to RUN or RUN-P mode
- Notifying (or deactivating) your COM object when WinLC goes to STOP or HALT mode
- Notifying WinLC of an event. You can create an event to cause one of the following OBs to be executed:
  - ❑ OB4x (Hardware interrupts)
  - ❑ OB80 (Time error, such as a watchdog alarm)
  - ❑ OB82 (Diagnostic alarm interrupt)
  - ❑ OB83 (Insert/Remove Module interrupt)
  - ❑ OB87 (Communication Error interrupt)

System Requirements

Hardware Requirements

To run WinLC and WinAC Basis ODK, it is recommended that your computer meet the following criteria:

- A personal computer (PC) with the following:
  - ❑ Pentium processor running at 166 MHz or faster
  - ❑ 64 Mbytes RAM (recommended)
  - ❑ Microsoft Windows NT version 4.0 (or higher), with service pack 3 (or higher) required
- VGA color monitor, keyboard, and mouse or other pointing device (optional) which are supported by Microsoft Windows NT
- Approximately 10 Mbytes of additional memory on your hard disk
- At least 1 Mbyte free memory capacity on drive C for the Setup program (Setup files are deleted when the installation is complete)

WinLC has been tested and operated successfully on PCs as slow as a 486 processor running at 66 MHz with 24 Mbytes of RAM. WinLC has also been successfully tested on high-end PCs with dual Pentium processors.

Software Requirements

WinAC Basis ODK requires that the following software packages be installed on your computer:

- WinLC version 3.0 (or greater)
- STEP 7 version 5, service pack 3 (or greater)
- Microsoft Visual Developers Studio version 6 (Visual C++), service pack 3 (or greater)

WinAC Basis ODK will not run under Microsoft Windows 3.11, Windows 95, Windows 98, or Windows for Workgroups.

Installing WinAC Basis ODK

Before installing WinAC Basis ODK, ensure that your computer meets the recommended system requirements.

Caution

Do not install any component of WinAC (such as WinAC Basis ODK) on a computer while any other component of WinAC (such as WinLC, the Computing SoftContainer, programs that use the SIMATIC controls provided by Computing, or the panel for the CPU 416-2 DP ISA) are being executed (are currently running) on that computer.

Since Computing, WinLC, and other elements of WinAC use common files, attempting to install any component of the WinAC software when any of the components of WinAC are being executed by the computer can corrupt the software files. Always assure that the following programs are not running when you install WinLC:

- WinLC
- Panel for CPU 416-2 DP ISA
- WinAC Computing SoftContainer
- TagFile configurator
- Tool Manager
- WinAC Computing Configuration
- Any program (such as a program created in Visual Basic) that uses one of the SIMATIC controls provided by Computing

The Setup program for WinAC Basis ODK guides you step-by-step through the installation process. You can switch to the next step or to the previous step from any position. Use the following procedure to start the Setup program:

1. Start the dialog box for installing software under Windows NT by double-clicking on the Add/Remove Programs icon in the Control Panel.
2. Click on "Install..."
3. Insert the CD and click on the "Next" button. Windows NT searches automatically for the installation program SETUP.EXE.
4. Follow the instructions displayed by the Setup program.

If a Version of WinAC Basis ODK Is Already Installed...

If the Setup program finds another version of WinAC Basis ODK on the PC or programming device, it displays a list of all components, with the previously installed components deselected. You can select any of these components if you want to re-install them.

Your software is better organized if you uninstall any older versions before installing the new version. Overwriting an old version with a new version has the disadvantage that if you then uninstall, any remaining components of the old version are not removed.

Uninstalling (Removing) WinAC Basis ODK

Use the Windows NT "Add/Remove Programs" procedure to remove the WinAC Basis ODK software:

1. Select the **Start > Settings > Control Panel** menu command to display the Windows NT control panel.
2. Double-click on the Add/Remove Programs icon to display the Add/Remove Programs Properties dialog box.
3. Select the entry for "SIMATIC WinAC Basis ODK" and click on the "Add/Remove" button.
4. Follow the instructions of the dialogs to remove the WinAC Basis ODK software. (If the Remove Enable File dialog box appears, click on the "No" button if you are unsure how to respond.)

Getting Started with a Sample Program

WinAC Basis ODK installs a sample program on your computer during the installation procedure. This sample program ("Histogram") contains the STEP 7 project and COM object (C++ program) for collecting data about the scan time "jitter" of WinLC. The data collected by this custom application can be displayed as a histogram.

By working with this sample program, you can gain an understanding about how to create your own custom COM object that interacts with WinLC.

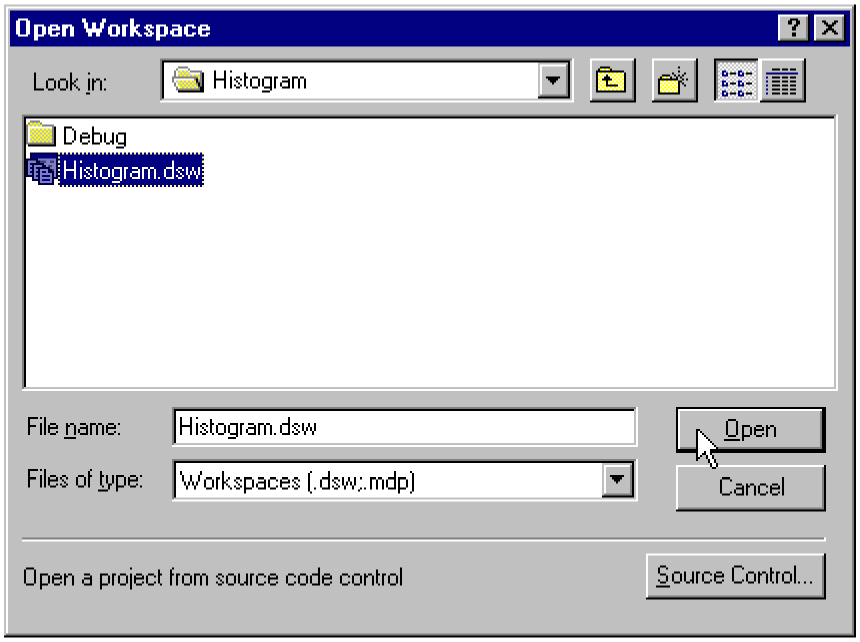
Basic Tasks for Using the Sample Program

In order to work with the sample program, you must perform the following tasks:

- Create the COM Object.

  1. Open Visual C++.
  2. Select the **File > Open Workspace** menu command and browse to the Histogram program in the following directory:

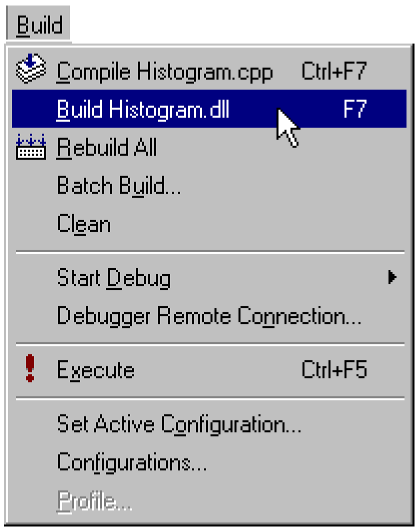
     Siemens\WinAC\Examples\Histogram



File name: Histogram.dsw

Files of type: Workspaces (.dsw;.mdp)

3. Select the Histogram.dsw file and click on Open.
4. Select the **Build > Histogram DLL** menu command to make the project.



- Run the Histogram program.
  1. Open both WinLC and the SIMATIC Manager.
  2. Open (load) the WinAC Basis ODK library.
  3. Open the Histogram project and download the Histogram program to WinLC.
  4. Change the operating mode of WinLC from STOP mode to RUN or RUN-P mode.
  5. Use the variable table (in the STEP 7 project) to view the updated data.

Sample Program ("Histogram")

WinAC Basis ODK provides a sample program ("Histogram") that exercises WinLC and builds a histogram of the execution times. This sample program consists of the following elements:

- STEP 7 project "Histogram" which includes:
  - STEP 7 library "Histogram"
  - STEP 7 program "Histogram"
- Variable table (VAT)
- COM object "Histogram"

STEP 7 Project (Histogram)

The STEP 7 program stores data about the scan cycle (current execution time, last execution time, maximum execution time, minimum execution time, mode or most frequent scan time, and deviation) and calls the COM object.

The Histogram program consists of two parts:

- Histogram library for STEP 7. This library uses WinAC Basis ODK to implement the functions. It encapsulates the calls to the ODK COM object and presents a meaningful view of the histogram functions.
- STEP 7 program ("Histogram"). This small program uses the Histogram library.

When the operating mode of WinLC changes from STOP to RUN, WinLC executes OB100 (warm restart).

- OB100 calls FB1 ("CREA_HISTOGRAM").
    - FB1 calls SFB65001 to create the COM object for the histogram program
    - FB1 verifies that the COM object was created.
- OB100 calls FB2 ("INIT_HISTOGRAM").
    - FB2 calls SFB65002 with the command "Init Command" to initialize the histogram program (by setting the initial values to 0).

After OB100 finishes, WinLC executes OB1 (main program cycle).

- OB1 calls FB3 (“UPDA_HISTOGRAM”).
- FB3 calls SFC64 ("TIME_TCK") and writes the current clock time to the input data.
- FB3 sets a deviation factor for the scan cycle. (Encountering a deviation from the mode greater than this value will be used to schedule OB80.)
- FB3 creates the command for updating the data in the COM object.
- FB3 calls SFB65002 ("EXEC_COM").

Sample COM Object ("Histogram")

The COM object "Histogram" is a C++ program that reads the data about the scan time and creates a histogram of the "jitter" in the WinLC scan time.

The sample program for the Histogram example includes the following key files:

- Histogram.h
- Histogram.ccp

These files show how to implement the histogram functionality. The application wizard (WinAC ODK AppWizard) generated the framework for this object. These two files contain the implementation.

