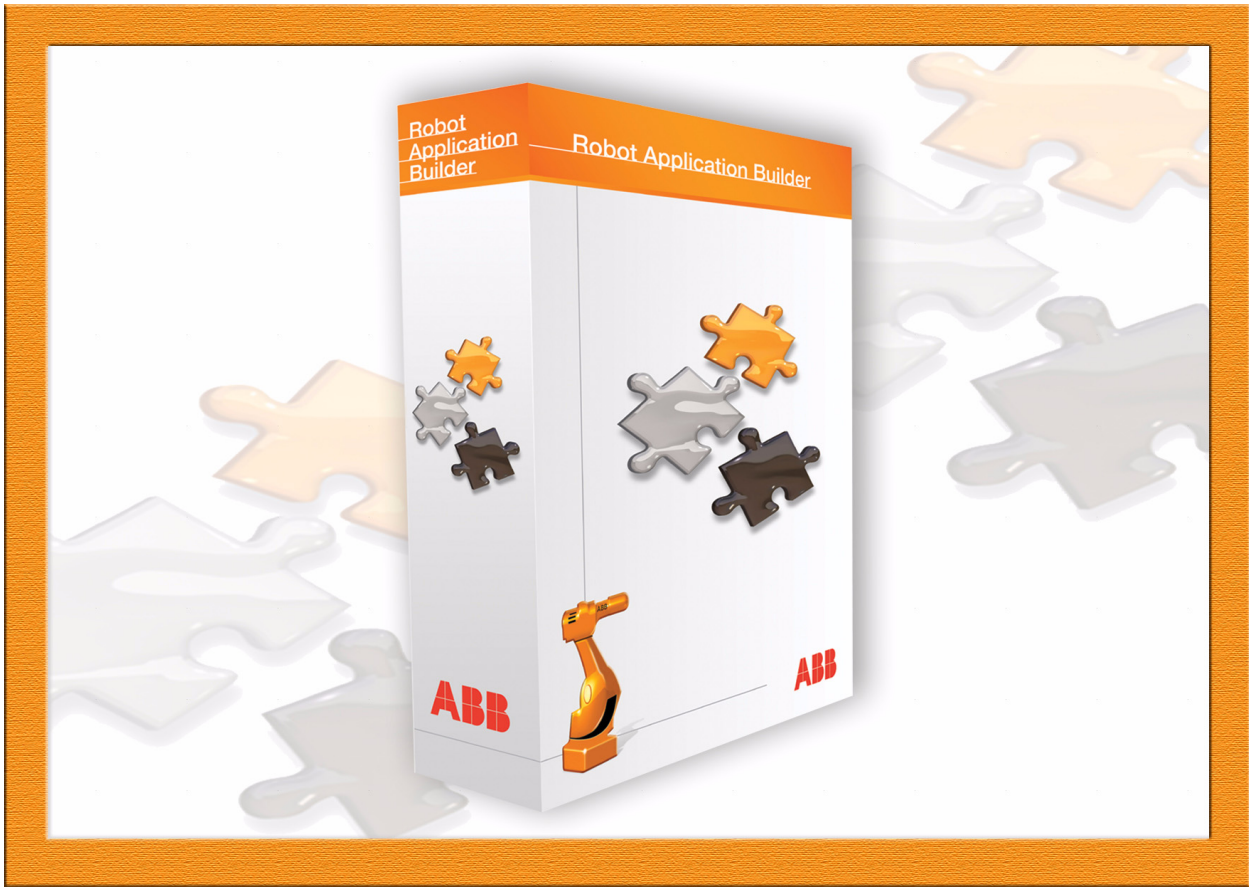
Application manual

Robot Application Builder

Industrial Software Products
RobotWare 5.0



Application manual

Robot Application Builder

RobotWare 5.0

Document ID: 3HAC028083-001

Revision: D

ABB AB
Robotics Products
SE-721 68 Västerås
Sweden

Overview . . . 7
Product documentation, M2004 . . . 10
Safety . . . 12

1 Introduction 13

1.1 About Robot Application Builder . . . 14
1.2 Documentation and help . . . 16
1.3 Terminology . . . 18

2 Installation and development environment 21

2.1 Installation overview . . . 22
2.2 How to obtain and install a license key for RAB 5.09 or earlier . . . 26
2.3 How to set up your PC to robot communication . . . 27
2.4 Development environment . . . 29
2.5 Two development models - virtual and real . . . 31
2.6 Conversion of VS 2005 projects to Visual Studio 2008 . . . 33

3 Run-time environment 35

3.1 Two platforms - PC and FlexPendant . . . 36
3.2 Running PC Applications . . . 40
3.2.1 Licence verification - applies only to versions earlier than RAB 5.10 . . . 40
3.2.2 Mastership . . . 41
3.2.3 PC application configuration . . . 43
3.2.4 Communication between PC and controller . . . 46
3.3 Running FlexPendant Applications . . . 47
3.3.1 Components, assemblies and dlls . . . 47
3.3.2 Deployment of FlexPendant application to a robot system . . . 48
3.3.3 Communication between FlexPendant and controller . . . 51
3.3.4 Understanding FlexPendant application life cycle . . . 52
3.3.5 FlexPendant TpsView attribute . . . 54
3.3.6 ITpsViewSetup and ITpsViewActivation . . . 58
3.4 Release upgrades and compatibility . . . 60

4 Developing RAB applications 63

4.1 Introduction . . . 64
4.2 Analysis and design . . . 65
4.3 Controller events and threads . . . 67
4.4 User Authorization System . . . 70
4.5 Exception handling . . . 72
4.6 How to use the online help . . . 75

5 Using the FlexPendant SDK 77

5.1 Introduction . . . 77
5.1.1 About this chapter . . . 77
5.1.2 System features supporting the use of customized screens . . . 78
5.2 Setting up a new project . . . 80
5.2.1 Using the project template in Visual Studio . . . 80
5.2.2 Setting up design support for FlexPendant controls . . . 83
5.3 Building the user interface . . . 84
5.3.1 Introduction to visual design support . . . 84
5.3.2 GUI controls and memory management . . . 93
5.3.3 Container style . . . 97
5.3.4 Command bar . . . 101

5.3.5 FlexPendant fonts . . . 103
5.3.6 The use of icons . . . 104
5.3.7 TabControl . . . 106
5.3.8 Button, TextBox and ComboBox . . . 109
5.3.9 AlphaPad . . . 110
5.3.10 ListView . . . 114
5.3.11 CompactAlphaPad and NumPad . . . 116
5.3.12 GTPUMessageBox . . . 117
5.3.13 GTPUFileDialog . . . 119
5.3.14 DataBinding of RAPID data and IO signals . . . 122
5.4 Launching other views . . . 128
5.4.1 Using launch service . . . 128
5.4.2 Using standard dialogs to modify data . . . 131
5.5 Using the Controller API . . . 133
5.5.1 ABB.Robotics.Controllers . . . 133
5.5.2 Accessing the controller . . . 136
5.5.3 Rapid domain . . . 140
5.5.3.1 Working with RAPID data . . . 140
5.5.3.2 Handling RAPID arrays . . . 148
5.5.3.3 ReadItem and WriteItem methods . . . 151
5.5.3.4 UserDefined data . . . 152
5.5.3.5 RAPID symbol search . . . 157
5.5.3.6 RAPID execution . . . 162
5.5.3.7 Modifying modules and programs . . . 164
5.5.4 IO system domain . . . 166
5.5.5 Event log domain . . . 172
5.5.6 Motion domain . . . 174
5.5.7 File system domain . . . 177
5.5.8 System info domain . . . 179

6 Robust FlexPendant applications 181
6.1 Introduction . . . 182
6.2 Memory management . . . 184
6.3 Performance . . . 188
6.4 Reliability . . . 193

7 Using the PC SDK 199
7.1 Controller API . . . 200
7.2 Create a simple PC SDK application . . . 202
7.3 Discovery domain . . . 210
7.4 Accessing the controller . . . 212
7.5 Rapid domain . . . 219
7.5.1 Working with RAPID data . . . 219
7.5.2 Handling arrays . . . 227
7.5.3 ReadItem and WriteItem methods . . . 230
7.5.4 UserDefined data . . . 231
7.5.5 RAPID symbol search . . . 239
7.5.6 Working with RAPID modules and programs . . . 245
7.5.7 Enable operator response to RAPID UI-instructions from a PC . . . 248
7.6 IO system domain . . . 254
7.7 Event log domain . . . 261
7.8 Motion domain . . . 263
7.9 File system domain . . . 265
7.10 Messaging domain . . . 268

8 Debugging and troubleshooting 279

8.1 FlexPendant - Debugging and troubleshooting . . . 279
8.1.1 Debug output . . . 279
8.1.2 Debugging the virtual FlexPendant . . . 282
8.1.3 Debugging the FlexPendant device . . . 286
8.1.4 Troubleshooting FlexPendant applications . . . 289
8.2 PC - Debugging and troubleshooting . . . 292
8.2.1 Debugging . . . 292
8.2.2 Troubleshooting . . . 296

9 Localizing a FlexPendant application 299

9.1 Adding support for several languages . . . 300

10 Packaging RAB applications 309

10.1 Deployment of a PC SDK application . . . 309
10.1.1 Overview . . . 309
10.2 Deployment of a FlexPendant SDK application . . . 311
10.2.1 Overview . . . 311
10.2.2 Deployment of an application without a license . . . 312
10.2.3 Deployment of a licensed application . . . 315
10.2.4 Deployment using FTP . . . 318

Overview

About this manual

Robot Application Builder (RAB) is a software tool, which enables programmers to develop customized operator interfaces for the IRC5 robot controller.

The purpose of this manual is to help software developers get started with RAB application development.

Usage

Robot Application Builder targets two different platforms. To develop a FlexPendant application you use the FlexPendant SDK. To develop a PC application, on the other hand, you use the PC SDK. This manual covers application development using both of these SDKs.

Who should read this manual?

This manual is mainly intended for software developers, who use RAB to create robot applications adapted to end-user needs, but is also useful for anyone who needs an overview of Robot Application Builder.

Prerequisites

The reader should

- be familiar with IRC5, the FlexPendant and Robot Studio.
- be used to Microsoft Visual Studio and Windows programming.
- be familiar with one of the .NET programming languages C# or Visual Basic.NET. For PC applications Visual J# and Visual C++ should also work.
- be used to object oriented programming.

Organization of chapters

Most chapters in this manual deal with topics that apply to both platforms (PC and FlexPendant). Chapter 5, 6 and 9, however, cover the FlexPendant SDK specifically, whereas chapter 7 deals only with the PC SDK. Code samples are written in C# and Visual Basic. The manual is organized as follows:

Chapter	Contents
1.	Introduction. Terminology. Safety.
2.	Installation and setup. Development environment . Virtual robot technology.
3.	Two run-time platforms: PC and FlexPendant. Selecting the platform. Software architecture. Run-time environment for PC/FlexPendant applications. How clients access controller resources and communicate with the robot controller. Application configuration. Life cycle of a FlexPendant application. Upgrades and compatibility.
4.	Developing RAB applications. Analysis and design. Important programming issues: controller events and threads, UAS, exception handling. Online help.
5.	Using the FlexPendant SDK. Visual design support. GUI controls. Launching standard views. Data binding. How to add controller functionality using the Controller API. Programming issues and code samples in VB and C#.
6.	How to develop well performing and robust FlexPendant applications. Memory management, performance and reliability. Exception handling.

Continues on next page

Continued

Chapter	Contents
7.	Using the PC SDK. How to add controller functionality using the Controller API. Programming issues and code samples in VB and C#.
8.	Testing, debugging and troubleshooting RAB applications. Using printouts, error codes in exceptions etc. Checklist for contacting a service organization.
9.	How to add support for several languages to a custom FlexPendant application.
10.	How to deploy RAB applications. How to create an additional option and how to make a product of a FlexPendant application.

References

Reference	Document Id
Operating Manual -IRC5 with FlexPendant	3HAC 16590-1
Operating Manual - RobotStudio	3HAC032104-001
Technical reference manual - RAPID Instructions, Functions and Data types	3HAC16581-1

Revisions

Revision	Description
-	First edition From RAB 5.08 onwards this manual replaces: Robot Application Builder - PC SDK User's Guide (3HAC 024913-001) and Robot Application Builder - FlexPendant SDK User's Guide (3HAC 024914-001)
A	Improvements and updates for RAB 5.09.
B	Additions and further improvements for RAB 5.10: Installation chapter: no license required, working with several PC SDK versions. (2.1- 2.2) *PC application configuration*, how to use *App.config*. (3.2.3) `TpsFont` internally retrieves appropriate font for the active language, e.g. Chinese (5.3.5). The operating system of the first generation FlexPendant device (SX TPU 1) does not support images of more than 256 colors. (5.3.6, 8.1.4) New FP SDK domain *SystemInfoDomain* (5.5.8). Maximum size of an FP SDK application (6.1). No initial events guaranteed (4.3, 5.5.2, 5.5.4, 7.3, 7.6). *Create a simple PC SDK application* (7.2). FP and PC SDK Controller API, working with RAPID data. (5.5.3.1 and 7.5.1). High priority event subscriptions in PC SDK (7.5.1). New PC SDK domain *Messaging* (7.10). Troubleshooting PC SDK applications (8.2.2). Localizing a FlexPendant application (9).
C	Improvements and updates for RAB 5.11. New installation described in *Installation overview* (2.1)(

Continues on next page

Continued

Revision	Description
D	Additions and improvements for RAB 5.12: RobotStudio Community (1.2) Some details about installing RAB on Windows Vista (2.1) Description of improved Project Wizard (3.3.5 and 5.3.3) Removed obsolete section about `TpsFont.FontName` (5.3.5) How to access user defined data (5.5.3.4 and 7.5.4) SearchRapidSymbol (5.5.3.5 and 7.5.5) How to access data declared in Shared module (5.5.3.4, 5.5.3.5 and 7.5.4, 7.5.5) Enable operator response to RAPID UI-instructions from a PC (7.5.7) Messaging - new system parameter *RmqMode* (7.10).

Product documentation, M2004

General

The robot documentation may be divided into a number of categories. This listing is based on the type of information contained within the documents, regardless of whether the products are standard or optional. This means that any given delivery of robot products will not contain all documents listed, only the ones pertaining to the equipment delivered.However, all documents listed may be ordered from ABB. The documents listed are valid for M2004 robot systems.

However, all documents listed may be ordered from ABB. The documents listed are valid for M2004 robot systems.

Product manuals

All hardware, robots and controllers, are delivered with a Product manual, which is divided into two parts:

Product manual, procedures

- Safety information
- Installation and commissioning (descriptions of mechanical installation, electrical connections)
- Maintenance (descriptions of all required preventive maintenance procedures including intervals)
- Repair (descriptions of all recommended repair procedures including spare parts)
- Additional procedures, if any (calibration, decommissioning)

Product manual, reference information

- Safety information
- Reference information (article numbers for documentation referred to in Product manual, procedures, lists of tools, safety standards)
- Part list
- Foldouts or exploded views
- Circuit diagrams

The product manual published as a PDF consists of only one file where the two parts are presented together, as one Product manual.

Technical reference manuals

The following manuals describe the robot software in general and contain relevant reference information:

Product manual, procedures

- **RAPID overview**: An overview of the RAPID programming language.
- **RAPID Instructions, Functions and Data types**: Description and syntax for all RAPID instructions, functions and data types.
- **System parameters**: Description of system parameters and configuration workflow

Continues on next page

Application manuals

Specific applications (e.g. software or hardware options) are described in Application manuals. An application manual can describe one or several applications.

An application manual generally contains information about:

- The purpose of the application (what it does and when it is useful)
- What is included (e.g. cables, I/O boards, RAPID instructions, system parameters, CD with PC software)
- How to use the application
- Examples of how to use the application

Operating manuals

This group of manuals is aimed at those having first hand operational contact with the robot, i.e. production cell operators, programmers and trouble shooters. It includes:

- **Getting started - IRC5 and RobotStudio**
- **IRC5 with FlexPendant**
- RobotStudio
- **Trouble shooting**

Safety

Safety of personnel

A robot is heavy and extremely powerful regardless of its speed. A pause or long stop in movement can be followed by a fast hazardous movement. Even if a pattern of movement is predicted, a change in operation can be triggered by an external signal resulting in an unexpected movement.Therefore, it is important that all safety regulations are followed when entering safeguarded space.

Safety of regulations

Before beginning work with the robot, make sure you are familiar with the safety regulations described in *Operating manual - IRC5 with FlexPendant.*

1 Introduction

1.1. About Robot Application Builder

Flexible user interfaces

Robots are usually delivered with a general operator interface. However, different processes require different operator handling and customers need flexible solutions, where the user interface is adapted to user specific needs.

Robot Application Builder (RAB) allows system integrators, third parties or end-users to add their own customized operator interfaces for the IRC5 controller. Such custom applications can either be added to the standard views of the FlexPendant or realized as independent PC applications, which communicate with the robot controller over a network.

To accommodate this, RAB includes the following two components:

- FlexPendant SDK
- PC SDK

NOTE!

RAB applications are not platform independent. You must choose to develop the application for either the FlexPendant or the PC platform.

Ease-of-use on the factory floor

A well-designed user interface presents relevant information and functionality at the right time. In this respect, customized user interfaces are clearly very desirable to the end-user. As tailored solutions are easier to operate, they also optimize user's investment in automation.

RAB is the tool enabling customized user interfaces for IRC5. It is important to keep in mind, however, that RAB itself does not guarantee increased customer value. To achieve this, RAB applications should be developed with care and with a heavy emphasis placed on ease-of-use. Understanding end-users' needs is in fact crucial to realizing the benefits of customized interfaces.

Continues on next page

.NET and Visual Studio

Robot Application Builder uses Microsoft .NET and Microsoft Visual Studio. It is thus assumed that the user knows how to program Windows platforms using Visual Studio. Among programmers .NET distinguishes itself by the programming model provided by the Microsoft .NET Framework. The programming model is very similar for the two run-time platforms supported by Robot Application Builder.

One feature is the programming language independence, leaving the choice to the developer to use any language provided by the integrated development environment Visual Studio. Most prefer C# or Visual Basic, which both offer safe and efficient development. For FlexPendant applications only these two languages are available. For PC applications any of the .NET languages should work, but ABB support is only offered for Visual Basic and C#.

For a Windows programmer familiar with Visual Studio and .NET, developing a customized operator view is rather straight-forward. RAB is fully integrated with Visual Studio, which means that a .NET programmer will recognize wizards for project setup and tools for visual design support and debug etc.

Considerable efforts have been made to allow RAB programmers to start working without having to overcome a steep learning curve. To further speed up the development process, the virtual IRC5 of RobotStudio can be used to test and debug RAB applications.

NOTE!

Some knowledge in Windows programming, object orientation and a .NET programming language is necessary to be able to use Robot Application Builder.

Robustness and performance

Do not underestimate the concern and effort required to achieve the quality and performance needed in industry.

Developing an application for the FlexPendant, a device with limited process and memory resources, can be quite demanding. Issues such as performance and memory management need to be addressed.

As for PC SDK applications as well, there are issues related to performance and reliability that you need to know about before getting started.

In short, even if you are an experienced Windows programmer, you are strongly recommended to read this manual to learn about specific RAB issues when moving to RAB development.

NOTE!

Take the time to study this manual along with the release notes, and avoiding rushing into coding.

1.2. Documentation and help

Introduction

Robot Application Builder includes an extensive on-line help module, which comes with the installation of the product. After having installed RAB, by clicking Windows' **Start** menu, then pointing at Programs > ABB Industrial IT > Robotics IT > Robot Application Builder 5.xx you will find:

- User's Guide - *Application manual - Robot Application Builder*
- FP SDK Reference
- PC SDK Reference
- FP StyleGuide

User's Guide

This user's guide, *Application manual - Robot Application Builder*, is the recommended way to get started if you are new to RAB development. It explains how RAB works. It has code examples in C# and VB and provides hands-on exercises.

The user's guide is provided in two formats, Html Help and PDF. Html is the recommended format for the PC screen and PDF is the best choice if you want printouts.

NOTE!

User's Guide.PDF can be found in the installation directory, at \Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder 5.xx\.

SDK Reference Help

The SDK Reference Help files should be used while programming.

Notice that the FlexPendant SDK and the PC SDK have separate help files:

- FP SDK Reference
- PC SDK Reference

These make up the complete reference to the RAB class libraries. Method signatures are provided in C# and Visual Basic.

Please note that they are **not** integrated with the Visual Studio Help function. Pressing F1 when pointing at code, for example, will open the *Visual Studio Programmer's Reference* or the *.NET Framework Class Library* for the specific language and topic. Many times this is what you want, but if your problem is RAB-related you need to open the appropriate SDK Reference Help to find a solution.

NOTE!

You are recommended to keep the help files open while programming, as you will frequently need them for RAB-related issues.

FP StyleGuide

Good usability is achieved when the program itself communicates possible actions and how to perform them. To encourage careful design of the visual appearance the *FP StyleGuide* is also part of the RAB installation. It is ABB Robotics' best practices for visual design of the FlexPendant user interface.

Continues on next page

RobotStudio Community

In 2008 ABB Robotics launched a new site, *RobotStudio Community*, for its PC Software users. The *Developer Section* of *RobotStudio Community* has information and some videos about programming with the FlexPendant and PC SDKs. At *Content Sharing* there is a complete FlexPendant SDK application available for download. It is recommended for average users and for beginners.

ABB encourage open conversations and believe everyone has something to contribute. The *User Forum* of *RobotStudio Community* has a section dedicated to Robot Application Builder. Here beginners as well as experts discuss code and solutions online. If you are facing a coding problem the *User Forum* should be your first choice, as there is a good chance that someone will give you the help you need to proceed.

RobotStudio Community also provides the means to share code and videos. Your contribution will be appreciated. Working together is many times the key to success.

RobotStudio Community is also where you find RAB releases for free download.

TIP!

Try it out at www.abb.com/robotics > RobotStudio Community.

RAB Product Specification

The product specification for Robot Application Builder (3HAC025595-001) is available from *RobotStudio Community* and from ABB Library. It is updated for each new release.

MSDN

MSDN (Microsoft Developer Network) at www.msdn.com is a one of many sources of information for general programming issues related to .NET and Visual Studio.

1.3. Terminology

About terms and acronyms

Some terms used in this manual are product specific and crucial for understanding. Moreover, acronyms, words formed from initial letters, are sometimes used instead of long terms. To avoid confusion, important terminology is clarified below.

Definitions

Term	Definition
IRC5	ABB's new generation robot controller.
Virtual IRC5	Virtual robot technology makes it possible to run a virtual IRC5 controller, virtual mechanical units and a virtual FlexPendant on the desktop. Included as freeware in ABB's RobotStudio from RAB 5.11.
FlexPendant	ABB's new generation hand held device, used with the IRC5 robot controller. It is developed with Microsoft's latest technology for embedded systems, Windows CE and .NET Compact Framework.
Device	The FlexPendant is a "smart device" in the .NET vocabulary, i.e. a complete computer in itself with its own processor, operating system etc.
Robot Application Builder	ABB software tool, which enables the development of custom operator interfaces for IRC5. Often referred to as RAB.
RAB programmer	A programmer who uses RAB to develop custom applications.
RAB application	A custom application developed with Robot Application Builder.
Controller Application Programming Interface	The public class libraries of Robot Application Builder, which offer robot controller functionality. Also referred to as CAPI.
Network socket	A communication end-point unique to a machine communicating on an Internet Protocol-based network.
Microsoft Visual Studio	The integrated development environment that developers work inside when using the .NET Framework.
Microsoft .NET Framework	An integral Windows component supporting the building and running of applications.
.NET Compact Framework (.NET CF)	Version of Microsoft's .NET framework providing the run-time environment for applications running on embedded devices, such as the FlexPendant. It includes a class library, which is almost a subset of the rich .NET framework for the desktop.
Common Language Runtime	The core runtime engine in the .NET Framework for execution of managed code. Provides services such as cross-language integration, code access security, object lifetime management, and debugging and profiling support.
C# and Visual Basic.NET	.NET programming languages.
Windows CE	The embedded operating system running on the FlexPendant-device.
managed code	Code that is executed and managed by the Microsoft .NET Framework's common language runtime. All code produced by Visual Studio executes as managed code.

Continues on next page

Continued

Term	Definition
unmanaged code	Code that is executed directly by the operating system, outside the .NET Framework. Unmanaged code must provide its own memory management, type checking, and security support, unlike managed code, which receives these services from the common language runtime. All code executing in the robot controller, as well as part of the code executing in the FlexPendant is unmanaged.
JIT compiler	When compiling managed code, the compiler translates the source code into Microsoft Intermediate Language (MSIL), which is a CPU-independent set of instructions. Before code can be executed, MSIL must be converted to CPU-specific code, usually by a just-in-time (JIT) compiler.

Acronym	Definition
CAPI	Controller Application Programming Interface
CLR	Common Language Runtime
GUI	Graphical User Interface
MSDN	Microsoft Developer Network, source of information for .NET developers at: http://msdn2.microsoft.com/en-au/netframework/default.aspx
VS	Visual Studio
RAB	Robot Application Builder
SDK	Software Development Kit
VB	Visual Basic
TAF	Teach Pendant Application Framework, all applications using the FlexPendant SDK must run as TAF clients. See *TAF - Application host framework on page 52* for detailed information.
TCP/IP	Transmission Control Protocol (TCP) and Internet Protocol (IP)

2 Installation and development environment

2.1. Installation overview

About this section

This section describes how to install Robot Application Builder. When the installation is complete, you can program, compile and test PC and FlexPendant applications for the IRC5 controller.

Supported platforms

The following software requirements have to be met:

- Operating system: Microsoft Windows XP + SP2 or Windows Vista + SP1
- Microsoft Visual Studio: VS 2005 (Standard Edition or better is required to use the FP SDK) or VS 2008 (Professional Edition or better is required to use the FP SDK). To use the PC SDK Standard or Express edition will do.

The following hardware requirement have to be met:

- 50 MB free disc-space on the installation disc

Both FlexPendant generations are supported:

- SxTPU-1, which executes with .NET CF 2.0 and WinCE 4.2.
- SxTPU-2, which executes with .NET CF 2.0 and WinCE 5.0.

NOTE!

When installing RAB on Windows Vista OS you may get a Visual Studio error message saying "*The operation could not be completed. The requested operation requires elevation*". This error is due to failure of the installation of the FP SDK templates and project wizard, which is prevented when Vista's User Account Control (UAC) feature is enabled. To solve the problem you need to uninstall RAB, disable UAC, reinstall RAB, then re-enable UAC (if desired). It is now possible to use the Project Wizard and create a FlexPendant project.

NOTE!

Robot Application Builder is developed and tested for the English version of Visual Studio. If you are running Visual Studio in another language you are therefore recommended to switch to the English version.

Requirements for installing and using Robot Application Builder

To install and use Robot Application Builder, the following requirements have to be met. Also make sure that you have administrator permissions on the computer that you are using.

Before...	you must...
installing Robot Application Builder	install RobotStudio and Microsoft Visual Studio 2005 or 2008. **Note!** If you are running under windows Vista you need to disable Vista's User Account Control (UAC) feature.
debugging using a virtual IRC5	learn how to run the virtual IRC5 in RobotStudio.

Continues on next page

Continued

Before...	you must...
debugging using the real FlexPendant device	install the *.NET Compact Framework 2.0 Service Pack 1 or 2*, which can be downloaded from http://www.microsoft.com/downloads. The User Forum has information on how to attach the Visual Studio debugger to the device. See also *Debugging the FlexPendant device on page 286*.
executing the application targeting a real IRC5 system	check that the robot system has the controller option *PC Interface* (for PC applications) or *FlexPendant Interface* (for FlexPendant applications). set up a connection between your PC and the robot controller. See *How to set up your PC to robot communication on page 27* for details about how this is done.