This sample shows a single ODK COM object which can execute two functions:

- Initialize the histogram
- Update the histogram

This sample program also shows how to schedule an asynchronous error OB in WinLC. This code is included as an example of how to schedule an OB. For most applications, special cases in the Execute function can be handled with a return code, while the scheduling of an OB would be more commonly done when an asynchronous event occurs. The scheduling of the error OB is included here as an example of how the interface can be used.

Histogram.h

```
// Histogram.h : Declaration of the CHistogram
#ifndef __WINLCLOGICEXTENSION_H_
#define __WINLCLOGICEXTENSION_H_

#include "resource.h" // main symbols
#include "HistogramIDL.h" // Generated from IDL file
///////////////////////////////////////////////////////////////////
// CWinLCLogicExtension
class CWinLCReadData;
class CWinLCReadWriteData;

class ATL_NO_VTABLE CHistogram :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CHistogram, &CLSID_Histogram>,
public IDispatchImpl<IWinLCLogicExtension, &IID_IWinLCLogicExtension,
&LIBID_WinACOdkLib>
{
  // Indexes into the output memory area
  enum OutputIndexes
  {
  idxErrMsec = 0 * sizeof(UINT32),
  idxCurMsec = 1 * sizeof(UINT32),
  idxLastMsec = 2 * sizeof(UINT32),
  idxMinMsec = 3 * sizeof(UINT32),
  idxMaxMsec = 4 * sizeof(UINT32),
  idxModeMsec = 5 * sizeof(UINT32),
  idxHistogram = 6 * sizeof(UINT32),
  };
  ///////////////////////////////////////////////////////////////////
  // The data buffer contains these fields:
  //
  // + offset 0 : Maximum acceptable deviation from scan time.
  // |
  // + offset 4 : Time of the current scan cycle
  // |
  // + offset 8 : Time at the last scan cycle
  // |
  // + offset 12 : Minimum cycle time
  // |
  // + offset 16 : Maximum cycle time
  // |
  // + offset 20 : Mode time = histogram peak time
  // |
  // + offset 24 : Start of histogram
  // *
  // *
  // *
  // + offset nOutput-1 : (End of outData)
  //
  ///////////////////////////////////////////////////////////////
public:
DECLARE_REGISTRY_RESOURCEID(IDR_WINLCLOGICEXTENSION)
DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CHistogram)
COM_INTERFACE_ENTRY(IWinLCLogicExtension)
END_COM_MAP()
// Public Class Members
public:
CHistogram()
  : WinLCSvc(0)
{
}
// Protected Class Members
protected:
void ScheduleOB80 (unsigned short mode, unsigned short deviation) const;
HRESULT ExecuteUpdate(CWinLCReadData & Input, CWinLCReadWriteData & Output);
HRESULT ExecuteInit(CWinLCReadWriteData & Output);

IWinLCServices* WinLCSvc;
// IWinLCLogicExtension methods
public:
STDMETHOD(Execute)(unsigned long command, long nInput, byte inData [], long
nOutput, long* nUsedOutput, byte outData []);
STDMETHOD(DeActivate)();
STDMETHOD(Activate)(IWinLCServices * IWinLCSvc);
};
#endif //__WINLCLOGICEXTENSION_H_
```

Histogram.ccp

```
///////////////////////////////////////////////////////////////////
// Histogram.cpp : Implementation of CHistogram
//

#include "stdafx.h"
#include "Histogram.h"
#include "HistogramIDL.h"
#include "WinLCReadData.h"
#include "WinLCReadWriteData.h"

////////////////////////////////////////////////////////////////////
// CHistogram

//*****************************************************************
// Activate is called by WinLC when WinLC goes from stop to Run
//*****************************************************************
STDMETHODIMP CHistogram::Activate(IWinLCServices *IWinLCSvc)
{
  // Set the local IWinLCServices pointer
  WinLCSvc = IWinLCSvc;

  // TODO: Add your implementation code here
```

```
    return S_OK;
  }

  //*****************************************************************
  // DeActivate is called by WinLC when WinLC goes from Run to Stop
  //*****************************************************************
  STDMETHODIMP CHistogram::DeActivate()
  {
    // Set the IWinLCServices pointer to NULL
    WinLCSvc = NULL;

    // TODO: Add your implementation code here
    return S_OK;
  }

  //*****************************************************************
  // Execute is called by WinLC when SFB65002 is encountered
  //*****************************************************************
  STDMETHODIMP CHistogram::Execute(unsigned long command,
    long nInput,
    byte inData [],
    long nOutput,
    long *nUsedOutput,
    byte outData [])
  {
    // Create The Helper Data Access Classes
    // - Call Methods on these classes to read and write data
    CWinLCReadData Input(nInput, inData);
    CWinLCReadWriteData Output(nOutput, outData);
  // Just return if there is not enough output space for the structure
    if (nOutput <= idxHistogram)
    return E_FAIL;

    // Execute the command
    switch(command)
    {
    case 'INIT':
    ExecuteInit(Output);
    break;
    case 'UPDA':
    ExecuteUpdate(Input, Output);
    break;
    default:
    return E_FAIL;
    } // switch
    *nUsedOutput = Output.LastByteChanged() + 1;
    return S_OK;
  } // Execute

  ////////////////////////////////////////////////////////////////////
  // Initialize the histogram data area.
  HRESULT CHistogram::ExecuteInit(CWinLCReadWriteData & Output)
  {
    // reset the histogram data.
    for (int n = 0; n < Output.GetBufferSize(); n++)
    {
    Output.WriteS7BYTE(n, 0);
    }
    // initialize MinMsecs with a giant value.
    Output.WriteS7DWORD(idxMinMsec, 0xffffffff);
    return S_OK;
  } // ExecuteINIT

  /////////////////////////////////////////////////////////////////////
  // Update the histogram data area.
  HRESULT CHistogram::ExecuteUpdate(CWinLCReadData & Input, CWinLCReadWriteData &
  Output)
  {
    // Get the max word index
    long maxUINT32idx = Output.GetBufferSize() / sizeof(UINT32);

    // Get copy of the input data
    UINT32 curMSecs;
    Input.ReadS7DWORD(idxCurMsec, curMSecs);

    // Compute difference and update histogram if not the first time
    UINT32 lastMSecs;
    Output.ReadS7DWORD(idxLastMsec, lastMSecs);

    if (lastMSecs != 0)
    {
    // Compute the mSecs for last scan
    lastMSecs = curMSecs - lastMSecs;
  // Update the min/max value
    UINT minMsecs, maxMsecs;
    Output.ReadS7DWORD(idxMinMsec, minMsecs);
    Output.ReadS7DWORD(idxMaxMsec, maxMsecs);

    if (lastMSecs < minMsecs)
    Output.WriteS7DWORD(idxMinMsec, lastMSecs);
    if (lastMSecs > maxMsecs)
    Output.WriteS7DWORD(idxMaxMsec, lastMSecs);

    // clamp input to not exceed the array size
    long histoSlot = idxHistogram + (lastMSecs * sizeof(UINT32));
    if (histoSlot > maxUINT32idx)
    histoSlot = maxUINT32idx;

    // Update the histogram
    UINT32 newCount;
    Output.ReadS7DWORD(histoSlot, newCount);
    Output.WriteS7DWORD(histoSlot, ++newCount);

    // update the mode
    UINT32 modeMSecs, oldCount;
    Output.ReadS7DWORD(idxModeMsec, modeMSecs);
    histoSlot = idxHistogram + (modeMSecs * sizeof(UINT32));
    if (histoSlot > maxUINT32idx)
    histoSlot = maxUINT32idx;
    Output.ReadS7DWORD(histoSlot, oldCount);
    if (newCount > oldCount)
    {
    modeMSecs = lastMSecs;
    Output.WriteS7DWORD(idxModeMsec, modeMSecs);
    }
    // schedule an error OB if the deviation is too much
    if (lastMSecs > modeMSecs)
    {
    UINT32 errDeviation;
    Input.ReadS7DWORD(idxErrMsec, errDeviation);
    if ((lastMSecs - modeMSecs) > errDeviation)
    {
    ScheduleOB80 (modeMSecs, lastMSecs - modeMSecs);
    }
    }
    } // if
    Output.WriteS7DWORD(idxLastMsec, curMSecs);
    return S_OK;
  } // ExecuteUPDA
  ///////////////////////////////////////////////////////////////////
  // ScheduleOB80
  void
  CHistogram::ScheduleOB80(unsigned short mode, unsigned short deviation) const
  {
    WinLCSvc->ScheduleOB(
    0x35, // execute asynchonous error OB
    0x01, // use cycle time error event number
    0xFE, // fill in configured sequence layer
    80, // execute OB80
    0xC4, // dataType2 (for long word) - two 16 bit words
    0x58, // dataType1 (for short word) - time in milliseconds
    deviation, // data1
    mode); // data2 (half is 0)
  }
```

Basic Tasks for Implementing a Custom COM Interface

1. Create the custom COM object that implements the IWinLCLogicExtension interface:
   - Use the application wizard for Visual C++ version 6 (WinAC ODK AppWizard) to create the program shell for the COM object.
   - Implement the IWinLCLogicExtension Execute function.
   - Implement the IWinLCLogicExtension Activate function. This can be left as an empty function.
   - Implement the IWinLCLogicExtension DeActivate function. This can be left as an empty function.
2. Create your STEP 7 program that interacts with the COM object:
   - Load the STEP 7 library that contains the SFBs supplied by WinAC Basis ODK.
   - Insert these SFBs into your STEP 7 program.
3. Debug your COM object with the STEP 7 program running on WinLC.