NOTE!

The Visual Studio installation installs .NET and Compact Framework 2.0.

About the Robot Application Builder installation

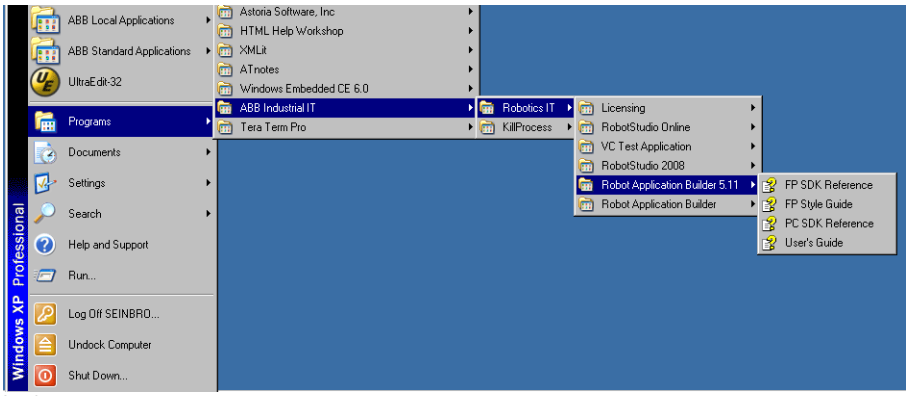
The RAB installation includes both PC and FlexPendant SDK. It is distributed as freeware on the RobotWare DVD. It can also be downloaded for free along with robotware and RobotStudio from http://www.abb.com/robotics > **RobotStudioCommunity** >To Download page >Developer Tools > Robot Application Builder.

For RAB 5.11, the installation was simplified. Before, any existing PC SDK was *upgraded* when a later RAB was installed. This is no longer the case; both SDK:s will now be installed side-by-side with any existing installation. This section describes the installation of RAB 5.11 and later.

RAB 5.11and later

RAB 5.11 and later installs PC SDK and FP SDK side by side with any previously installed versions. This makes it easier to work with several versions of the PC SDK on a single computer.

The figure below shows what it looks like when clicking the Window's Start button of a PC that has both RAB 5.10 and RAB 5.11 installed.



2.1_0

Continues on next page

Continued

RAB 5.10

RAB 5.10 upgraded any previously installed PC SDK to 5.10 and installed FlexPendant SDK 5.08, 5.09 and 5.10 side-by-side. The reason for the side-by-side installation of several FP SDK versions was to make it easier for FP SDK users to work on FP SDK applications targeting different RobotWare versions. Today, as the use of RAB is free of charge you can just download any version you need, and work with several versions on your PC if you need to. Earlier RAB releases can be downloaded from http://www.abb.com/robotics > **RobotStudioCommunity** >Developer Tools > Robot Application Builder Overview.

What is installed?

The installation generates the following features on your PC:

- SDK assemblies and resources
- This User's Guide (*Application manual - Robot Application Builder*)
- FlexPendant Style Guide
- PC SDK Reference
- FP SDK Reference

Working with several versions

A RAB application normally targets a specific RobotWare release. Assuming that you are developing a FP SDK application for a new customer, who will use RW 5.12 and at the same time you are maintaining an existing FP SDK application for a customer whose robot system uses RW 5.09. You will then need to work with two different RAB releases on your PC. See *Release upgrades and compatibility on page 60* for details about releases and compatibility.

FlexPendant applications

If you install RAB 5.10 and RAB 5.12 you will have FP SDK 5.08, 5.09, 5.10 and 5.12 on your PC. You choose which FP SDK version to use when you set up your application project in Visual Studio. See *Using the project template in Visual Studio on page 80* for detailed information.

You should make sure that the FP SDK GUI controls the Visual Studio Designer uses is of the same version. If you have worked with another FP SDK version before, you will need to remove the GUI controls that you have added to the Visual Studio Toolbox and then add them again, pointing to the correct version in the browser. See *Setting up design support for FlexPendant controls on page 83* for details.

From RAB 5.10 no license is required to develop, build or run RAB applications.To be able to use the 5.08 or 5.09 FP SDK versions of the 5.10 release, however, you will need to install a license key. RAB license keys are no longer available via the order form or SoFa, but ABB's software support organization can supply a license key at no charge if necessary.

PC applications

If you install RAB 5.10 and RAB 5.12, PC SDK 5.10 and 5.12 will exist on your PC. You choose which PC SDK version to use when adding PC SDK references to your application project in Visual Studio (browse to the installation directory that matches the version when adding the PC SDK references to the project). You should also set the Reference Property *Specific Version* to true to ensure that the correct version of the PC SDK dlls in the Global Assembly Cache (GAC) is used in run-time.

Continues on next page

Installation procedure

The installation procedure is very simple. An installation wizard will guide you through it.

NOTE!

Click Window's Start button and locate the Robot Application Builder folder when the installation is ready (Start >Programs >ABB Industrial IT > Robotics IT >Robot Application Builder 5.XX). This is where you find the user documentation: User's guide, FP Style Guide and the PC and FP SDK References.

You are also strongly advised to study the Release Notes that you will find on the RW DVD and on the web, as these hold the most up-to-date information, including new features and any known limitations of the release.

2.2. How to obtain and install a license key for RAB 5.09 or earlier

Overview

In RAB 5.10 the license check was removed from the software, which allows anyone to use Robot Application Builder for free. This means you do no longer need to bother about getting a license, or including a licx file in your PC application.

NOTE!

For RAB version 5.09 and earlier, licensing is the second part of the installation procedure. In case you need to develop a RAB application for RW 5.09 or earlier you need to turn to support to get a free license key.

Install licence key

Follow these steps when you have received the e-mail with the license key file:

Step	Action
1	Detach the license key file from the e-mail and save it to a folder on your PC.
2	Double-click the license key file. This opens the **License Install Wizard**.
3	Follow the instructions in the wizard.

NOTE!

To execute RAB applications towards a real robot controller you must connect your PC to the robot controller, either via a network or directly to the service port on the controller. For detailed information, see *How to set up your PC to robot communication on page 27*.

2.3. How to set up your PC to robot communication

Overview

This section describes how to connect your PC to the robot controller.

You can either connect the PC to the controller via an Ethernet network or directly to the controller service port. When using the controller service port, you can either obtain an IP address for the PC automatically, or you can specify a fixed IP address.

When the PC and the controller are connected correctly, the controller is automatically detected by RobotStudio.

NOTE!

A PC SDK application requires RobotStudio or ABB Robot Communications Runtime to connect to a controller in run-time. The latter is included in the RAB installation. If RobotStudio is not (and will not be) installed on your PC, you must run the Setup.exe located at C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder 5.xx\redistributable\RobotCommunicationRuntime.

Why is a connection needed?

Connecting the PC to the controller is necessary for all online tasks performed in RobotStudio: downloading a robot system or files to the controller, editing configuration files, programming and so on.

It is necessary for executing a RAB PC application targeting a real robot controller.

The connection is also used for downloading a FlexPendant application to the controller file system and test it on the real FlexPendant device.

It also enables you to communicate with the controller by means of a console window on the PC and get valuable information about controller status, FlexPendant memory consumption and the like.

Ethernet network connection

If the controller is connected to an Ethernet network, you can connect the PC to that network as well. The settings to use on the PC depends on the network configuration. To find out how to set up your PC, contact the network administrator.

Continues on next page

Continued

Service port connection with automatic IP address

An alternative to network connection is using the controller service port. It has a DHCP server that automatically gives your PC an IP address if it is configured for this. See *Windows Help* on *Configure TCP/IP settings* for detailed information about configuring the PC to obtain an IP address automatically.

NOTE!

Obtaining an IP address automatically might fail if the PC already has an IP address from another controller or Ethernet device. To make sure that you get a correct IP address if the PC has already been connected to an Ethernet device, do one of the following:

- Restart the PC before connecting to the controller.
- Run the command “ipconfig /renew” from the command prompt after connecting the PC to the controller

Service port connection with fixed IP address

Instead of obtaining an IP address automatically, you can specify a fixed IP address on the PC you connect to the controller.

Use the following settings for connecting with a fixed IP address:

Property	Value
IP address	192.168.125.2
Subnet mask	255.255.255.0
Default Gateway	192.168.125.1

Related information

For information about	See
How to set up PC network connections	*Windows Help - Configure TCP/IP settings*.
How to connect the PC to the Controller service port	*Connect a PC to the Service Port* in the RobotStudio help.

2.4. Development environment

Overview

This section presents an overview of the development environment used to create RAB applications for PC or the FlexPendant. In either case, you program and debug the application using Microsoft Visual Studio 2005 or 2008.

Microsoft .NET and Microsoft Visual Studio

Microsoft Visual Studio is supported by the .NET Framework. A core component of the .NET Framework is the common language runtime (CLR). It manages code execution, threads and memory, while also enforcing type safety.

Another major component is the Base Class Library, which is a comprehensive, object-oriented collection of reusable types. To become a skilled .NET programmer it is essential to learn the functionality offered by the Base Class Library.

It is not in the scope of this manual to teach how to use Visual Studio. For this purpose Msdn (Microsoft Developer Network) at http://msdn.microsoft.com is a useful source of information.

NOTE!

From RAB 5.11 Visual Studio 2008 is also supported. See *Conversion of VS 2005 projects to Visual Studio 2008 on page 33* for information about upgrading an existing RAB project to Visual Studio 2008.

Visual design support and data binding

The most significant improvement of Robot Application Builder with Visual Studio 2005 was the visual design support for FlexPendant applications. Thanks to enhanced abilities of the .NET Compact Framework 2.0. for building user interfaces, FlexPendant specific controls are available in the Visual Studio toolbox since RAB 5.08.

Another very useful feature of .NET CF 2.0 is data binding, which allows you to connect a `RapidDataBindingSource` or a `SignalBindingSource` to a GUI control without having to write a single line of code. (Except the `Dispose` call when the binding sources are no longer needed.)

Continues on next page

Continued

Choosing a programming language

Together with Visual Basic, C# is the most widely used.NET language.

C# is an object-oriented language derived from C, with some features from C++, Java and Visual Basic. It was designed for .NET and offers the power and richness of C++ along with the productivity of Visual Basic. Both PC and FlexPendant SDK are implemented using C#.

As for FlexPendant SDK applications only C# and Visual Basic are supported. As for PC SDK applications, on the other hand, any of the .NET languages can be used. ABB support, however, is offered only in C# and Visual Basic. Likewise, in this manual there are code samples in C# and Visual Basic, but none in J# or Visual C++.

At run-time it does not matter which language you have used, as compiled .NET code is language independent. The source code compiles from a high-level language into Intermediate Language (IL), which is then executed, at runtime, by the Common Language Runtime. This makes it possible to use different programming languages, even within the same application. See *Definitions on page 18* for further explanation of .NET terms.

NOTE!

It is presumed that you are already a .NET programmer. If not, you need to start by learning the programming language to be used. There are numerous books teaching C# and Visual Basic.

Integration with Visual Studio

When Robot Application Builder is installed on your computer, it is integrated with Visual Studio. You will notice when starting a new project, for example, that the project type */Smart Device/FlexPendant* is available in the **New Project** window. When using the wizard to create a FlexPendant project, common SDK references are added to the project and some code is auto generated.

The visual design support for the FlexPendant will be accessible from the **Toolbox** in VS and work the same way as the design support for an ordinary Windows application. As for a PC application you use the standard design support. As you will see, using RAB is quite intuitive for a developer used to Visual Studio programming.

NOTE!

The help module is not integrated with the Visual Studio Help function. Pressing F1 when pointing at code, for example, will open the *Visual Studio Programmer's Reference* or the *.NET Framework Class Library* for the specific language and topic. If your problem is RAB-related this will not help you.

TIP!

Depending on what kind of application you are working at, locate the *FP SDK* or *PC SDK Reference*. You will find it by clicking Windows' **Start** button, then pointing at Programs > ABB Industrial IT > Robotics IT > Robot Application Builder 5.xx. Keep the reference file open while programming, as you will be needing it all the time.

2.5. Two development models - virtual and real

About this section

When trying out a custom application, you can either use a virtual robot controller or a real robot system. This section provides information on how to use both development models.

Virtual robot technology

The virtual IRC5 of ABB's RobotStudio allows the IRC5 controller software to execute on a PC, and supports RAB application developers with a purely virtual environment to be used for development, test and debug.

When you start the virtual IRC5 in RobotStudio, a virtual robot cabinet along with a virtual FlexPendant will appear on the PC screen.

As a real robot controller is normally not readily at hand for application development, virtual technology is very valuable.

Requirements for virtual environment

The following software components must be installed to develop, test and debug using the virtual environment:

- ABB RobotStudio (including the virtual IRC5 and RobotStudio Online)
- ABB Robot Application Builder
- Microsoft Visual Studio 2005 or 2008

Controller option PC Interface or FlexPendant Interface may not be needed in the virtual environment.

Requirements for real environment

The following software components must be installed to develop, test and debug using a real robot controller:

- ABB RobotStudio (RobotStudio Online is needed to create the robot system)
- ABB Robot Application Builder
- Microsoft Visual Studio 2005 or 2008
- Controller option PC Interface or FlexPendant Interface
- Network connection between PC and robot controller

For information about how to set up the network, see *How to set up your PC to robot communication on page 27*.

Virtual test and debug

Using the virtual environment a FlexPendant application executes on the Virtual FlexPendant as an assembly (dll). You start the application from the ABB menu of the Virtual FlexPendant just like you start it on the real FlexPendant.

A PC application, on the other hand, will run as an independent executable (exe). Using the virtual environment it targets the virtual IRC5 instead of a real robot controller.

Debugging is easy using the virtual IRC5 and Visual Studio. You attach the application process to Visual Studio, set a break point in the code and step through it as it executes. See *Debugging the virtual FlexPendant on page 282* for further information.