Using the Application Wizard to Create Your COM Object

The application wizard guides you through the following tasks:

- Creating the ATL/COM project that implements the IWinLCLogicExtension interface required by WinAC Basis ODK
- Creating a C++ class that you can use to execute functions asynchronously from the WinLC scan cycle (optional)
- Creating a C++ class that you can use to monitor one or more attributes of your system (optional)

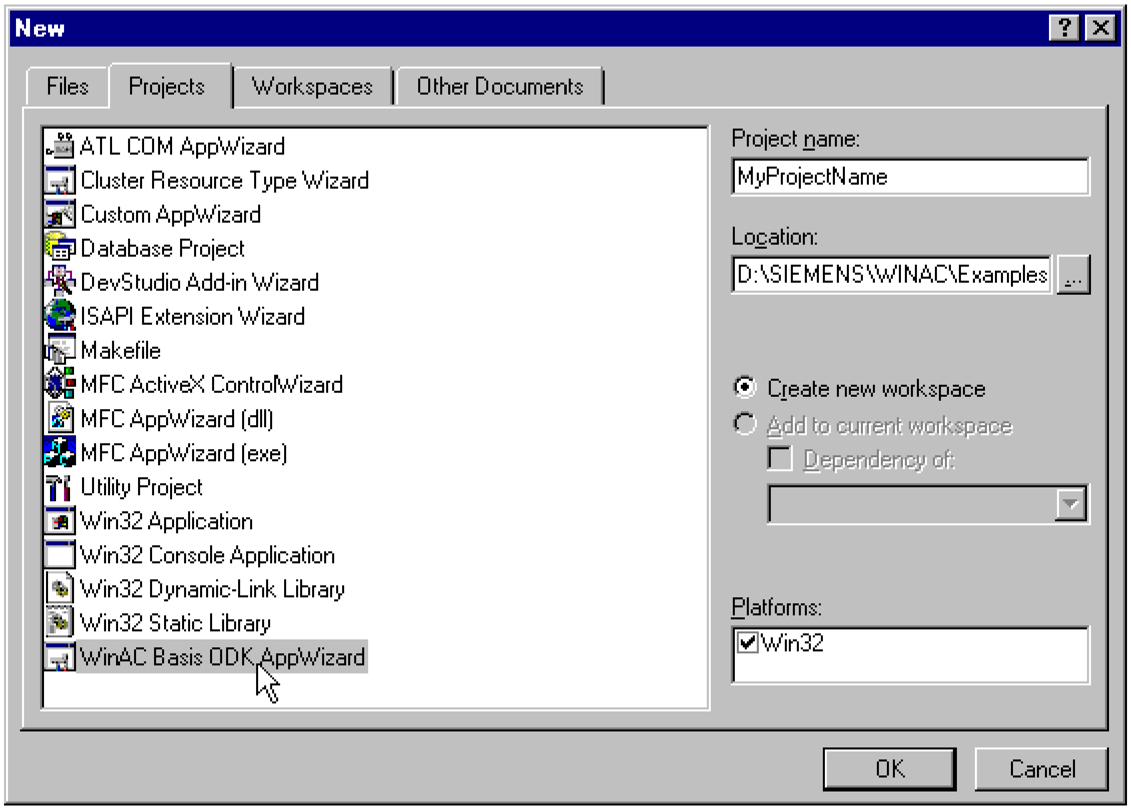
The application wizard provides a summary of the options that you selected for the type of COM object or DLL that you plan to develop. After you confirm these choices, the application wizard generates the C/C++ program shell.

Note

The WinAC Basis ODK application wizard can only be used in the Microsoft Visual C++ 6.0.

Start the WinAC ODK Application Wizard

1. Start Microsoft Visual C++ V6.
2. Open a new project: select the **File > New** menu command.
3. Click on the Projects tab and select the WinAC ODK AppWizard icon.
4. Enter the project name and location for the project and click on OK.



Project name: MyProjectName

Location: D:\SIEMENS\WINAC\Examples

Platforms: ☑ Win32

Visual C++ starts the WinAC Basis ODK application wizard. Use the wizard to create all of the elements for the program shell.

Enter the ATL/COM Class and Interface Information

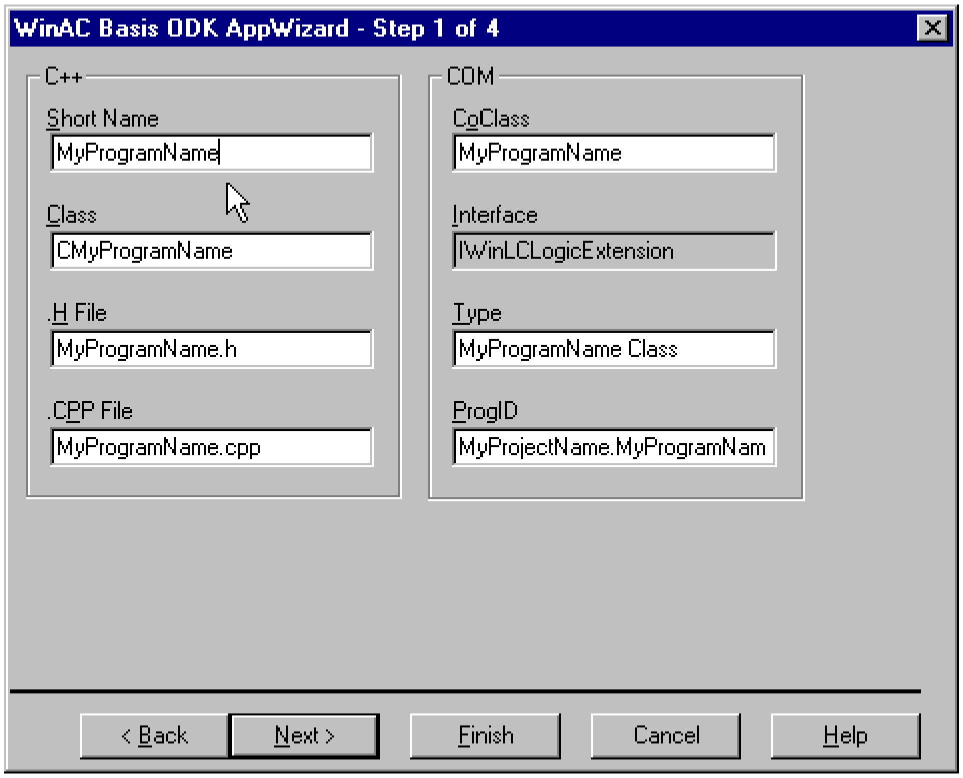
1. Using the WinAC Basis ODK application wizard, enter the ATL/COM class and interface information by entering the Short Name for the application. Notice that all of the fields for the application wizard are created automatically, using the Short Name as a default. You can also modify these other fields individually without changing any of the other fields.

   Note:

   The ProgID field is the required parameter in SFB65001 ("CREA_COM").

   Also, you cannot change the interface name (IWinLCLogicExtension). This interface is required by SFB65002 ("EXEC_COM") and must be present in all WinAC Basis ODK COM objects.

2. After the correct information has been entered for all of the fields, click on Next.



C++

Short Name: MyProgramName

Class: CMyProgramName

.H File: MyProgramName.h

.CPP File: MyProgramName.cpp

COM

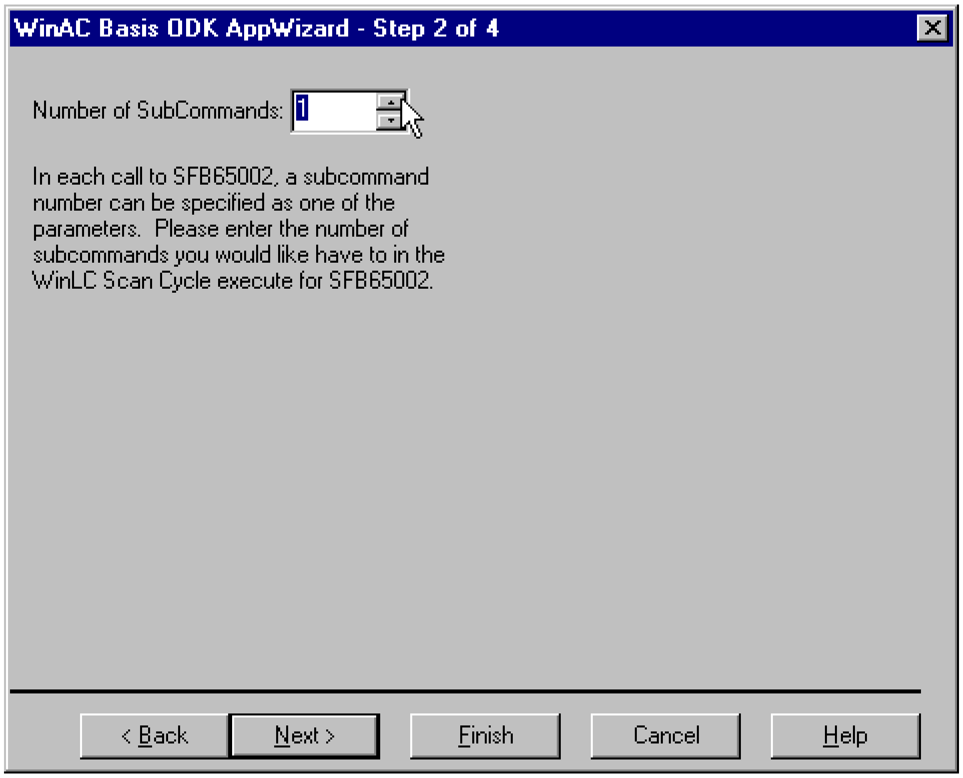
CoClass: MyProgramName

Interface: IWinLCLogicExtension

Type: MyProgramName Class

ProgID: MyProjectName.MyProgramNam

3. Enter the number of subcommands to be executed for SFB65002 ("EXEC_COM") and click on Next.
4. A subcommand is a unique function or task within the SFB. Using subcommands allows you to create one COM object that performs a variety of tasks, rather than several COM objects that perform single tasks. The application wizard allows you to create up to 100 subcommands.

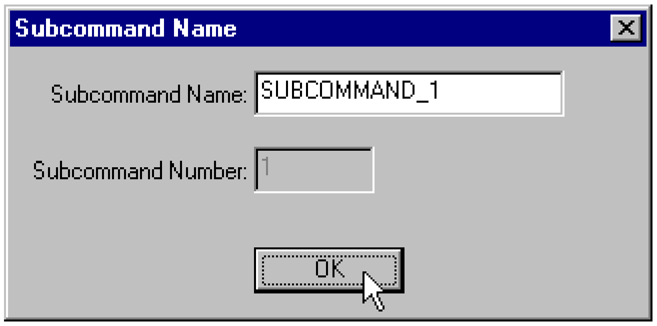


WinAC Basis ODK AppWizard - Step 2 of 4

Number of SubCommands: 1

In each call to SFB65002, a subcommand number can be specified as one of the parameters. Please enter the number of subcommands you would like have to in the WinLC Scan Cycle execute for SFB65002.

5. The application wizard displays a dialog box for each of the subcommands to be configured. Enter the name for the subcommand and click on OK.



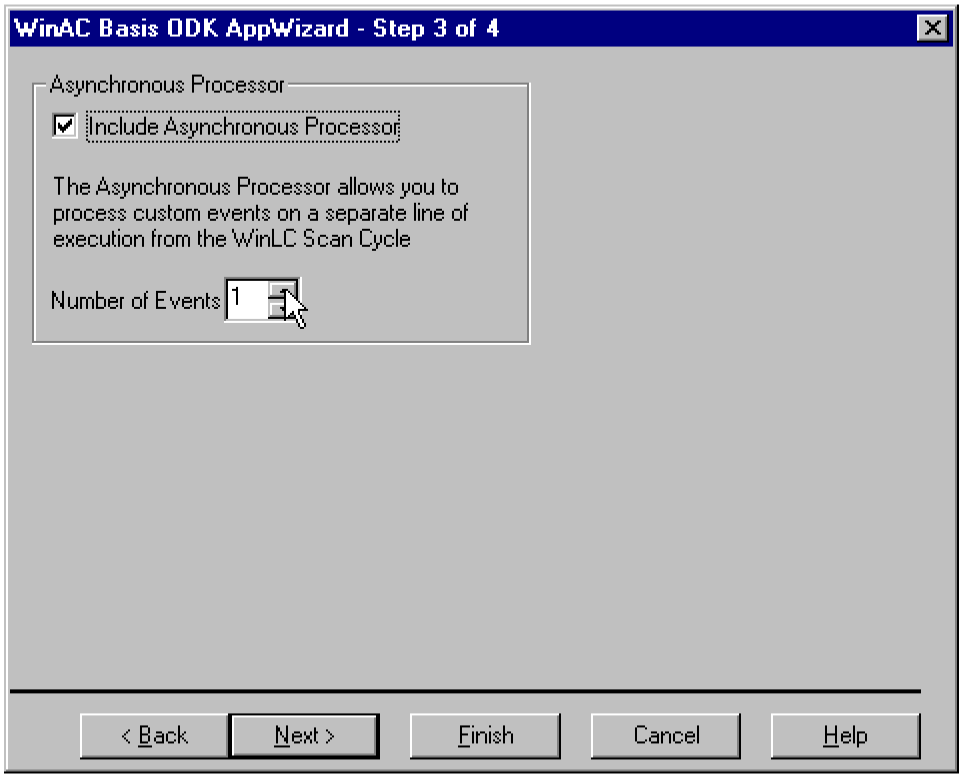
Subcommand Name

Subcommand Name: SUBCOMMAND_1

Subcommand Number: 1

Select Whether to Enable Asynchronous Processing

1. Asynchronous Processing allows commands to be executed outside of the main IWinLCLogicExetension Execute() function. The asynchronous processor uses objects derived from the EventMsg class to carry out event specific functions.
2. To add asynchronous processing to the generated WinAC Basis ODK project, select the "Include Asynchronous Processor" check box.
3. Enter the number of events to create in the "Number of Events" field. The number of events that can be included in the project is limited to 100.
4. Click on Next.


WinAC Basis ODK AppWizard - Step 3 of 4

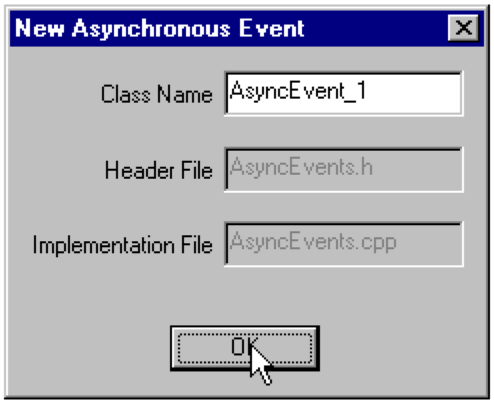
Asynchronous Processor

☑ Include Asynchronous Processor

The Asynchronous Processor allows you to process custom events on a separate line of execution from the WinLC Scan Cycle

Number of Events: 1

5. The application wizard displays a dialog box for each of the asynchronous event to be configured. Enter the class name for each event and click on OK.
6. The application wizard places the event classes in the following files: AsyncEvent.h and AsyncEvent.cpp.


New Asynchronous Event

Class Name: AsyncEvent_1

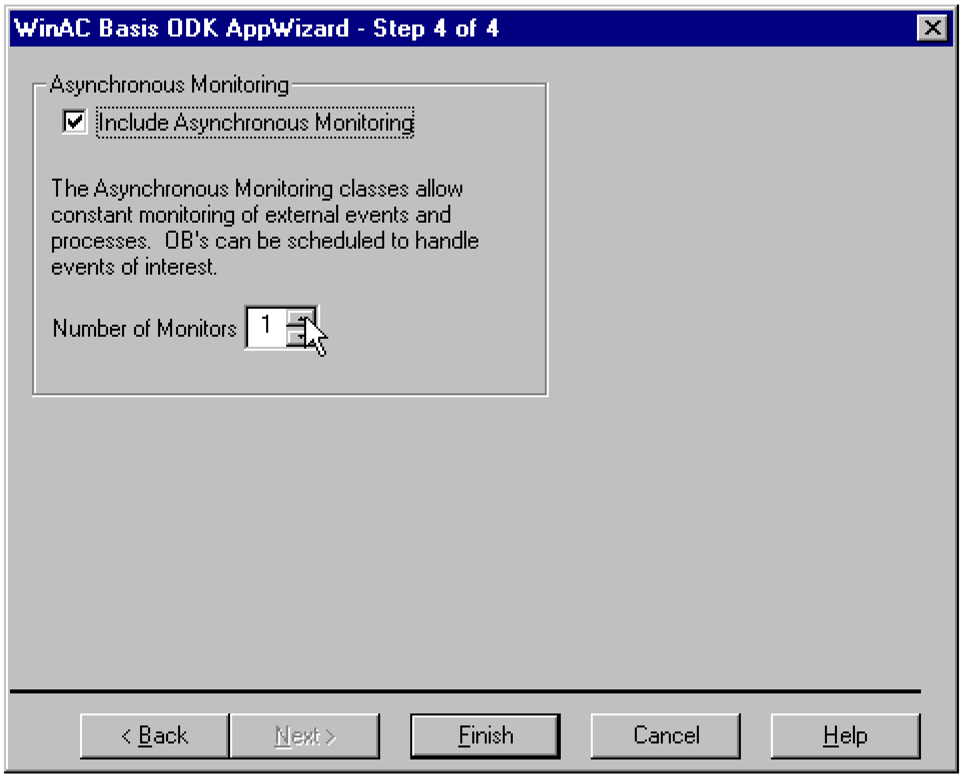
Header File: AsyncEvents.h

Implementation File: AsyncEvents.cpp

Select Whether to Enable Asynchronous Monitoring

After all of the event classes have been given a name, the application wizard displays the option to enable asynchronous monitoring. Use asynchronous monitoring if your COM object needs to implement some functionality in the background (for example, to monitor data or wait for an event to occur asynchronous from the scan cycle).

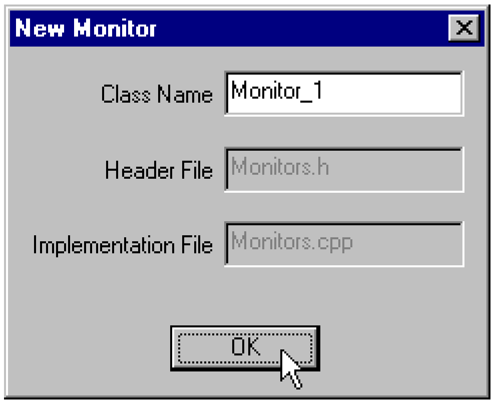
1. To include asynchronous monitoring in the WinAC Basis ODK project, select the check box for the Include Asynchronous Monitoring option.
2. In the Number of Monitors field, enter the number of monitoring classes and threads to be generated by the application wizard. Each generated class can perform custom functionality by overloading the Execute() function.


WinAC Basis ODK AppWizard - Step 4 of 4
Asynchronous Monitoring
Include Asynchronous Monitoring
< Back
Next >
Finish
Cancel
Help

The Asynchronous Monitoring classes allow constant monitoring of external events and processes. OB's can be scheduled to handle events of interest.

Number of Monitors: 1

3. Click on Finish. The application wizard displays a New Monitor dialog box that allows you to enter a name for each class name. (A dialog box is generated for each monitor class that you configured.)