Continues on next page

Continued

Real tests necessary

The virtual environment is a very convenient choice, especially for testing and debugging. You should be aware, however, that the virtual FlexPendant is more forgiving than the real device. Using only the virtual FlexPendant, it is very easy to neglect the restraints on memory consumption imposed by the real device. Images, for example, can easily consume all the FlexPendant memory available for custom applications!

This means that potential problems may be hard to detect until you test the application using a real robot system. It is almost as easy to debug code running on the real FlexPendant device. See *Debugging the FlexPendant device on page 286* for detailed information.

You should also be aware that your application shares CPU, memory and application host with all other FlexPendant applications. This means that a custom application can impact the overall performance of the FlexPendant.

NOTE!

Before shipping a FlexPendant application, it has to be tested properly, using a real system. Relying only on the virtual environment is far too risky. Also study the chapter *Robust FlexPendant applications on page 181* carefully.

Porting the application from virtual to real IRC5

As for a PC SDK application, you will hardly notice any difference when using it with a real IRC5 controller. The only real difference is that the communication between the application and the controller will now be done over a network, which may have an impact on performance.

A FlexPendant application, on the other hand, may run perfectly on the Virtual FlexPendant, but not work at all on the real device. There may be a lag in response time due to TCP/IP communication, but the main problem is limited resources on the device, both memory and processor power.

The FlexPendant SDK does not slow down performance. Therefore your application is supposed to perform like any standard application of the FlexPendant. See *Performance on page 188* for some advice on how to speed up a slow FlexPendant application.

Deployment to customer

During development, deployment to the controller is done manually. When the development phase is over and the application needs to be deployed to the customer, this should be done differently.

For information about how this should be done, see *Packaging RAB applications on page 309*.

2.6. Conversion of VS 2005 projects to Visual Studio 2008

Overview

Converting an existing RAB Visual Studio 2005 project to Visual Studio 2008 is simple. When you open a VS 2005 project in VS 2008, the Visual Studio Conversion Wizard will automatically appear. The procedure which converts the project to VS 2008 is easy to follow. It consists of a few dialogs providing information about what will happen.

NOTE!

If the RAB project is an FP SDK projects, you need to manually edit the post-build event that builds the *gtpu.dll. Find *Build Events* in the *Project Properties* and adapt the path to *vcvarsall.bat* to the new development environment.

2.6. Conversion of VS 2005 projects to Visual Studio 2008

3 Run-time environment

3.1. Two platforms - PC and FlexPendant

About this chapter

There are two different platforms that a custom application may use: PC and FlexPendant. Accordingly, RAB consists of two platform dependent SDKs the PC SDK and the FlexPendant SDK.

This chapter provides an overview of the run-time environment of custom applications, including illustrations of the software architecture of the FlexPendant and PC SDK respectively.

How communication is carried out between the client and the robot controllers is explained, as well as how clients access controller resources.

Application configuration is detailed as well as deployment of a FlexPendant application to a robot controller. The life cycle of a FlexPendant application is also explained.

Most of the contents of this chapter is separated in PC and FlexPendant specific sections, as their respective run-time environments considerably differ.

Selecting the platform your application should use

Before you start developing a custom application, you must know which platform is best suited for meeting the needs of the customer. The platform choice depends on what tasks should be performed using the application, whether these tasks must be done at the robot cell and whether there is a PC at hand where the job needs to be done.

If the application should be used at the robot cell and there is no PC available, there evidently is no choice. It has to be a FlexPendant application. On the other hand, if the tasks can be done from a PC this is often a convenient platform, provided that the PC SDK offers the required functionality.

Continues on next page

Continued

Local vs remote client

One difference between the two platforms is that a FlexPendant application is a local client, whereas a PC application is a remote client.

Remote clients do not have all the privileges of a local client. Both PC and FP applications can reset the program pointer and start RAPID execution, for example, but for a PC SDK application to do this there are certain restrictions. Mastership of the Rapid domain must be requested explicitly by the application programmer and the IRC5 controller has to be in automatic operating mode.

An advantage of a remote client, on the other hand, is the possibility to monitor and access several robot controllers from one location. As for large applications the PC platform is also less limited than the FlexPendant as regards memory resources and process power.

NOTE!

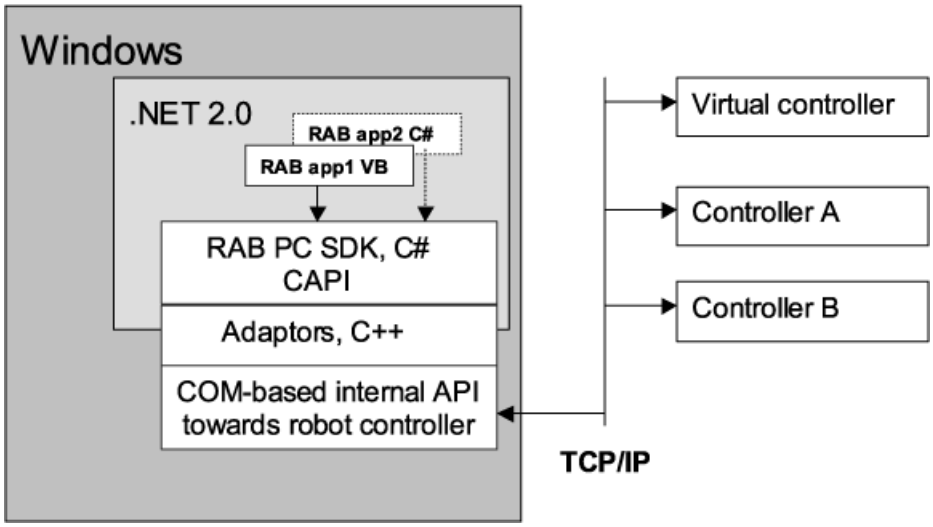
A minimum response time for a real controller should be expected to be in the order of 10-100 milliseconds, meaning that hard real time demands cannot be met on any platform. See *Communication between PC and controller on page 46* and *Communication between FlexPendant and controller on page 51* for further information.

Software architecture

The FlexPendant is an integral part of IRC5, yet a complete computer in itself. It has been developed with Microsoft's latest software technology for embedded systems, Windows CE and .NET Compact Framework, which is a subset of the full .NET Framework that the PC uses.

Simple illustrations of the software architecture of the two SDKs are shown below. As you can see the two run-time platforms, PC and FlexPendant, have a lot in common.

PC platform



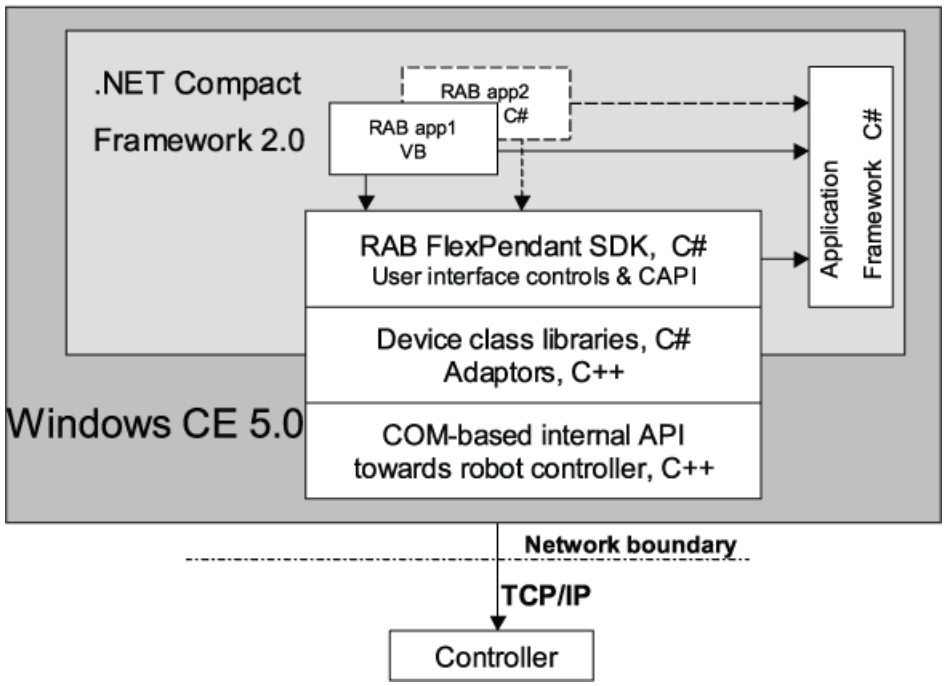
3.1_1

Figure 1 - PC platform. Two PC SDK applications developed on top of the PC SDK. The PC SDK CAPI is the public API offering controller functionality. A PC SDK application can control many robot controllers on the network. All communication with these is done via the internal RobApi.

Continues on next page

Continued

FlexPendant platform



3.1_2

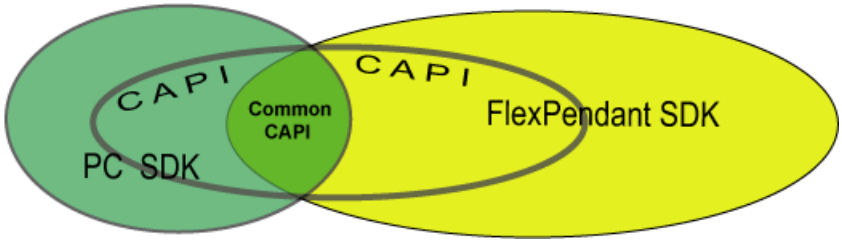
Figure 2 - FlexPendant (SxTPU-2) platform. Two FlexPendant applications, one using VB and the other C#, developed on top of the FlexPendant SDK. The FlexPendant SDK CAPI is the public API offering controller functionality. All communication with the robot controller is done via the internal RobApi. The FlexPendant SDK provides ABB made UI controls suitable for the FlexPendant.

CAPI

Both SDKs offer controller functionality through a public application interface called CAPI. The interface can be seen as a specification of the controller services available.

Part of the controller services provided by the PC and FlexPendant SDK are the same. This means that part of the specification is also common (identical method signatures). The internal implementation of these controller services may differ, but for an external client, such as a RAB application, these methods are platform independent and work for both PC and FlexPendant applications.

As illustrated below, however, only a small part of the PC and FlexPendant SDK is common. Therefore, you cannot port a PC application to the FlexPendant platform or vice versa.



3.1_3

Continues on next page

Continued

Figure 3 - Common CAPI methods can be used for both platforms. The other CAPI methods are platform dependent. As you can see the FlexPendant SDK is more extensive than the PC SDK. This is mainly due to the fact that the FlexPendant SDK offers its own GUI controls. In this figure these would be in the FlexPendant circle outside of the CAPI circle.

3.2 Running PC Applications

3.2.1. Licence verification - applies only to versions earlier than RAB 5.10

Overview

A PC SDK application runs as a .NET executable, started either by double clicking the exe-file or by browsing to the program using the Windows **Start** menu.

Deployed PC applications do license verification during execution, checking that all application assemblies have been built on a PC with a valid PC SDK license key. If the key is missing some functions in the PC SDK will raise an exception during execution.

NOTE!

The license verification was removed in the 5.10 release. So the licx file detailed in the next section is no longer needed.

Licenses.licx

The license key should be placed in a “Licenses.licx” file, which should be added to your project as an embedded resource. For your convenience, such a file is included in the RAB installation at C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder\PC SDK. The key for the PC SDK in VS 2005 is:

```
ABB.Robotics.Controllers.Licenses.PCSdk, ABB.Robotics.Controllers
```

TIP!

The License Compiler (Lc.exe) is a .Net Framework tool. It generates a .license file from a .licx file. Search in MSDN (licx , lc.exe) if you want more detailed information.

3.2.2. Mastership

Controlling controller resources

Controller resources must be controlled by a single client at a time. Several people can be logged on to the controller, but only one person at a time can run commands or change RAPID data. This is for security reasons as well as for protecting data from being accidentally overwritten.

When logged on to a controller you can have either read-only access or write access. Read only is the default access right. To get write access the client needs to request mastership of the controller resource it wants to manipulate.

NOTE!

In addition to the access right system, there is the User Authorization System, which restricts what each user is allowed to do with the robot. See *User Authorization System on page 70* for further information.

Manual and automatic mode

When the controller is in manual mode, the FlexPendant has priority to write access, and mastership will not be given to a remote client unless an operator explicitly allows this via the FlexPendant. At any time, the operator can press *Revoke* on the FlexPendant to get the write access back.

In automatic mode, the client who first requests write access will get it. If a remote client has taken mastership of a domain other remote clients will not be allowed write access, but will get an exception if they try. For the operator there is no way to revoke mastership to the FlexPendant, but to switch the operating mode of the controller to manual.

As for FlexPendant SDK applications, requesting and releasing mastership is handled internally by the FlexPendant SDK, and the application programmer does not need to worry about it.

As for a remote client, such as a PC SDK application, however, it has to be carefully implemented by the application programmer.

PC SDK mastership domains

The PC SDK domains which require mastership are:

- Rapid
- Configuration

Continues on next page

Continued

For code examples see *Start program execution on page 208* in the PC SDK section.

NOTE!

Operations that require mastership are more resource demanding. Mastership should therefore be released as soon as an operation is completed.

Remote privilege in manual mode

Most of the time, it is not very convenient to have a PC SDK application perform operations that require mastership when the controller is in manual mode. Starting program execution for example is not even possible.

In manual mode when a remote client, e.g. RobotStudio or a PC SDK application, requests mastership a dialog box will appear on the FlexPendant. It enables the operator to grant mastership to the requesting client.

If mastership is granted, the remote application has the privilege to access robot controller resources. Meanwhile, the FlexPendant is locked and cannot be used until the remote application releases mastership or mastership is lost for any of the reasons mentioned in *Losing mastership on page 42*.

Losing mastership

Remote clients have to be prepared that mastership may be lost without a warning for a number of reasons. Among these are:

- change from automatic to manual mode
- controller restart
- lost communication
- in manual mode forced revocation of mastership by another client with higher priority - for example the FlexPendant

If mastership is lost, it has to be taken back explicitly by the client. The controller does not store the information.

NOTE!