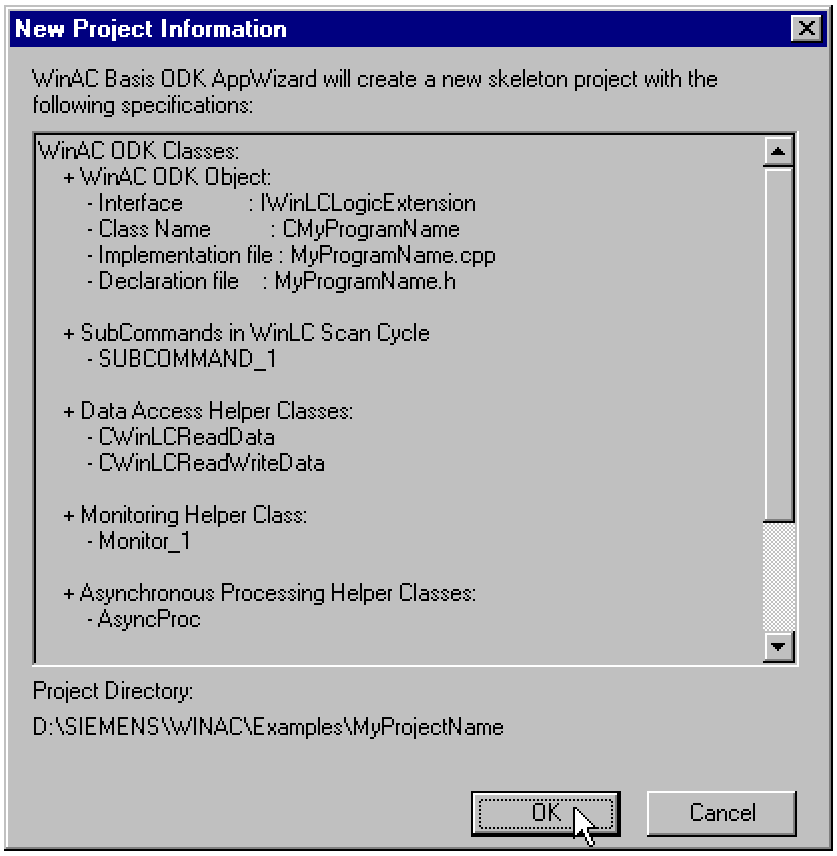


Class Name: Monitor_1
Header File: Monitors.h
Implementation File: Monitors.cpp

4. Enter the class name for the monitor and click on OK.

Generate the WinAC Basis ODK Object

1. Click on Finish and review the summary of the files to be created for the new project.
2. Click on OK to confirm the options and to generate the WinAC Basis ODK project.


New Project Information

WinAC Basis ODK AppWizard will create a new skeleton project with the following specifications:

```
WinAC ODK Classes:
  + WinAC ODK Object:
     - Interface           : IWinLCLogicExtension
     - Class Name          : CMyProgramName
     - Implementation file : MyProgramName.cpp
     - Declaration file    : MyProgramName.h

  + SubCommands in WinLC Scan Cycle
     - SUBCOMMAND_1

  + Data Access Helper Classes:
     - CWinLCReadData
     - CWinLCReadWriteData

  + Monitoring Helper Class:
     - Monitor_1

  + Asynchronous Processing Helper Classes:
     - AsyncProc
```

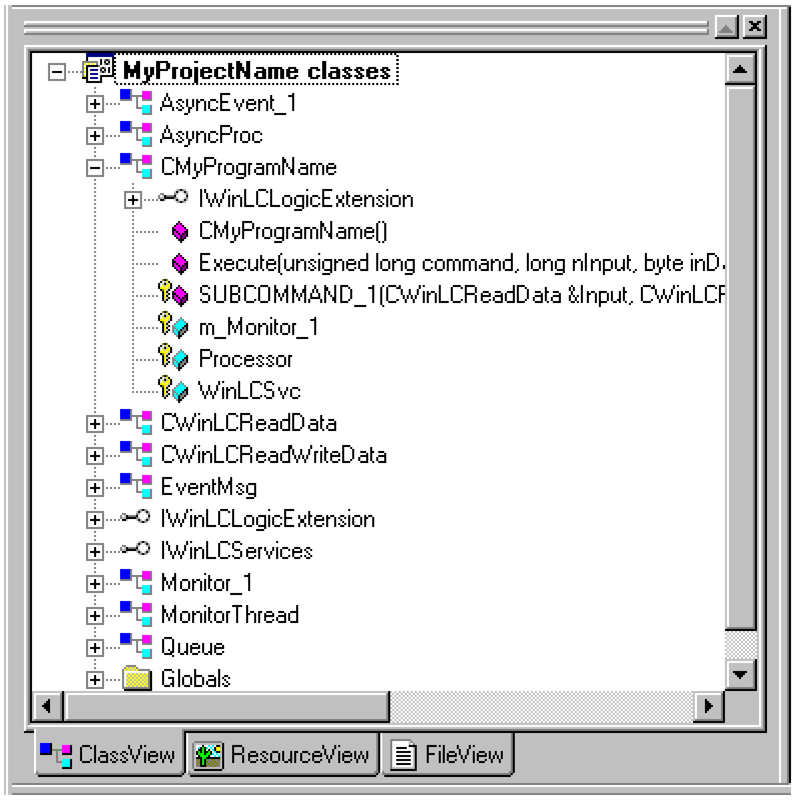
Project Directory:

D:\SIEMENS\WINAC\Examples\MyProjectName

Project Shell

The WinAC Basis ODK application wizard creates the program shell for the project.

Class View of the C++ Object Shell



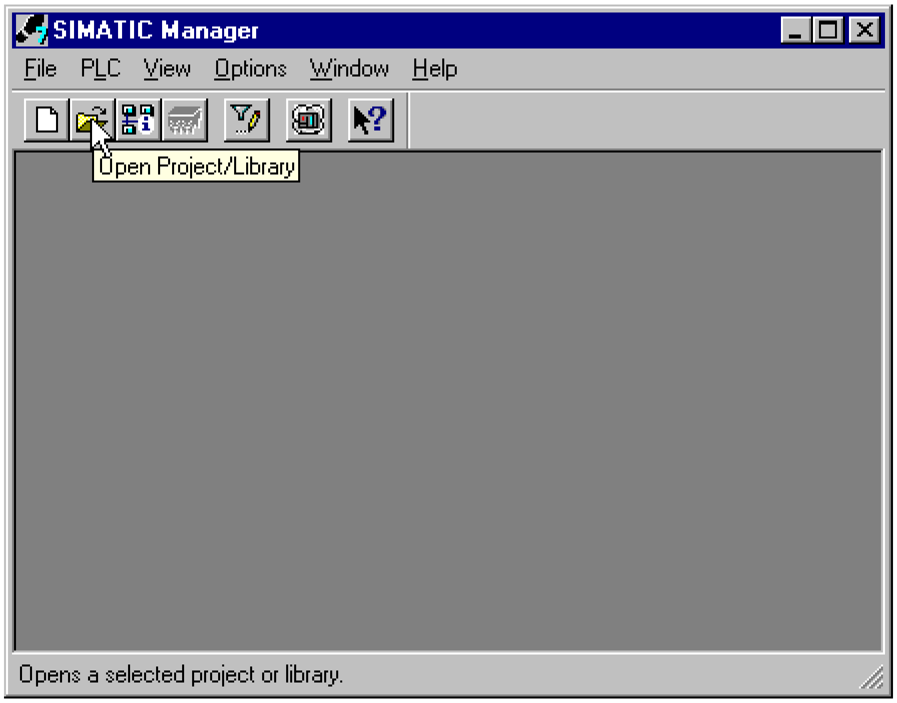
```
MyProjectName classes
    AsyncEvent_1
    AsyncProc
    CMyProgramName
        IWinLCLogicExtension
        CMyProgramName()
        Execute(unsigned long command, long nInput, byte inD
        SUBCOMMAND_1(CWinLCReadData &Input, CWinLCF
        m_Monitor_1
        Processor
        WinLCSvc
    CWinLCReadData
    CWinLCReadWriteData
    EventMsg
    IWinLCLogicExtension
    IWinLCServices
    Monitor_1
    MonitorThread
    Queue
    Globals
```

Creating the STEP 7 Program

Loading the WinAC Basis ODK Library into STEP 7

WinAC Basis ODK installs a custom STEP 7 library that contains the SFBs that allow the STEP 7 program to interact with your custom COM object. Before you can use these SFBs in your STEP 7 program, you must load the library:

1. Start the SIMATIC Manager of STEP 7 by selecting the **Start > SIMATIC > SIMATIC Manager** menu command.
2. Select the **New > Open Project/Library** menu command.


SIMATIC Manager
File PLC View Options Window Help
Open Project/Library
Opens a selected project or library.

3. If you have not previously loaded the WinAC Basis ODK library, click on Browse and select the directory where the WinAC Basis ODK library was installed (for example, the directory D:\SIEMENS\STEP7\S7libs).
4. Select the WinAC Basis ODK library and click on OK.
5. STEP 7 opens the WinAC Basis ODK library. This library contains the following elements:
   - SFB65001 ("CREA_COM")
   - SFB65002 ("EXEC_COM")

Inserting the WinAC Basis ODK SFBs into the STEP 7 Program

The two SFBs provided by WinAC Basis ODK allow you to use your custom COM object as part of the STEP 7 program being executed by WinLC:

- SFB65001 ("CREA_COM") creates the instance of your COM object (DLL).
- SFB65002 ("EXEC_COM") sends an execution command to the COM object created by SFB65001.

WinLC executes these SFBs like any other SFB. The "EXEC_COM" SFB (SFB65002) calls the execute function of your ODK COM object, and the scan time is extended by the time required to execute this function. If you need to execute a command that can take a long time (relative to your scan time requirements), use the asynchronous processor (AsyncProc) to execute the command. Because SFB65002 ("EXEC_COM") does not finish its execution until the COM object finishes, the time required to execute your custom application is added to the scan cycle.

In order to use these SFBs in your STEP 7 program, you must have loaded the WinAC Basis ODK library. You can then insert these SFBs into your program just like any other STEP 7 element:

1. Open the STEP 7 program by selecting the **Start > SIMATIC > SIMATIC Manager** menu command.
2. Select and drag the SFBs from the WinAC Basis ODK library to the program blocks of the STEP 7 program.
3. Edit your program to call the SFB65001 ("CREA_COM").

Note

Typically, the STEP 7 program calls SFB65001 ("CREA_COM") when the program starts in the start-up OB (such as OB100) to create the instance of the COM object. The program handle returned from this call is then saved to a memory location for further reference. You can also have the program call SFB65001 from other logic blocks, such as an FB or in OB1.

4. Edit your program to call SFB65002 (EXEC_COM).

WinAC Basis ODK COM Object

WinAC Basis ODK Application Wizard

The WinAC Basis ODK Application Wizard provides a skeleton Microsoft Visual C++ 6.0 project that, when compiled, produces a dynamic link library (DLL) that can be used with the WinAC Basis ODK update to WinLC. The WinAC Basis ODK application wizard features include the following:

- Data access helper class: The application wizard always generates the CWinLCReadData and CWinLCReadWriteData classes.
- Asynchronous processor class: The application wizard generates the AsyncProc, EventMsg, and Queue classes only when you select the Asynchronous Processing option.
- Monitor class: The application wizard generates the MonitorThread class only when you select the the Asynchronous Monitoring option.
- Skeleton implementation of the IWinLCLogicExtension interface.

The following table lists the support and helper classes generated by the application wizard.

Class	Description
Data access helper class	
CWinLCReadData	The read-only data access helper class serves as a wrapper to the input buffer passed into the IWinLCLogicExtension Execute() function. Functions are provided to access the data in the buffer as STEP 7 data types.
CWinLCReadWriteData	The read/write data access helper class serves as a wrapper to the output buffer passed into the IWinLCLogicExtension Execute() function. Functions are provided in this class to read and write STEP 7 data types to the buffer.
Asynchronous processor class	
AsyncProc	The asynchronous processor class processes events posted to it on a thread of execution separate from the main program.
EventMsg	This is the base class to Asynchronous Events: all asynchronous events posted to the asynchronous processor should be derived from this class. The Execute() function should be overloaded in the derived class to provide custom processing for the event.
Queue	This is a basic queue ("first in, first out") class. The asynchronous processor uses it for scheduling events to process.
Monitor class	
MonitorThread	This is the base class for asynchronous monitors: all classes for monitoring external events and processes should be derived from this class. The Execute() function should be overloaded to provide customized monitoring actions.

Execution Rules

IWinLCLogicExtension Execute Method

The Execute method is executed when SFB65002 ("EXEC_COM") is called in your STEP 7 program. It is the function that is called as part of your STEP 7 program and becomes part of the OB execution time.

IWinLCLogicExtension Activate and Deactivate Methods

The Activate call is made before the switch to STARTUP or RUN. Similarly, DeActivate is called after the switch. Activate is always called before a call to the Execute method, and DeAQctivare is always called after any call to the Execute method.

Note

The CPU is in HALT when it reaches a breakpoint in the STEP 7 editor.

The WinLCState returned from IWinLCServices::ReadState should always be START_OB100, START_OB101, START_OB102, or RUN when queried from Activate, DeActivate, or Execute functions.

Activate is called after the COM object is created and before the first call to Execute.

- Activate is called before:
  - WinLC transition from STOP mode to STARTUP mode
  - WinLC transition from HALT mode to RUN mode
- DeActivate is called after:
  - WinLC transition from RUN mode to STOP or HALT mode
  - WinLC transition from STARTUP mode to STOP or HALT mode

Note

WinLC releases your COM objects on a memory reset (MRES) or on CPU shutdown. Because both the MRES and CPU shutdown first change WinLC to STOP mode, DeActivate will always be called before the COM objects are released.

IWinLCServices Interface

When WinLC calls the Activate method in your COM object, you are given a pointer to IWinLCServices. This is a COM object implemented in WinLC that gives you access to WinLC features that are not otherwise available in the Execute function. The methods of the IWinLCServices interface can be called from IWinLCLogicExtension Execute, Activate, or DeActivate. They can also be called asynchronously from the asynchronous processor (AsyncProc) or monitor (MonitorThread) classes.

The IWinLCServices interface provides three methods:

- ReadState
- ScheduleOB
- ReadSysData (not implemented in the current version of WinAC Basis ODK)

HRESULT ReadState(WinLCState * state)

This function retrieves the current state (operating mode) of the WinLC controller. The value of *state* will be one of the following modes:

- STOP
- HALT
- STARTUP_100
- STARTUP_101
- STARTUP_102
- RUN

Return Value

- *HRESULT*: S_OK means that the call succeeded.

Parameters

- *WinLCState **: pointer to a variable of type WinLCState

HRESULT ScheduleOB(byte class_ID, byte eventNr, byte seqLayer, byte obNum, byte dataType2, byte dataType1, unsigned short data1, unsigned long data2)

This function schedules an OB to be scheduled by WinLC. The OB will be scheduled to run at a priority relative to other OBs as configured by the Hardware Configuration utility of STEP 7. (That is, if an OB80 is scheduled, it interrupts OB1, OB35, etc.)

When you schedule an OB, select an OB whose standard S7 behavior is closest to the way you plan to use the OB. You should also choose an OB that is normally triggered by some asynchronous event, such as an error or a diagnostic event.

When selecting an OB to schedule, consider the following OBs:

- OB80 (Time error, such as a watchdog alarm)
- OB4x (Hardware interrupts)
- OB82 (Diagnostic Alarm interrupt)
- OB83 (Insert/Remove Module interrupt)
- OB87 (Communication Error interrupt)

The parameters that you enter for the ScheduleOB function are stored in the first 12 bytes of local data (L memory) when the OB which was scheduled by this function is executed by WinLC.

Note

The last 8 bytes of the local data contain the time stamp when the event was created. This data is entered when you call the ScheduleOB function.

To understand how the parameters of the ScheduleOB function relate to the local data of the specific OB to be scheduled, refer to the STEP 7 manual *System and Standard Functions for S7-300 and S7-400*. The arguments in the ScheduleOB function follow the same order as that in the documentation.

The documentation also describes data words for each OB. Depending on the type of OB, the documentation divides the data words (the last two parameters) in different ways. However, you can use these data words according to your own requirements: WinLC does not interpret the data type or data parameters when scheduling the OB. The data words are copied to the local data (L memory) for the OB when the OB is scheduled to be executed. You can then access this information in your implementation of the OB that you schedule.

Note

If you require that the entry in the Module Information/Diagnostic Buffer of STEP 7 be displayed with descriptive text, you must use the correct data types. These data types are typically listed as "Reserved" entries and are not documented in the S7 or STEP 7 documentation.

Consider the following valid data types for the datType2 and dataType 1 parameters of the ScheduleOB function:

For the dataType2 parameter:

Value (hexadecimal)	Description
C1	32-bit double word
C4	Two 16-bit binary values
C8	32-bit signed value
C9	Two 16-bit signed values
CA	32-bit floating point value
CD	32 Relative time in milliseconds

For dataType1 parameter:

Value (hexadecimal)	Description
51	16-bit field: unspecified numeric value
58	16-bit field: time in milliseconds
59	16-bit integer value
5B	Two 8-bit binary value

Data Access Helper Class

Caution

Your in-process function (thread) can corrupt WinLC memory if invalid addresses are used when writing data. When developing your application, always follow proper programming guidelines and industrial standards. Always ensure that your application has been carefully tested before running the application with WinLC or any other application.

The CWinLCReadData and CWinLCReadWriteData classes comprise the data access helper class. The read and write functionality has been divided up as a means of providing very basic security on the input and output parameters of the Execute function.

Use the functions of the data access helper class to access data in WinLC. These functions help avoid programming errors, such as out-of-range values or writing to invalid pointers. They also perform the necessary byte-swapping to convert data from the "big endian" format used in the S7 CPU architecture to the "little endian" format required for Microsoft operating systems, including Windows NT.

The data access helper class references the data and does not copy it, so it is only valid within the scope of the Execute function.

CWinLCReadData Member Functions

The CWinLCReadData retrieves data from the input data buffer passed into the IWinLCLogicExtenston Execute() function as S7 data types. Each function follows the following format:

```
ReadS7<datatype>(long byteOffset, <datatype>& value)
```

(where **`<datatype>`** is the name of the S7 data type. For more information about the S7 data types, refer to the online help for STEP 7.)

These functions return either true or false, depending on whether or not the Read function succeeded or failed. (For example, the Read function could fail if the value is out of range.) The **`byte offset`** is the beginning location of the value in the data buffer in bytes (for example, the fourth DWORD is at byte offset 12), and **`value`** is the destination location for the data item to be placed. The Read function automatically performs any byte-format conversions necessary between native byte format and internal WinLC byte format.

bool ReadS7BOOL(long byteOffset, int bitNo, bool &value)

This function retrieves the value of the bit requested and return that value in *value*.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *int bitNo*: bit number to retrieve (indexing from right to left)
- *bool &value*: value of the bit to retrieve

bool ReadS7BYTE(long byteOffset, BIT8 &value)

This function retrieves a byte (8 bits) from the input buffer. The value is returned in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *BIT8 &value*: value of the byte retrieved

bool ReadS7CHAR(long byteOffset, char &value)

This function retrieves an 8-bit character from the input buffer. The character is returned in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *char &value*: value of the character retrieved

bool ReadS7DATE(long byteOffset, UINT16 &value)

This function retrieves a 16-bit value from the input buffer and returns it as an unsigned integer (S7 data type: Date) in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *UINT16 &value*: value of the data retrieved

bool ReadS7DINT(long byteOffset, SINT32 &value)

This function retrieves 32 bits from the input buffer and returns the value as a signed integer.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *SINT32 &value*: value of the data retrieved

bool ReadS7DWORD(long byteOffset, BIT32 &value)

This function retrieves a 32-bit double word from the input buffer and returns the value as a BIT32 in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *BIT32 &value*: value of the data retrieved

bool ReadS7INT(long byteOffset, SINT16 &value)

This function reads 16 bits from the input buffer and returns the value as a signed 16-bit integer in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *SINT16 &value*: value of the data retrieved

bool ReadS7REAL(long byteOffset, float &value)

This function reads 32 bits from the input buffer and returns the value as a floating-point number in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *float &value*: value of the data retrieved

bool ReadS7S5TIME(long byteOffset, BIT16 &value)

This function retrieves a 16-bit value from the input buffer and returns the data in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *BIT16 &value*: value of the data retrieved

bool ReadS7STRING(long byteOffset, UINT8 readMax, char* string)

This function reads a string from the input buffer, beginning at *byteOffset* and continuing until all characters of the string are read or until *readMax* characters are read. The string is placed in the buffer pointed to by *string*.

Caution

Reading a value larger than the buffer size could cause loss of data. The *readMax* value should always be equal to or less than the size of the buffer pointed to by *char*string*. Otherwise, this operation could inadvertently write over data.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *UINT8 readMax*: maximum number of characters to retrieve
- *char* string*: buffer to place the string in

bool ReadS7STRING_LEN(long byteOffset, UINT8 &maxLen, UINT8 &curLen)

This function retrieves the length information about a string. The maximum length of the string is returned in the *maxLen* parameter, and the current length of the string is returned in the *curLen* parameter.