The FlexPendant may also lose mastership without any warning. This may happen in automatic mode, when a RobotStudio user or a PC SDK application asks for write access to the controller, for example. The status bar of the FlexPendant will then indicate *“Remote Access in Auto”*.

3.2.3. PC application configuration

Application configuration file

All .NET Winform applications are designed to read configuration from an *App.config* file in the application directory. It is not mandatory to use such a file in a PC SDK application, but it is sometimes a handy way to add application flexibility.

For your convenience an *App.config* file that you can use is included in the RAB installation. The default values, which the PC SDK uses if there is no configuration file to read, are entered in the file. To modify application behavior you thus need to change the values of the attributes of this file.

NOTE!

Even if you use the App.config file to specify which controllers to work with you must still use the netscan functionality to be able to establish a connection from your PC application. See *Discovery domain on page 210* for further information.

Add App.config to the project

Start by copying the *App.config* file at C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder 5.xx\PC SDK to the directory of your .csproj file. Then add the file to the project by right-clicking the project icon, pointing to **Add** and selecting **Existing Item**.

NOTE!

The *Copy Local* property of the PC SDK references used by your application must be set to true to make use of the App.config file. (In the *Solution Explorer* in Visual Studio, right-click the reference and select *Properties*.)

Section tag

The `<section>` tag in the `<configSection>` part of the *App.config* should specify that there is a *capi* section in the file. It should also specify which type is to be used as the configuration container object as well as the assembly that this type belongs to:

```
<section name="capi"
     type="ABB.Robotics.Controllers.ConfigurationHandler,
     ABB.Robotics.Controllers"/>
```

Capi section

The PC SDK application specific configuration data should be added to the `<capi>` section.

```
<capi>
  ...
</capi>
```


Continues on next page

Continued

The following tags are implemented in the PC SDK:

<defaultSystem>

If there is a controller (robot system) in the network that you connect to often, you may want to use the `<defaultSystem>` tag. It has an `id` attribute containing a string embraced by curly brackets. It is the system's globally unique identifier (GUID), which can be find in the *system.guid* file in the INTERNAL folder of the robot system.

```
<defaultSystem id="{469F56DF-938E-4B06-B036-AABBB3E61F83}" />
```

Using this mechanism enables you to use the default constructor to create a `Controller` object for the specified controller:

```
VB:
Dim aController As Controller = New Controller()

C#:
Controller aController = new Controller();
```

<remoteControllers>

It is possible to add controllers outside the local network when scanning for available controllers. One way of doing that is to add the IP address of these controllers in the `<remoteControllers>` tag:

```
<remoteControllers><controller id="192.168.0.9" />
<controller id="192.168.0.19" />
</remoteControllers>
```

<discovery.networkscanner>

You can configure how long (in ms) a scan operation will last, and increase the value if netscanning fails. The default value is 200, but if you have a slow PC longer time might be needed.

```
<discovery.networkscanner delay="400" />
```

<defaultUser>

The `<defaultUser>` tag holds information about user name, password and an optional application name for the default user. It is used by the `UserInfo` class to log on to a controller. If an application name is not supplied, the process name is used.

```
<defaultUser name="user name" password="password"
     application="application"/>
```

<rmmp>

When mastership is requested in manual mode by a remote client such as RobotStudio or a PC SDK application, a dialog is launched on the FlexPendant asking the operator to confirm that mastership should be passed from the FlexPendant to a remote client. As long as there is no confirmation on the FlexPendant the PC SDK application is not given mastership. The time-out parameter is the time in seconds that the PC SDK application will wait for someone to confirm remote access in the FlexPendant dialog. The cycle parameter is the time in ms between poll calls from the PC SDK to check whether mastership has been granted.

```
<rmmp cycle="550" timeout="65" />
```

Continues on next page

Continued

<controllerCall>

You can add a time-out in ms and a multiplicand for slow calls to the controller. The time-out parameter is the maximum time a call through the Controller API will be permitted. If no answer is returned within the time specified, an exception is thrown. A slow call is a call that takes longer than usual, usually operations which require a UAS grant:

```
<controllerCall timeout="27000" slow="2.1" />
```

<eventStrategy>

The default way to handle events from the controller is to use asynchronous delegates (`AsyncDelegate`), applying the `Invoke` method to synchronize events and GUI.

By using an `<eventStrategy>` tag, you can choose to use a windows postback delegate instead. To make this work you must also implement a subscription to the event from a windows form, or else the event handler will not receive the event:

```
<eventStrategy name="WindowDelegate" />
```

NOTE!

Using this strategy for event handling may affect the performance of your application.

3.2.4. Communication between PC and controller

COM technique

The PC SDK uses an internal Controller API based on COM technique to communicate with the controller. This API uses sockets and the local TCP/IP stack (see *Definitions on page 18*) towards both real and virtual controllers.

NOTE!

You should be aware that the .NET garbage collector does not collect COM objects, but these need to be disposed of explicitly by the application programmer. See *Memory management in PC applications on page 213* for further information.

Resource identification

All controller resources, both real and virtual, are described using object based hierarchy. Each resource is uniquely identified, including information about which controller owns the resource by use of the unique system id or IP address of the controller.

The controller is the top object of the hierarchy:

```
"/<Controller>/<Domain>/<SubDomain1>/<SubDomain2>/etc"
```

TIP!

Error messages including such a path indicate where to look for the problem.

Hard real-time demands

You should be aware that the PC CAPI cannot meet hard real-time demands. This is for several reasons:

- Part of the API executes on a non-real-time operating system.
- Communication is performed with TCP/IP over a network
- The controller sometimes has tasks to perform that have a higher right of priority.

NOTE!

A minimum response time for real controllers should be expected to be in the order of 10-100 milliseconds.

3.3 Running FlexPendant Applications

3.3.1. Components, assemblies and dlls

Building blocks

The .NET Framework is a library of components and supporting classes and structures, designed to make component development easy and robust. Components are packaged in assemblies, also called dlls.

Assemblies are the building blocks of .NET applications. An assembly is a reusable collection of types and resources, which are built to work together and form a logical unit of functionality. The simplest assembly is a single executable.

One or several assemblies

A FlexPendant project compiles to a dll, which cannot run as an independent application, but needs to be started by the Teach Pendant Application Framework (TAF), the application manager of the FlexPendant. (See *Understanding FlexPendant application life cycle on page 52* for further information about how this works.)

In a normal case, a custom application for the FlexPendant is developed as a single component, but it is also possible to separate functionality into several components. This way the application will consist of several dlls. The reason for doing so might be one of the following:

- The amount of code is substantial. A modular design with small and well tested building blocks put together is one way of handling complexity.
- Different developers are working on the same custom application. For reasons of efficiency, they can split the functionality between them and work on one component each.

NOTE!

It is possible to use different programming languages for different components.

3.3.2. Deployment of FlexPendant application to a robot system

Proxy assembly

When you compile a FlexPendant SDK application an additional assembly named *gtpu.dll is automatically created. This is done by a tool, the *ABB Compliance Tool*, which verifies that the application complies to the FlexPendant requirements. This proxy dll is necessary to run the application on the FlexPendant.

To test the application on a real FlexPendant both assemblies must be downloaded to the SYSTEM or HOME directory of the robot controller. After this the FlexPendant should be restarted. At startup it loads the application assemblies from the controller.

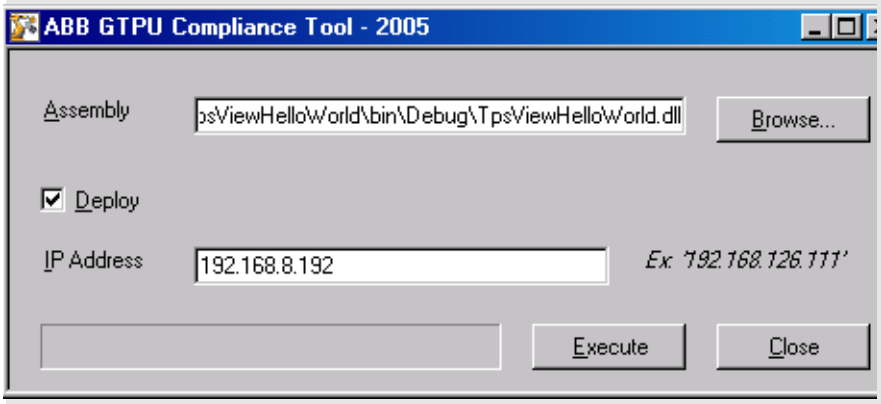
TIP!

An advantage of deploying the dlls to the HOME directory is that they will be included in a system backup.

Download to real controller

To download your application assemblies to the controller you can use the *Online* function *File Transfer* of RobotStudio.

Another way of downloading the application to the robot controller is using the *ABB Compliance Tool*.

Step	Action
1	Verify that the following requirements are met before starting the procedure: • A network connection between the PC and the controller has to be configured, either using the controller LAN port or the controller service port. For further information see *How to set up your PC to robot communication on page 27*. • The RobotWare option *FlexPendant Interface* is required to run the application on a real system. Without the option you will not see the application in the ABB menu.
2	Open Windows Explorer.
3	Start abbct.exe at C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder 5.xx\FlexPendant SDK.  4.3.2_1
4	Click **Browse** and locate your Visual Studio project and the application assembly in the \bin\debug (or \bin\release) sub-folder.

Continues on next page

Continued

Step	Action
5	Check the **Deploy** box, enter the **IP Address** of the robot controller and press **Execute**. Deployment is done to the current system of the controller. **Note!** If the application consists of several assemblies you need to repeat the procedure for all of these.
6	Restart the FlexPendant. See *Restart the FlexPendant on page 49* for information about different ways to do this.

Using the command window

The *ABB Compliance Tool* can be used via the command window instead of the graphical interface. To do so you write:abbct.exe /deploy="192.168.8.192" <PATH>\TpsViewHelloWorld.dll <PATH>\TpsViewHelloWorld.gtpu.dll.

It is also possible to perform build and deployment in one step. To do this the deploy argument should come last:

```
abbct.exe <PATH>\TpsViewHelloWorld.dll /deploy="192.168.8.192"
```

Both the application and proxy assembly are deployed to the controller after the build.

FTP deployment

You can also use an FTP client to transfer files from your PC to the robot controller.

To use FTP you need:

- FTP client program
- Configured connection to the controller
- RobotWare option *FTP Client*
- Name and password

Step	Action
1	Transfer resources (e.g. icons) and the application and proxy assemblies to the *HOME* or *SYSTEM* directory of the current system of the controller.
2	Restart the FlexPendant. See the next section for information on how to do so.

NOTE!

For deployment to a customer when the application is ready see *Deployment of a FlexPendant SDK application on page 311*.

Restart the FlexPendant

If you want to restart the FlexPendant device without restarting the controller, choose one of these alternatives:

Alternative	Action
1	Write the command `fpcmd "-restart"` in the controller console window on your PC.
2	Perform the following sequence while holding the FlexPendant joystick: Move the joystick three times to the right, once to the left and once down. Confirm your wish to reset the FlexPendant in the dialog that will appear.

Continues on next page

Continued

Alternative	Action
3	Unplug and plug the FlexPendant (power on/ power off). **Note**! This activates emergency stop.

Deploy application to virtual IRC5

Follow these steps to deploy an application to the virtual FlexPendant:

Step	Action
1	Copy application and proxy assemblies and images to the HOME directory of the system you want to use on your PC. **Note!** Close the virtual FlexPendant first if you are *replacing* assemblies.
2	Restart the virtual FlexPendant.

TIP!

If you have problems running your application, try to put all files (dlls, gif files etc.) in the vcbin directory of the robotware your system uses. This setup is as close to the real system setup as you can get.

3.3.3. Communication between FlexPendant and controller

COM technique

The FlexPendant SDK uses an internal Controller API based on COM technique to communicate with the controller. This API uses sockets and TCP/IP (see *About terms and acronyms on page 18*) towards both real and virtual controllers.

Calls from the FlexPendant SDK to the controller are synchronous, i.e. are done immediately through the COM servers. This increases execution speed and causes less overhead, which is important on a resource limited device.

Resource identification

All controller resources, both real and virtual, are described using object based hierarchy. Each resource is uniquely identified, including information about which controller owns the resource by use of the unique system id or IP address of the controller.

The controller is the top object of the hierarchy:

```
"/<Controller>/<Domain>/<SubDomain1>/<SubDomain2>/etc"
```

TIP!

Error messages including such a path indicate where to look for the problem.

Hard real-time demands

The FlexPendant SDK cannot meet hard real-time demands. This is for several reasons:

- Part of the API executes on a non-real-time operating system.
- The controller sometimes has tasks to perform that have a higher right of priority.

NOTE!

The FlexPendant SDK does not affect performance in a negative way. You should expect your custom application to perform like any other application on the ABB menu.

3.3.4. Understanding FlexPendant application life cycle

Overview

Understanding the FlexPendant application life cycle improves your ability to design and debug the application.

TAF - Application host framework

The Teach Pendant Application Framework (TAF) is the application service provider that runs on the FlexPendant. It targets .NET CF and is implemented in C#. All applications using the FlexPendant SDK must run as TAF clients, as TAF contains services for hosting controls and for managing applications.

TAF uses a customized configuration file to create the appearance and behavior of the hosted application. It also defines a set of rules that have to be followed.

Starting a custom application

When the FlexPendant starts up TAF is already in the flash memory. Applications that will execute in the TAF container are now loaded from the controller.

If the RAB application is to be started manually by the end-user the application icon and text are placed in the ABB Menu. The other alternative is to have it started automatically by TAF at FlexPendant startup. See *FlexPendant TpsView attribute on page 54* to learn how this is configured.

Application life cycle

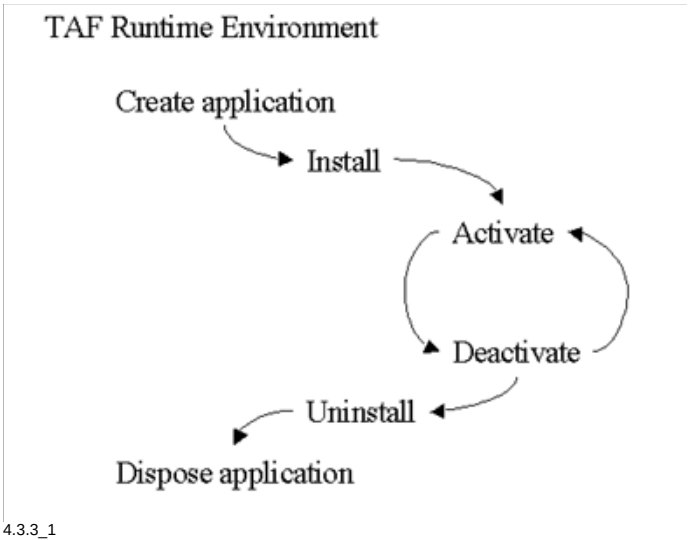
TAF handles the life cycle of a custom application, starting by calling the constructor of its TpsView class. After this, the `Install` method and then the `Activate` method in the same class execute.

During its lifetime, the application switches between the active and the passive state. Each time, either `Activate` or `Deactivate` is executed. In its active state the application is visible in the client view, in the passive state another application may have been opened or the user may have opened another application via the FlexPendant task bar.

When the application is closed via the close button, first the `Deactivate` method runs and then `Uninstall`. After this the `Dispose` method of the TpsView class is called. Then the application instance is disposed of by TAF. See *ITpsViewSetup and ITpsViewActivation on page 58* for further information about how you can implement these methods.

Continues on next page

Illustration



4.3.3_1

The figure illustrates the life cycle of a FlexPendant application.

Limited resources

As the FlexPendant is a device with very limited resources compared to a PC, you should learn and use the mechanisms implemented to assist you in writing efficient code.

Both process power and memory resources are limited compared to the virtual environment on the desktop. An application may run very well on the virtual FlexPendant, but encounter serious problems in the real environment because of these limitations. See *Technical overview of the FlexPendant device on page 182* for further information.

NOTE!

You are strongly recommended to read *Robust FlexPendant applications on page 181* in this manual before starting coding. Always keep the limitations of the device in mind when you develop custom FlexPendant applications.

3.3.5. FlexPendant TpsView attribute

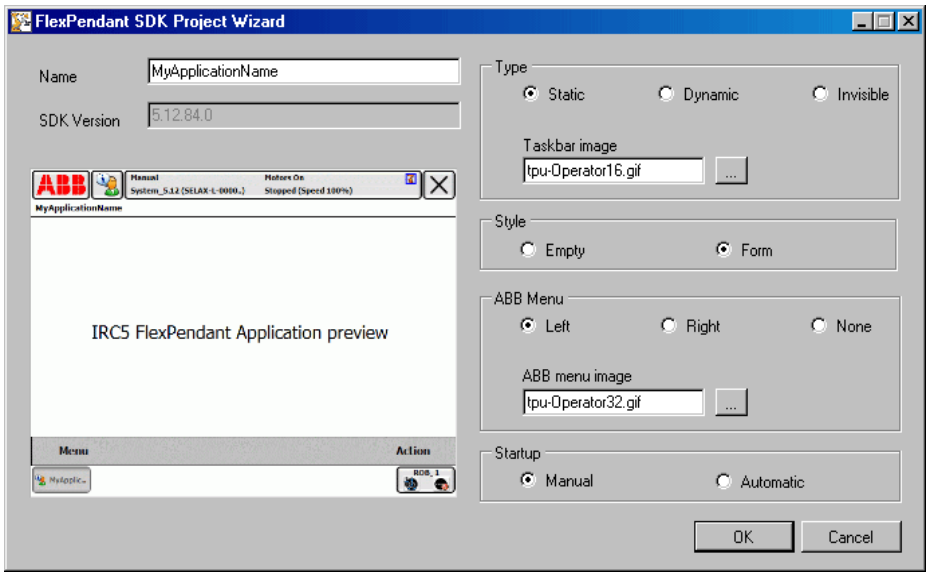
Overview

A custom FlexPendant application must implement the assembly attribute `TpsView`. It is used to determine the visual appearance of the application in the ABB menu, for example. The `TpsView` attribute is auto generated and located before the namespace definition in the application *view* class, i.e. the class that displays the first view of a FlexPendant SDK application.

In this section all parameters of the `TpsView` attribute will be detailed.

Project wizard settings

The `TpsView` attribute is auto generated according to your settings in the **FlexPendant SDK Project Wizard** in Visual Studio:



6.2.2_1

In C# the settings of the figure will render this code:

```
[assembly: TpsView("MyApplicationName", "tpu-Operator32.gif",
     "tpu-Operator16.gif", "TpsViewHelloWorld.dll",
     "TpsViewHelloWorld.TpsViewHelloWorld",
     StartPanelLocation.Left, TpsViewType.Static, StartupType =
     TpsViewStartupTypes.Manual)]
```


Continues on next page

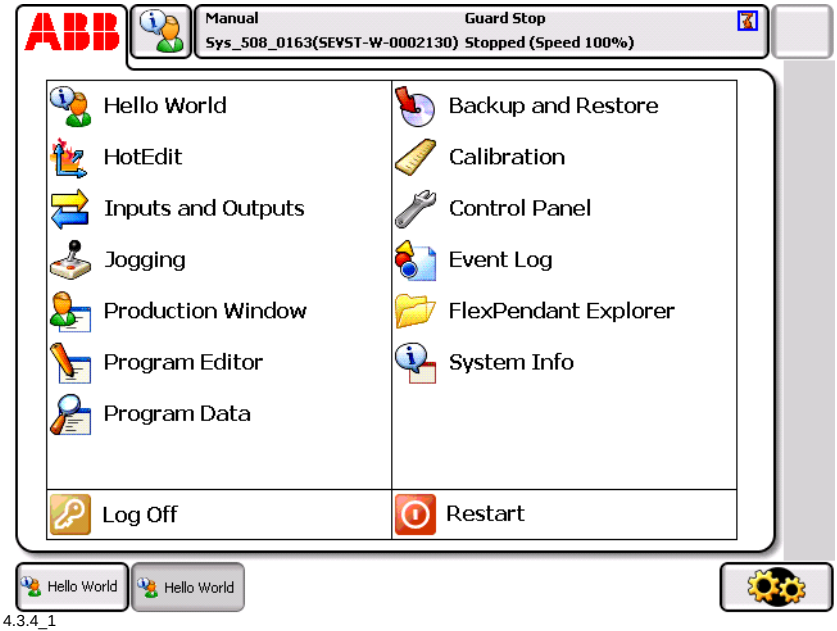
Continued

NOTE!

You can edit your settings directly in the auto generated code if you want to.

Visual appearance

In run-time the most obvious result of this code is the way the custom application is presented in the ABB menu. In this example the first `TpsView` parameter has been changed from “MyApplicationName” to “Hello World”.



4.3.4_1

Application name

The first parameter of TpsView is the application name as it should appear in the ABB menu and on the task bar. The example uses `"HelloWorld"` as the application name.

Application icon

The second parameter is the file to be used as the application icon in the ABB menu. Usually a customized icon is used for the application.The example uses the default icon: “`tpu-Operator32.gif`".

TaskBar icon

The third parameter is the file to be used as the application task bar icon. The example uses the default icon `"tpu-Operator16.gif"`.

Continues on next page

Continued

Application assembly

The fourth parameter is the assembly name. If you change the name of the application assembly you also need to change this parameter to reflect that change. In the example, the assembly name is `"TpsViewHelloWorld.dll"`.

NOTE!

The assembly name must start with *"TpsView"* for TAF to identify it as an application to be loaded. If you forget this an error will be generated by the ABB verification tool when you try to build the project.

Class name

The fifth parameter specifies the fully qualified class name of the initial view of your application, which you chose in the **New Project** dialog box. In the example, the fully qualified class name is `"TpsViewHelloWorld.TpsViewHelloWorld"`.

Application location

The sixth parameter determines the location of the application icon and text in the ABB menu. In the example these are displayed in the left column: `startPanelLocation.Left`.

Parameter value	Result
StartPanelLocation.Left	application visible to the left in the ABB menu.
StartPanelLocation.Right	application visible to the right in the ABB menu.
StartPanelLocation.None	application is not visible at all in the ABB menu.

NOTE!

`StartPanelLocation.None` was introduced in 5.11.01. Applications that use it can therefore NOT be run on RobotWare releases older than 5.11.01.

Application type

As you can tell from the two Hello World icons visible in the task bar, two instances of the Hello World application have been started. To enable this the seventh parameter is changed to : `TpsViewType.Dynamic`. Possible view type values are shown in the table:

Parameter value	Result
TpsViewType.Dynamic	The user can start multiple instances of the application.
TpsViewType.Static	The user can only start one instance of the application.
TpsViewType.Invisible	A background application with no GUI and no icon on the task bar. Can be started automatically or manually. Only one instance is possible.

Continues on next page

Continued

NOTE!

Unless there is special need for it, you should allow the user to start only one instance of the application: `TpsViewType.Static`. The reason is that working with several instances takes up valuable memory resources.

Startup type

The eighth parameter determines how the application is started. In the example the startup type is `TpsViewStartupTypes.Manual` and the application is started from the ABB menu. Using the manual startup type it is also possible to have the application started by RAPID or at operating mode change to auto for example. See *System features supporting the use of customized screens on page 78*for information about how to do this.

If TpsViewStartupTypes.Automatic is chosen, the application is started automatically by TAF whenever the FlexPendant is restarted or a new user logs on.

Related information

For information about the **Style** setting of the **Project Wizard**, see *Container style on page 97*.

To find the **FlexPendant SDK Project Wizard** in Visual Studio, see *Setting up a new project on page 80*.

3.3.6. ITpsViewSetup and ITpsViewActivation

ITpsViewSetup

An application that TAF needs to initialize must have a default (empty) constructor and must implement the interface `ITpsViewSetup`, which specifies the two methods `Install` and `Uninstall`.

Install and Uninstall

`Install` is called when the application is created in TAF. The parameters are used by TAF and should not be modified. It is recommended to add code for further initiations and allocation of system resources in this method, e.g. load assemblies, open communication channels, etc.

When the user closes the application `Uninstall` is called. This happens right before the application is deleted. It is recommended to add code for disposal of system resources in this method, e.g. unload assemblies, close sockets, etc.

ITpsViewActivation

All custom applications should implement the `ITpsViewActivation` interface, which specifies the two methods `Activate` and `Deactivate`. These are used by TAF to notify applications when they get focus and when they lose it.

Activate and Deactivate

`Activate` is run every time the application gets focus. This happens at initialization, after the `ITpsViewSetup.Install`l method has been run, and when the user presses the application icon in the task bar.

`Deactivate`, accordingly, is run every time the application loses focus. This happens when the user closes the application, before the `ITpsViewSetup.Uninstall` method has been run, and when the user presses another application icon in the task bar.

NOTE!

It is recommended to activate and deactivate timers in these methods. This way timers will not run when other applications are in focus, thus saving processor power. For the same reason, any subscription to controller events should be disabled in `Deactivate` and enabled again in `Activate`. Note that current values should be read before enabling the subscription.

Simple code examples

This table shows the basic things that the `ITpsView` methods are used for in a custom application:

Method	Usage

Install	Create the `Controller` object:
	VB:
	`AController = New Controller`
	C#:
	`aController = new Controller();`

Continues on next page

Continued

Method	Usage

Activate	Add subscription to controller event:
	VB:
	`AddHandler AController.OperatingModeChanged, AddressOf UpdateUI`
	C#:
	`AController.OperatingModeChanged += new OperatingModeChangedEventHandler(UpdateUI);`

Deactivate	Remove subscription to controller event:
	VB:
	`RemoveHandler AController.OperatingModeChanged, AddressOf UpdateUI`
	C#:
	`AController.OperatingModeChanged -= new OperatingModeChangedEventHandler(UpdateUI);`

Uninstall	Remove the controller object:
	VB:
	`If Not AController Is Nothing Then`
	`AController.Dispose()`
	`AController = Nothing`
	`End If`
	C#:
	`if (aController != null)`
	`{`
	`aController.Dispose();`
	`aController = null;`
	`}`

3.4. Release upgrades and compatibility

About this section

Why did RAB 5.08 enforce an upgrade to Visual Studio 2005?

Will a RAB application still work if the customer upgrades the robot system with the latest RobotWare release?

What happens if you develop a RAB application using the 5.11 release and the customer system uses RobotWare 5.10?

Such questions are dealt with in this section.

Platform upgrades

The Microsoft platform that the FlexPendant uses still goes through major improvements, which ABB needs to take advantage of. This may concern performance or other issues.

In RAB 5.08 for example, Visual Studio 2003 was no longer supported. It may seem that this was uncalled-for, but there are always reasons why such changes occur.

The Visual Design support for the FlexPendant SDK, had long been on the wish list. When Microsoft provided the support necessary to meet this need, with CF 2.0, an upgrade of the FlexPendant software platform was made. The development of the design support for FlexPendant controls for RAB 5.08 made a transition to Visual Studio 2005 necessary.

NOTE!

RAB 5.10 supports Visual Studio 2005. RAB 5.11 supports VS 2005 and VS 2008.

Matching RAB and RobotWare release

You should be aware that the RAB SDKs are developed and tested for a specific RobotWare release. The general rule is therefore that you develop an application for a certain release.

Compatibility between revisions is however guaranteed (e.g. RW 5.11.01 will be compatible with RAB 5.11).

RobotWare upgrades

At some time during the lifetime of your application, a robot system that your application targets may be upgraded with a later RobotWare version.

As for a FP SDK application this means that the runtime will change, i.e. the FP SDK assemblies located in RobotWare will be different from the ones that were used when the application was developed. Generally, the only way to be sure that the existing application will work with a newer RobotWare version is to compile the source code with the FP SDK version that matches the intended RobotWare version. Normally compilation succeeds without any code changes, as FP SDK 5.09 - 5.11 are fully compatible.