Caution

Reading a value larger than the buffer size could cause loss of data. The *readMax* value should always be equal to or less than the size of the buffer pointed to by *char*string*. Otherwise, this operation could inadvertently write over data.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *UINT8 &maxLen*: maximum length of the string
- *UINT8 &curLen*: current length of the string

bool ReadS7TIME(long byteOffset, SINT32 &value)

This function retrieves a 32-bit value from the input buffer and returns it as a signed 32-bit integer in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *SINT32& value*: value of the data retrieved

bool ReadS7TIME_OF_DAY(long byteOffset, UINT32 &value)

This function reads a 32-bit value from the input buffer and returns it as an unsigned 32-bit integer in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *UINT32 &value*: value of the data retrieved

bool ReadS7WORD(long byteOffset, BIT16 &value)

This function retrieves a 16-bit value from the input buffer and returns it as a BIT16 in the *value* parameter.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to retrieve
- *BIT16 &value*: value of the data retrieved

CWinLCReadWriteData Member Functions

The CWinLCReadWriteData class extends the CWinLCReadData class by adding methods for writing data as S7 data types to the input/output data buffer passed into the IWinLCLogicExtension Execute() function. Each function follows the following format:

`WriteS7<datatype>(long byteOffset, <datatype>& value)`

(where **`<datatype>`** is the name of the S7 data type. For more information about the S7 data types, refer to the online help for STEP 7.)

These functions return either true or false, depending on whether or not the write succeeded or failed. The **`byte offset`** is the value's beginning location in the data buffer, and **`value`** is the data item to be placed into the buffer. The function automatically performs any byte-format conversions necessary between native byte format and internal WinLC byte format.

Note

The CWinLCReadWriteData class inherits from the CWinLCReadData class, so all of the read functions appearing in the CWinLCReadData class are available in the CWinLCReadWriteData class.

bool WriteS7BOOL(long byteOffset, int bitNo, bool &value)

This function sets the bit at position *bitNo* in byte *byteOffset* to the value of *value*.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *int bitNo*: bit number to write (indexing from right to left)
- *bool &value*: value of the bit to write

bool WriteS7BYTE(long byteOffset, BIT8 &value)

This function writes the data in *value* to a byte (8 bits) in the output buffer.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *BIT8 &value*: value of the byte to be written

bool WriteS7CHAR(long byteOffset, char &value)

This function writes an 8-bit character to the output buffer in position *byteOffset*.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *char &value*: value of the character to write

bool WriteS7DATE(long byteOffset, UINT16 &value)

This function writes the 16-bit *value* to the output buffer.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *UINT16 &value*: value of the data to write

bool WriteS7DINT(long byteOffset, SINT32 &value)

This function writes the 32-bit *value* to the output buffer at position *byteOffset*.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *SINT32 &value*: value of the data to write

bool WriteS7DWORD(long byteOffset, BIT32 &value)

This function writes the 32-bit double word *value* to the output data buffer at position *byteOffset*.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *BIT32 &value*: value of the data to write

bool WriteS7INT(long byteOffset, SINT16 &value)

This function writes the 16-bit signed integer *value* to the output buffer at position *byteOffset*.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *SINT16 &value*: value of the data to write

bool WriteS7REAL(long byteOffset, float &value)

This function writes the 32-bit floating-point *value* to the output buffer at position *byteOffset*.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *float &value*: value of the data to write

bool WriteS7S5TIME(long byteOffset, BIT16 &value)

This function writes a 16-bit *value* to the output buffer.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *BIT16 &value*: value of the data to write

bool WriteS7STRING(long byteOffset, char* string)

This function writes a string to the output buffer. The string is expected to be NULL terminated. If the entire string cannot fit into the allocated space (for example, the current length is larger than the maximum length, or there is not enough space left in the output buffer), the write operation fails.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *char* string*: pointer to a NULL terminated string

bool WriteS7TIME(long byteOffset, SINT32 &value)

This function writes the 32-bit *value* to the output buffer.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *SINT32& value*: value of the data to write

bool WriteS7TIME_OF_DAY(long byteOffset, UINT32 &value)

This function writes the 32-bit *value* to the output buffer.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *UINT32 &value*: value of the data to write

bool WriteS7WORD(long byteOffset, BIT16 &value)

This function writes a 16-bit *value* to the output buffer.

Return Value:

- *bool*: success or fail of the operation (true = success)

Parameters:

- *long byteOffset*: buffer index of the data to write
- *BIT16 &value*: value of the data to write

Asynchronous Processor Class

The asynchronous processor executes user-defined events on a thread of execution separate from the WinLC Logic Extension (IWinLCLogicExtension). This allow the COM object to schedule and execute actions that may take a long time while allowing WinLC to continue processing.

For example: WinLC controls a stapler assembly line. One of the WinLC tasks is to send an E-mail notification when supplies are running low. This can be accomplished by using WinAC Basis ODK. However, putting all of the E-mail functionality in the SFB65002 Execute() function could cause an unacceptable increase in the scan cycle. You could avoid this problem by using the asynchronous processor (AsyncProc) to schedule a "Send E-mail Notification" event. This allows WinLC to maintain a fast scan cycle while sending E-mail at the same time.

The following classes comprise the asynchronous processor class:

- AsyncProc: This class processes events posted to it on a thread of execution separate from the main program.
- EventMsg: This is the base class to Asynchronous Events: all asynchronous events posted to the asynchronous processor should be derived from this class. The Execute() function should be overloaded in the derived class to provide custom processing for the event.
- Queue: This is a basic queue ("first in, first out") class. The asynchronous processor uses it for scheduling events to process.

The following table lists the general sequence of events for using the asynchronous processor:

Pre-runtime	1. Create a new event class derived from the class EventMsg. 2. Add your new data and methods in the new class. 3. Override the Execute() function to perform the required actions (using the data and methods that were added to the derived class).
Runtime	1. Create an event class using the "new" operator. 2. Use the ScheduleEvent method to post the event to the asynchronous processor. 3. Call the GetStatus() method of the event to determine if the event has been processed. 4. Call the GetResult() method of the event to get success/fail information returned from the Execute() function.

If you use the WinAC Basis ODK application wizard to create the COM object, the COM object accesses the asynchronous processor through the WinLC Logic Extension class (the class containing the IWinLCLogicExtension interface).

Asynchronous Events

All events posted to the asynchronous processor should be derived from the EventMsg class. In the derived class, the event execution code needs to be placed in an override of the Execute() function. The asynchronous processor calls this Execute() function to perform the event-specific tasks.

You are responsible for creating the event object on the heap (for example, using the "new" operator). However, you can set the responsibility for deallocating the object’s memory. Using the SetDelTime() method for the event, the event object can be set for deallocation either by the asynchronous processor or by the your code, depending on whether or not post-processing status information is required.

AsyncProc Class

The AsyncProc class processes events posted to it on a thread of execution separate from the main program (IWinLCLogicExtension).

AsyncProc()

Default Constructor. This function constructs a AsyncProc object. The object creates a new thread of execution and waits to be started using the ResumeThread() function.

Return: none

Parameters: none

AsyncProc(IWinLCServices *WinLCSvc)

This constructor sets the IWinLCServices interface pointer in the member data.

Parameters:

- *IWinLCServices* WinLCSvc*: the pointer to an IWinLCServices interface

void PauseThread()

This function temporarily stops the thread execution. Use the ResumeThread function to continue processing.

Return: none

Parameters: none

void ResumeThread()

This function restarts the thread processing.

Return: none

Parameters: none

void ResumeThread(IWinLCServices* WinLCSvc)

This function restarts the thread processing and reset the IWinLCServices interface pointer.

Return: none

Parameters: none

void StopThread()

This function signals the thread to terminate.

Return: none

Parameters: none

void Execute()

Note

The Execute function is a member of the derived class that you created with the application wizard. You "implement" or "define" this function, but you do not call the Execute function.

This function contains the monitoring loop for the thread. All custom user logic should go in this function, following the "// Add Thread execution code here" comment.

Return: none

Parameters: none

EventMsg Class

This is the base class to Asynchronous Events: all asynchronous events posted to the asynchronous processor should be derived from this class. The Execute() function should be overloaded in the derived class to provide custom processing for the event.

EventMsg()

Default constructor.

long Execute()

This function performs the event-specific tasks. Execute() should be overridden in the derived event class to provide user-defined functionality. This function can only be called by the asynchronous processor.

Note

You "implement" or "define" this function, but you do not call the Execute() function.

Return:

- *Long*: success or fail of the Execute() function. Your code determines the definition of success or failure.

Parameters: None

UINT GetDeleteTime()

This function returns the currently set deletion time for the event object. Return values can be: ON_EXECUTE or USER_DELETE. For more information, refer to the description of the DelTime enumeration.

Return:

- *UINT*: DelTime enumeration value

Parameters: None

long GetResult()

This function returns the success or failure status, as received from the Execute() function.

Return:

- *Long*: The return value from the Execute() function

Parameters: None

UINT GetStatus()

This function returns the current processing state of the event. Return values can be: PENDING, EXECUTING, or COMPLETE. For more information, refer to the description of the EventStatus enumeration.

Return:

- *UINT*: EventStatus enumeration value

Parameters: None

void SetDelTime(DelTime WhenDelete)

This function sets the responsibility for deallocating the memory space for the object. The asynchronous processor will delete the object if the delete time is ON_EXECUTE. Otherwise, you are responsible for managing the deallocation of the memory space for the object.

Return: None

Parameters:

- *DelTime WhenDelete*: Enumerated type defined in EventMsg. *WhenDelete* can be one of two values:
    - ON_EXECUTE : The asynchronous processor deletes the object.
    - USER_DELETE: Your logic deletes the object.

Enumerated Types

DelTime

DelTime is an enumeration of constants used to define when the event object should be deleted:

- ON_EXECUTE: The asynchronous processor deletes the object after it has been executed.
- USER_DELETE: You are responsible for deleting the object. Use this when post-processing information is required from the event.

EventStatus

EventStatus is an enumeration of values used to define the current processing status of the event:

- PENDING: The event has been queued, but has not yet been processed.
- EXECUTING: The event is currently being executed.
- COMPLETE: The event has finished being processed, and return information is available

Queue Class

This is a basic queue ("first in, first out") class. The asynchronous processor uses it for scheduling events to process.

long Dequeue()

This function removes an object from the queue of objects to be executed by the asynchronous processor.

Return: none

Parameters: none

long Enqueue()

This function places an object into the queue of objects to be executed by the asynchronous processor.

Return: none

Parameters: none

long GetSize()

This function returns the current size of the queue.

Return:

Parameters: none

Queue(long Size)

This function allocates a specified amount of memory for the queue.

Return: none

Parameters:

- *Long Size*: Inital size of the queue.

Queue()

This function allocates memory for the queue..

Return: none

Parameters: none

Monitor Class

The Monitor class consists of the MonitorThread class.

MonitorThread Class

The MonitorThread class provides a separate line of execution for the WinAC Basis ODK to monitor events external to the its process.

The Execute() function is where the main thread monitoring loop is located. Place all custom processing or monitoring immediately following the comment: "// Add Thread execution code here"

Note

The Execute function is a member of the derived class that you created with the application wizard. You "implement" or "define" this function, but you do not call the Execute function.

The MonitorThread class consists of:

- Construction functions
- Thread Control functions

Construction Functions

MonitorThread ()

Default Constructor. This function constructs a MonitorThread object. The object creates a new thread of execution and waits to be started using the ResumeThread() function.

Parameters: none

MonitorThread(IWinLCServices* WinLCSvc)

This constructor sets the IWinLCServices interface pointer in the member data. The IWinLCServices interface provides the mechanism for scheduling an OB to be processed by WinLC. This can also be set using the ResumeThread(IWinLCServices* WinLCSvc) overloaded function.

Parameters:

- *IWinLCServices* WinLCSvc*: the pointer to an IWinLCServices interface

Thread Control Functions

void PauseThread()

This function temporarily stops the thread execution. Use the ResumeThread function to continue processing.

Return: none

Parameters: none

void ResumeThread()

This function restarts the thread processing.

Return: none

Parameters: none

void ResumeThread(IWinLCServices* WinLCSvc)

This function restarts the thread processing and reset the IWinLCServices interface pointer.

Return: none

Parameters: none

void StopThread()

This function signals the thread to terminate.

Return: none

Parameters: none

void Execute()

Note

The Execute function is a member of the derived class that you created with the application wizard. You "implement" or "define" this function, but you do not call the Execute function.

This function contains the monitoring loop for the thread. All custom user logic should go in this function, following the "// Add Thread execution code here" comment.

Return: none

Parameters: none

WinAC Basis ODK Library for STEP 7

WinAC Basis ODK provides a STEP 7 library ("WinAC Basis ODK") that includes two SFBs:

- SFB65001 ("CREA_COM")
- SFB65002 ("EXEC_COM")

You insert these SFBs into your STEP 7 program to execute your application program as part of the WinLC scan cycle. In order to use these SFBs in your STEP 7 program, you must perform the following tasks:

- You must have loaded the WinAC Basis ODK library into STEP 7.
- You must insert both SFBs into your STEP 7 program.

SFB65001 ("CREA_COM")

SFB65001 creates an instance of the COM object specified by the ProgID parameter. Your COM object is required to implement the interface specified for WinAC Basis ODK objects. The following table shows the interface for SFB65001 ("CREA_COM"):

Address	Declaration	Name	Data Type	Comment
0.0	in	ProgID	STRING[254]	Program ID of the object to be created
256	out	Ret_Val	WORD	SFB return code: Error code or object instance handle

SFB65001 performs the following actions:

1. SFB65001 calls the CoInitializeEx function (or ensures that it was previously called) with the COINIT_MULTITHREADED option. SFB65001 then converts the input parameter ProgID to a ClassID.
2. If the COM object has already been created, SFB65001 maintains the WinAC Basis ODK handle for the object already created (an index to locate the object pointer). If the COM object has not been created, SFB65001 calls the CoCreateInstance function to create the object.
3. SFB65001 creates the IWinLCServices COM object if it hasn't been created yet. (There is only one instance of this object created.)
4. If the COM object has not already been created, SFB65001 creates it, using the ClassID from the program and the interface ID defined by WinAC Basis ODK. SFB65001 adds this object instance to the internal list of created WinAC Basis ODK objects.
5. SFB65001 invokes the Activate method if this is the first call to this SFB after leaving STOP or if the COM object was just created.
6. SFB65001 sets the Ret_Val to the WinAC Basis ODK handle (or error code) and sets the BR bit. To see ways to check for the return value, refer to the sample program ("Histogram") installed by WinAC Basis ODK.

Error Codes for SFB65001

Error Code	Message	Description
0x807F	ERROR_INTERNAL	An internal error occurred.
0x8001	E_EXCEPTION	An exception occurred.
0x8102	E_CLSID_FAILED	The call to CLSIDFromProgID failed.
0x8103	E_COINITIALIZE_FAILED	The call to CoInitializeEx failed.
0x8104	E_CREATE_INSTANCE_FAILED	The call to CoCreateInstance failed.

SFB65002 ("EXEC_COM")

SFB65002 calls the Execute function of the COM object specified by the OBJHandle parameter. The following table shows the interface for SFB65002 ("EXEC_COM"):

Address	Declaration	Name	Data Type	Description
0.0	in	OBJHandle	WORD	Handle returned from SFB65001 ("CREA_COM")
2.0	in	Command	DWORD	Index of function or command to execute
6.0	in	InputData	ANY	Pointer to input function area
16.0	in	OutputData	ANY	Pointer to function output area
26.0	out	STATUS	WORD	SFB error code or return code from Execute.

SFB65002 performs the following actions:

1. SFB65002 verifies that SFB65001 ("CREA_COM") was called and that the object handle is valid.
2. SFB65002 processes the ANY pointers and returns error codes for invalid ANY pointer parameters.
3. SFB65002 assigns the input and output ANY pointer areas to the WinLC Data Access COM object.
4. SFB65002 invokes the customer WinAC Basis ODK Execute function.
5. SFB65002 sets the STATUS with the Execute return code (unless there was a previous error) and returns to the STEP 7 program.

Error Codes for SFB65002 ("EXEC_COM")

All other error codes are user-defined in the individual COM servers.

Error Code	Message	Description
0	NO_ERRORS	Success
0x807F	ERROR_INTERNAL	An internal error occurred.
0x8001	E_EXCEPTION	An exception occurred.
0x8002	E_NO_VALID_INPUT	Input: the ANY pointer is invalid.
0x8003	E_INPUT_RANGE_INVALID	Input: the ANY pointer range is invalid.
0x8004	E_NO_VALID_OUTPUT	Output: the ANY pointer is invalid.
0x8005	E_OUTPUT_RANGE_INVALID	Output: the ANY pointer range is invalid.
0x8006	E_OUTPUT_OVERFLOW	More bytes were written into the output buffer by the COM object than were allocated.
0x8007	E_NOT_INITIALIZED	COM system has not been initialized: no previous call to SFB65001 ("CREA_COM").
0x8008	E_HANDLE_OUT_OF_RANGE	The supplied handle value does not correspond to a valid COM object.

Debugging the COM Object

Use the debugger of Visual C++ to test your COM object with the STEP 7 program running on WinLC.

Note

The use of breakpoints in your COM object can cause WinLC to exceed the maximum scan cycle.

Debugging your application requires the following steps:

- Create the STEP 7 program (using the WinAC Basis ODK SFBs).
- Provide the path name for the WinLC executable.
- Download the STEP 7 program to WinLC.
- Debug your application (DLL).

Use the following procedure to test your custom application (DLL) with WinLC:

1. In Visual C++, select the Debug tab. Visual C++ requests the path name for the executable (.EXE) that uses your COM object.
2. Enter the path name for WinLC by browsing to the directory where you installed WinLC and select the following file:

S7WLCAPX.EXE

3. After Visual C++ starts WinLC, start the SIMATIC Manager and open the STEP 7 program that uses the SFBs of WinAC Basis ODK.
4. Download your STEP 7 program to WinLC.
5. Place WinLC in RUN mode or RUN-P mode.
6. Test your COM object by triggering events and setting breakpoints.

Note

Using breakpoints to test your application could cause WinLC to exceed the maximum scan cycle time.

You can use the tuning panel of WinLC to monitor the effects of COM object on the scan cycle.

In Case WinLC Crashes while Testing Your COM object

If your application causes WinLC to crash (to abort unexpectedly):

- If you are using the Debugger of Visual C++ to test your application, select the **Debug > Stop Debugging** to recover from the crash.
- If you are not testing your application with the Visual C++ Debugger, terminate the following process:
  - ❑ S7WLCAPX.EXE
  - ❑ S7WLPMSX.EXE

  Do not terminate the S7WLSAPX.EXE process. This is a daemon process that always runs in the background.

Remarks Form

Your comments and recommendations will help us to improve the quality and usefulness of our publications. Please take the first available opportunity to fill out this questionnaire and return it to Siemens.

Please give each of the following questions your own personal mark within a range from 1 (very good) to 5 (very poor).

- £ Do the contents meet your requirements?
- £ Is the information you need easy to find?
- £ Is the text easy to understand?
- £ Does the level of technical detail meet your requirements?
- £ Please rate the quality of the graphics and tables.

Additional comments:

_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _

Please check any industry that applies to you:

❒ Automotive
❒ Chemical
❒ Electrical Machinery
❒ Food
❒ Instrument and Control
❒ Non electrical Machinery
❒ Petrochemical
❒ Pharmaceutical
❒ Plastic
❒ Pulp and Paper
❒ Textiles
❒ Transportation
❒ Other ____________________________

Mail your response to:

SIEMENS ENERGY & AUTOMATION, INC

ATTN: TECHNICAL COMMUNICATIONS M/S 519

3000 BILL GARLAND ROAD

PO BOX 1255

JOHNSON CITY TN USA 37605-1255

Include this information:

From

Name: _

Job Title: _

Company Name _

Street: _

City and State: _

Country: _

Telephone: _

Index