The PC SDK it is normally compatible with a newer RobotWare release. The PC that hosts the PC SDK application at the customer, however, still needs an upgrade of the Robot Communication Runtime, so that it matches the new robotware release. See *ABB Industrial Robot Communication Runtime.msi on page 310*If you decide to upgrade the PC SDK application, you must also remember to upgrade the runtime environment of the customer's PC. See *Deployment of a PC SDK application on page 309* for details.

Continues on next page

Continued

NOTE!

You find all the details about compatibility between different RAB versions in the Release Notes.

TIP!

When compiling your project notice any warnings of obsolete methods, as these will probably be removed in the next RAB release.

Prepared for change

To sum up, it is important to keep source code safe and available for maintenance.

TIP!

Ildasm is a Microsoft tool, which comes with the installation of Visual Studio, that you may find useful. It enables you to open the manifest of a specified assembly and quickly find out about dependencies for example.

Find out more about it at http://msdn2.microsoft.com/en-us/library/aa309387(VS.71).aspx

3.4. Release upgrades and compatibility

4 Developing RAB applications

4.1. Introduction

About this chapter

This chapter deals with analysis, design and implementation of RAB applications.

It also discusses some specific programming issues that are important for both PC and FlexPendant SDK users:

- thread conflicts and how to avoid them
- controller events and event handling
- errors and exception handling
- the User Authorization System

The chapter does not include hands-on information on how to set up your first project or detailed information on how to use the PC and FlexPendant SDK class libraries, as these topics are covered in dedicated chapters.

Basic approach

In most aspects, using the PC or FP SDK for application development presents no major difference compared to ordinary .NET development. The .NET class libraries are used for everything that is not robot specific. In addition, you use the public Controller API of the SDKs.

NOTE!

When using the .Net class libraries for FlexPendant development, see *Version Information* to be sure that the class or method is supported on .NET Compact Framework 2.0, which runs on the FlexPendant.

Take this good advice if you are going to program your first RAB application

Step	Action
1.	**Before you start** Learn the basics about RAB programming by reading all relevant sections of this manual. Feel reassured that this is a timesaving activity and do not rush into coding. If you are going to develop an application for the FlexPendant, a device with limited resources compared to a PC, studying the chapter *Robust FlexPendant applications on page 181* is crucial.
2.	**During development** Frequently test application functionality. Do not wait too long before you test a FlexPendant application on a real FlexPendant device.
3.	**After development** If you develop a FlexPendant application, set up a long term test before use in production. If not, you risk that the FlexPendant slowly runs completely out of memory and crashes due to your application. Verify application behavior, performance and memory consumption. Use the services offered by the ABB.Robotics.Diagnostics namespace, e.g. memory tracking. See *FlexPendant - Debugging and troubleshooting on page 279* for further information.

4.2. Analysis and design

About this section

The purpose of Robot Application Builder is to provide operator interfaces that fulfill specific customer needs. This section focusses on the development phases preceding the actual coding: analysis and design.

Object oriented software development

.NET is entirely object-oriented. Platform services are divided into different namespaces such as `System.Collections`, `System.Data`, `System.IO`, `System.Security` and so on. Each namespace contains a set of related classes that allow access to platform services.RAB, too, is completely object oriented. Its class libraries are organized in different namespaces such as `ABB.Robotics.Controllers.RapidDomain`, `ABB.Robotics.Controllers.MotionDomain` etc.

Some experience in object orientation is necessary to start developing custom applications. It is presumed that you feel comfortable with concepts such as objects, classes, methods, inheritance, encapsulation etc.

Object oriented Analysis and Design

Object Oriented Analysis and Design, OOAD, is a popular topic in computer science literature, where the importance of doing a thorough analysis and design before starting coding is commonly accepted. A well designed OO application has a true representation in the real world. Classes have well defined areas of responsibility and collaborate in an efficient way to achieve what is required.

Analysis based on communication and use cases

The main idea of Robot Application Builder is, as has already been pointed out, that custom operator interfaces can be developed close to end-users, taking their specific needs in consideration. It therefore goes without saying that analysis is crucial.

The result of the object-oriented analysis is a description of what we want to build, often expressed as a conceptual model. Any other documentation that is needed to describe what we want to build, for example pictures of the User Interface, is also part of analysis.

The most important aspect for RAB development is communication with end-users. Activities which support a shared view of what should be developed are strongly recommended. Such activities may include:

- creating and discussing use cases together
- coding or drawing prototypes and get end-user feedback

Continues on next page

Continued

In short, involving end-users from the early stages and keeping them involved throughout the development project is the best strategy.

NOTE!

Customer satisfaction is what has driven the development of Robot Application Builder. Do make sure that you have really understood what the end-users of your application need.

Design is about managing complexity

The result of the object-oriented design details how the system can be built, using objects. Abstraction is used to break complex problems into manageable chunks. It makes it possible to comprehend the problem as a whole or to study parts of it at lower levels of abstraction.

It takes years to become a skilled object oriented designer. Design theory must be transformed into practical experience and iterated over and over again.

The goal is to produce high quality code, which is cost-efficient and easy to maintain. This is achieved, for example, when adding new functionality will involve minimal changes to existing code and most changes will be handled as new methods and new classes.

Do you need to do design?

There is a huge difference in complexity when creating software such as .NET framework, for example, and a custom operator view for IRC5. Obviously, the more complex a system the more careful design is needed. Accordingly, the larger and more complex a custom application needs to be, the more likely you are to spend time on design.

This table presents some considerations before deciding how well you need to design your application before starting coding:

Consideration	Advice
How much code is it going to be?	If it is going to be a very simple application with just one view and a few buttons there is no need even to split the code between different classes and files. If there will be a substantial amount of code and there might be further extensions later on, spending time on design becomes more relevant.
Will different developers work on different classes/ components? Will you maintain the code yourself, or may it be done by others?	If yes, spending time om design becomes more relevant.
Is the real time aspect of the application important?	If yes, coding efficiently is important. This will much more easily be achieved if you spend some time on design. **Note!** You are also recommended to read through the chapter *Robust FlexPendant applications on page 181* before starting coding.

As complex or as easy as you wish

A simple custom application can be created in a day or two using RAB. A large custom application with a number of different views, offering advanced robot system functionality, however, may take months to develop and will require considerable programming skill.The recommendation is to start developing a simple application, which you execute on the target platform, before moving on to advanced RAB programming.

4.3. Controller events and threads

Overview

A controller event is a message from the controller that something has happened. Events can be caught and acted upon by RAB applications.

Controller events use their own threads. This means that user interface threads and controller event threads can get into conflict. This section gives information on how to prevent this.

Controller events

RAB applications can subscribe to a number of controller events. These are all described in the reference documentation for the SDKs.

The table shows some events that exist both in the PC and FlexPendant SDK.

The event...	occurs when...
StateChanged	the controller state has changed.
OperatingModeChanged	the controller operating mode has changed.
ExecutionStatusChanged	the controller execution status has changed.
Changed	the value or the state of the I/O signal has changed.
MessageWritten	the EventLog has a new message
ValueChanged	the value of a RAPID data has changed.

Continues on next page

Continued

NOTE!

There is no guarantee you will get an initial event when setting up/activating a controller event. You need to read the initial state from the controller.

GUI and controller event threads in conflict

You should be aware that controller events use their own threads both on the FlexPendant and PC platform. If a GUI thread and a controller event thread get into conflict, deadlocks or overwritten data may occur. This may happen when a controller event is succeeded by an update of the user interface, which is indeed a very common scenario.

You then need to take action in your code to control that the user interface update is executed by the GUI thread and not by the controller event thread. This is done by enforcing a thread switch using the `Invoke` or `BeginInvoke` method. See *Invoke method on page 68* for information on how this is done along with code examples.

On the other hand, if the controller event should NOT cause any update of the user interface, you should not take any special action. Using `Invoke / BeginInvoke` is performance demanding and should not be used more than necessary.

NOTE!

Thread conflicts often cause hangings. The FlexPendant touch screen or the PC application UI then stops responding and the application has to be restarted.

Examine what exception has been thrown when you encounter such a situation. The exceptions `System.NotSupportedException` (FlexPendant platform) and `System.InvalidOperationException` (PC platform) indicate thread conflicts. See the next section for information on how to use `Invoke` to solve the problem.

Invoke method

All PC application windows and FlexPendant views must inherit `Control` / `TpsControl`, which implement `Invoke` and `BeginInvoke`. These methods execute the specified delegate/event handler on the thread that owns the control's underlying window handle, thus enforcing a switch from a worker thread to the GUI thread. This is precisely what is needed when a controller event needs to be communicated to the end user of the system.

`Invoke` should be called inside the event handler taking care of the controller event. Notice that you have to create a new object array for the sender and argument objects:

VB:

```
Me.Invoke(New EventHandler(AddressOf UpdateUI), New Object()
     {sender, e})
```

C#:

```
this.Invoke(new EventHandler(UpdateUI), new Object[] {sender, e});
```


Continues on next page

Also notice that if you use `EventHandler` in the `Invoke` method and not the specific delegate class, e.g.`DataValueChangedEventHandler`, you need to typecast the argument in the delegate which updates the user interface. How this is done is shown by the example below:

VB:

```
Private Sub UpdateUI(ByVal sender As Object, ByVal e As
      System.EventArgs)
  Dim Args As ExecutionStatusChangedEventArgs
  Args = DirectCast(e, ExecutionStatusChangedEventArgs)
  Me.Label1.Text = e.NewStatus.ToString()
End Sub
```

C#:

```
private void UpdateUI(object sender, System.EventArgs e)
{
  ExecutionStatusChangedEventArgs args;
  args = (ExecutionStatusChangedEventArgs) e;
  this.label1.Text = e.NewStatus.ToString();
}
```

NOTE!

The difference between `Invoke` and `BeginInvoke` is that the former makes a synchronous call and will hang until the GUI operation is completed, whereas `BeginInvoke` executes the specified event handler asynchronously. Which method you want to use depends on the logics of your program. The recommendation is to choose `BeginInvoke` whenever possible.

NOTE!

If your code tries to access a GUI control from a background thread the .NET common language runtime will throw a `System.NotSupportedException` (FlexPendant platform) or a `System.InvalidOperationException` (PC platform).

TIP!

If you are using the FlexPendant SDK there is further information about threads in *Thread affinity on page 196* and *Invoke on page 196*.

4.4. User Authorization System

Overview

In the robot controller there is a system controlling user access: the *User Authorization System (UAS)*. If this feature is used each user needs a user name and a password to log on to a robot controller via the FlexPendant or RobotStudio. If the controller connection for any reason is lost, the user has to log on again.

The controller holds information on which operations different users are allowed to perform. The UAS configuration is done in Robot Studio.

TIP!

To learn more about UAS use the help function in Robot Studio.

Accessing UAS from custom applications

Before sensitive controller operations are performed, a FlexPendant SDK application should check that the user currently logged on has the corresponding UAS rights.

Accessing UAS is done by using the property `AuthorizationSystem` on the controller object:

VB:

```
Dim UAS As UserAuthorizationSystem =
     Me.AController.AuthorizationSystem
```

C#:

```
UserAuthorizationSystem uas =
     this.aController.AuthorizationSystem;
```

Grants and Groups

UAS rights are called *Grants*. The specific user belongs to one of several defined *Groups*, where each group has a number of specified grants.

To ensure that the user has the necessary grant to perform an operation, you use the `CheckDemandGrant` method on the `AuthorizationSystem` object. The grant to check is passed as an argument:

VB:

```
If UAS.CheckDemandGrant(Grant.ModifyRapidProgram) Then
  ATask.LoadModuleFromFile(ALocalFile, RapidLoadMode.Replace)
End If
```

C#:

```
if (uas.CheckDemandGrant(Grant.ModifyRapidProgram)) {
  aTask.LoadModuleFromFile(localFile, RapidLoadMode.Replace);
}
```


Continues on next page

NOTE!

The RAB application cannot override the UAS configuration. This means that the system will in the end prevent the user from performing an action that is not allowed.

MessageBox feedback

If a UAS grant is missing the user should get information about it. This can be done in a message as shown in this example:

```
msg = "You are not allowed to perform this operation, talk to your
      system administrator if you need access."
title = "User Authorization System"
```

For the PC platform (VB and C#):

```
MessageBox.Show(msg,title,MessageBoxIcon.Exclamation,MessageBoxB
      uttons.OK)
```

For the FlexPendant platform:

VB:

```
GTPUMessageBox.Show(Me, Nothing, msg, title,
      System.Windows.Forms.MessageBoxIcon.Asterisk,
      System.Windows.Forms.MessageBoxButtons.OK)
```

C#:

```
GTPUMessageBox.Show(this, null, msg, title,
      System.Windows.Forms.MessageBoxIcon.Asterisk,
      System.Windows.Forms.MessageBoxButtons.OK);
```

GetCurrentGrants and DemandGrant

Another possibility is to retrieve all grants for the current user calling `GetCurrentGrants`, then iterate over the grants collection and search the necessary grants.

Yet another solution is to call `DemandGrant` with one of the static `Grant` members as in argument.

If the user does not have the specified grant the FlexPendant SDK throws a `UasRejectException` and the PC SDK throws a `GrantDemandRejectedException`.

TIP!

Learn more about UAS and `Grant` members in the SDK Reference Help.

4.5. Exception handling

Overview

The .NET programming languages provide built-in support for exception handling, which allows the program to detect and recover from errors during execution.

In managed code, execution cannot continue in the same thread after an unhandled exception. The thread terminates, and if it is the program thread, the program itself terminates. To avoid this, accurate exception handling should be used.

Try-catch-finally

Exceptions are handled in `try` - `catch` (-`finally`) blocks, which execute outside the normal flow of control.

The `try` block wraps one or several statements to be executed. If an exception occurs within this block, execution jumps to the `catch` block, which handles the exception.

The `finally` block is executed when the `Try` block is exited, no matter if an exception has occurred and been handled. It is used to clean up system or controller resources.

If you do not know what exceptions to expect or how to handle them, you can catch them and do nothing. This, however, may result in difficult error tracing, as exceptions include information on what caused the problem. Therefore, try at least to display the exception message, either by using a message box or the types `Debug` or `Trace`. See *Debug output on page 279* and *Trace and Debug on page 281* for further information.

Typecasting

When typecasting `Signal` or `RapidData` values, for example, there is a potential risk of typecast exceptions. To avoid this you can check the object using the `is` operator for both value and reference types:

VB:

```
If TypeOf ARapidData.Value Is Num Then
  Dim ANum As Num = DirectCast(ARapidData.Value, Num)
  .....
```

C#:

```
if (aRapidData.Value is Num)
{
  Num aNum = (Num) aRapidData.Value;
  ....
}
```

In C# it is also possible to use the `as` operator for reference types. A null value is returned if the type is not the expected one:

C#:

```
DigitalSignal di = this.aController.IOSystem.GetSignal("UserSig")
     as DigitalSignal;
if (di == null)
{
  MessageBox.Show(this, null, "Wrong type");
}
```

Continues on next page

Continued

Exception handling for the PC platform

Exceptions thrown from the controller are handled by the PC SDK `ExceptionManager`, which converts the internal HRESULT to a .NET exception with a reasonable exception description, before it is thrown to the custom application layer. The application handling of these exceptions should apply to general .NET rules.

Exceptions are expensive in a performance perspective and should be avoided if there are other alternatives. If possible use a `try-finally` block to clean up system and unmanaged resource allocations.

Exception handling for the FlexPendant platform

The FlexPendant SDK provides several exception classes, which are used when errors occur. If the operation is rejected by the controller safety access restriction mechanism, for example, a `RejectException` is thrown.

If an unhandled exception occurs the application crashes and TAF displays a gray message on the FlexPendant. After confirmation the FlexPendant will restart. To avoid this you should make sure that any potential error situations are dealt with properly by your code.

5.5_1

The message that you get when an unhandled error situation occurs may look like this. Do NOT contact ABB, but fix the error handling of your application.

Continues on next page

Continued

NOTE!

Learn how the exception classes of the FlexPendant SDK work by using the *FlexPendant SDK Reference Documentation*. Also see *SDK exception classes on page 194* to get more detailed information about exception handling for the FlexPendant platform.

.NET Best Practices

.The *.NET Framework Developer's Guide* presents the following best practices for exception handling:

- Know when to set up a try/catch block. For example, it may be a better idea to programmatically check for a condition that is likely to occur without using exception handling. For errors which occur routinely this is recommended, as exceptions take longer to handle.
- Use exception handling to catch unexpected errors. If the event is truly exceptional and is an error (such as an unexpected end-of-file), exception handling is the better choice as less code is executed in the normal case.Always order exceptions in catch blocks from the most specific to the least specific. This technique handles the specific exception before it is passed to a more general catch block.

4.6. How to use the online help

Overview

The online help comes with the installation of Robot Application Builder and is accessible from Windows **Start** menu.

The recommendation is to read this user's guide carefully as you develop your first RAB application. *FP SDK Reference* and *PC SDK Reference* are important complements to this manual, as these make up the complete reference to the class libraries of RAB. See *Documentation and help on page 16* for details.

NOTE!

The *SDK Reference* is NOT integrated in Visual Studio. You must open it from the **Start** menu.

TIP!

See *Documentation and help on page 16* for the web address to *RobotStudio Community*, where RAB developers discuss software problems and solutions online.

4.6. How to use the online help

5 Using the FlexPendant SDK

5.1 Introduction

5.1.1. About this chapter

Overview

This chapter gives detailed information on how to use the FlexPendant SDK.

These topics are covered:

- How to take advantage of some system features that support the use of customized screens.
- How to utilize the integrated Visual Studio wizard to set up a FlexPendant project.
- How to add the FlexPendant SDK GUI controls to the Visual Studio Toolbox.
- How to build the user interface using the integrated design support.
- How to program FlexPendant SDK GUI controls.
- How to launch other applications from your application.
- How to implement controller functionality using CAPI.

The design support in Visual Studio enables you to visually lay out the application, reducing the need to write code. This speeds up development and gives you a more precise control of the appearance of your FP screens.

Using the FlexPendant SDK it is possible to launch several of the standard FlexPendant applications from your application, which is often a very handy alternative to handling complicated procedures on your own, such as reading and writing RAPID data for example.

The Controller API (CAPI) is at the core of the FlexPendant SDK. It is used to access the robot controller, which the FlexPendant is attached to. First there is information about how this public API is organized. Then each domain of CAPI is dealt with separately. There are code samples in C# and VB throughout the chapter.

5.1.2. System features supporting the use of customized screens

Flexible user interfaces

The FlexPendant can be adapted to end-users' specific needs in many different ways. It can be operated in 14 different languages, including Asian character-based languages such as Japanese and Chinese. Left-handed operators can adapt the device from its default setting by simply rotating the display through 180 degrees. Four of the eight hard keys are programmable, i.e. their function can be assigned by the end-user.

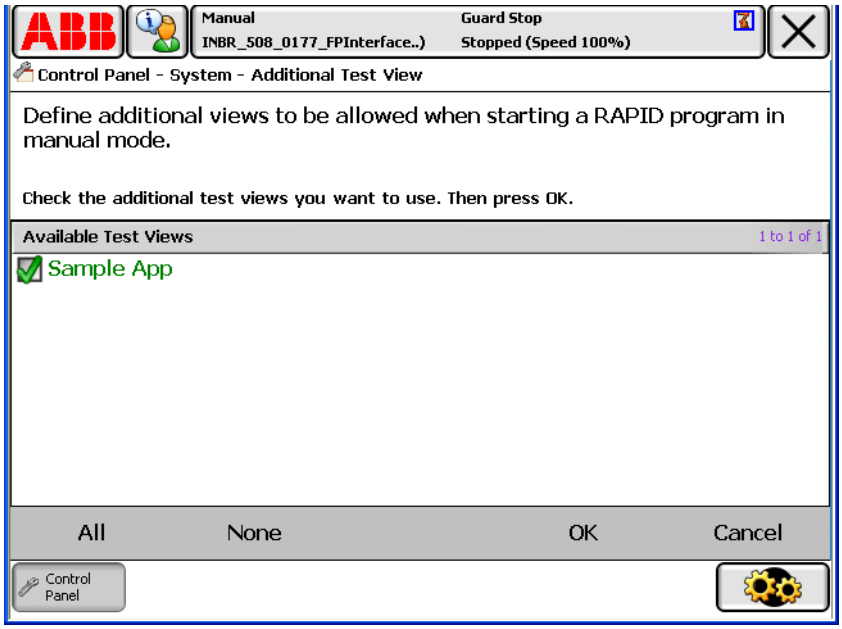
Customized FlexPendant screens, tailored to end-users' needs is yet another way of rendering the flexible solutions required by many customers. To support the use of customized screens there are a couple of features that you may want to tell the end-users of your application about.

Configure the FlexPendant

Using the FlexPendant configuration facilities (Control Panel - FlexPendant) it is possible to configure the FlexPendant to allow RAPID execution in manual mode from an FP SDK view. You can also make the FlexPendant automatically display an SDK view at operating mode change.

Additional Test View

Set the FlexPendant configuration property *Additional Test View* if you want to be able to start RAPID execution in manual mode with a custom application as the active view.



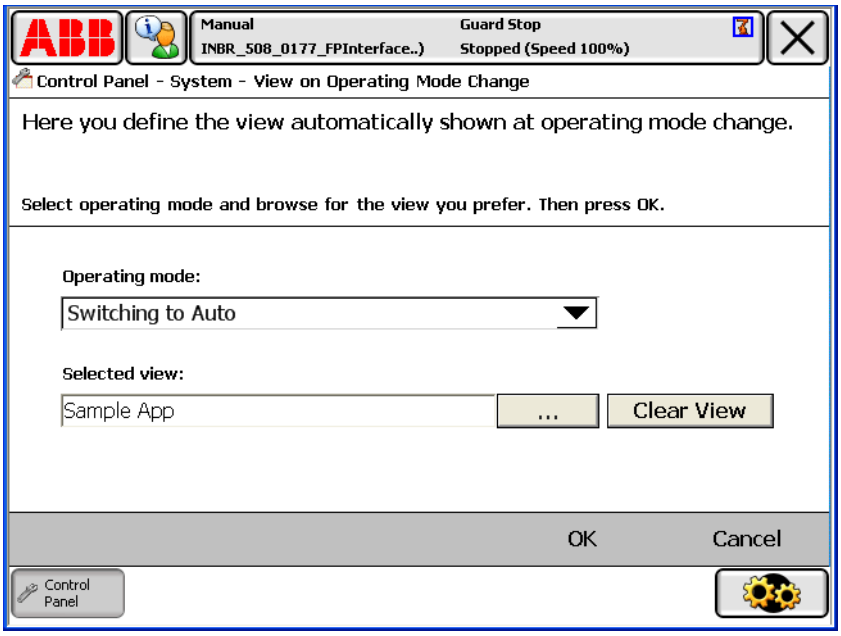
6.1.0_1

Continues on next page

Continued

View On Operating Mode Change

Set the FlexPendant configuration property *View On Operating Mode Change* if you want the custom application to become active when the controller operating mode is switched to auto, manual or manual full speed.



6.1.0_2

Use RAPID instruction to launch RAB application

The RAPID instruction `UIShow` (User Interface Show) is used to communicate with the user of the robot system via a FlexPendant application. Both RAB applications and standard applications can be launched using this instruction.

Example:

The RAPID code below launches the custom application *TpsViewMyApp*.

```
CONST string Name :="TpsViewMyApp.gtpu.dll";CONST string Type
     :="ABB.Robotics.SDK.Views.TpsViewMyApp";UIShow Name, Type;
```

For this to work the robot system must have the RobotWare option *FlexPendant Interface*. The assemblies TpsViewMyApp.dll and TpsViewMyApp.gtpu.dll must be located in the HOME directory of the active robot system. (When the assemblies have been downloaded to the controller the FlexPendant must be restarted in order to load them.)

NOTE!

See the RAPID reference manual for further information about the `UIShow` instruction.

5.2 Setting up a new project

5.2.1. Using the project template in Visual Studio

Overview

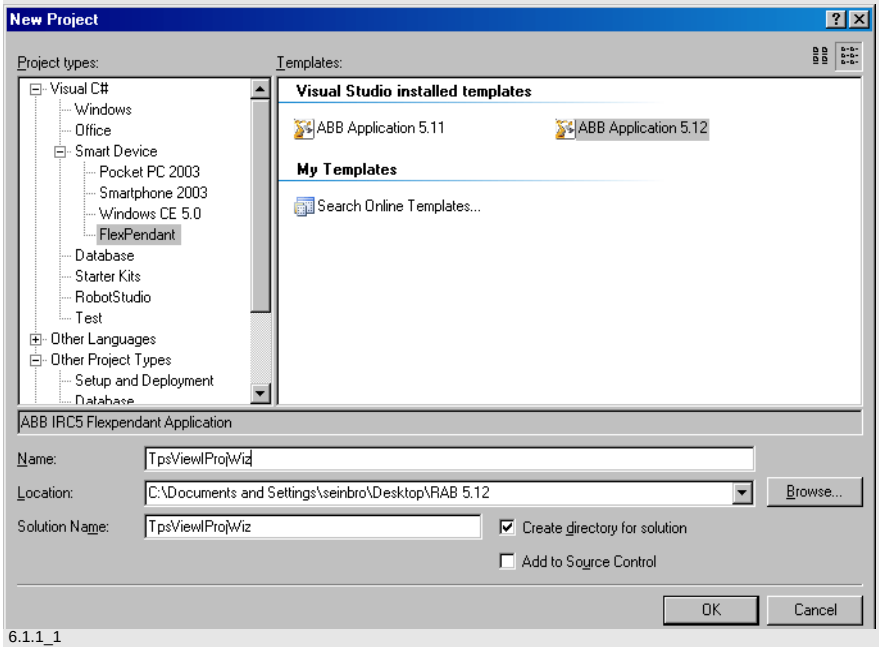
It is very simple to set up a new FlexPendant project using the integrated project template that comes with the installation of Robot Application Builder. It will automatically add the most common SDK references to the project and auto-generate some source code for the main application window, in the selected programming language, either C# or Visual Basic.

NOTE!

To add another view to the FlexPendant project, you do not need to start a new project. See *Adding a view to a custom application on page 99* for information about how to do it.

Setup procedure

Follow these steps to create a FlexPendant project:

Step	Action
1.	On the **File** menu in Visual Studio, point to **New** and then click **Project**.
2.	In VS 2005, in the **New Project** dialog select *Visual C# / Smart Device / FlexPendant* or *Other Languages/Visual Basic/Smart Device/FlexPendant*. In VS 2008, select *Visual C# / FlexPendant* or *Other Languages/Visual Basic/FlexPendant*).  6.1.1_1 **NOTE!** If you have several RAB installations on your PC, there will be several templates to choose from. Make sure you select the template that match the RobotWare version that your application should target.

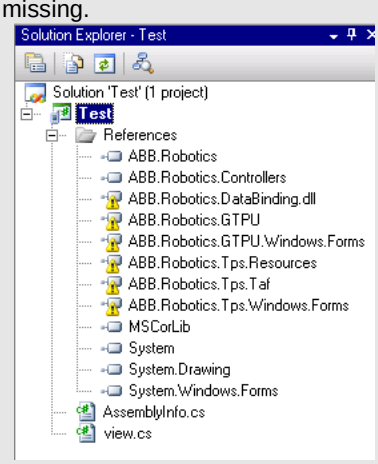
Continues on next page

Continued

Step	Action
3.	Enter the name of the application in the **Name** field and where you want it to be stored in the **Location** field. Click **OK**. **NOTE!** The name has to start by *"TpsView"* . If you forget it an error will be generated by the ABB verification tool when the project is built.
4.	The **FlexPendant SDK Project Wizard** is now launched. For information about how to configure the application using this wizard see *FlexPendant TpsView attribute on page 54* and *Container style on page 97*. When you are ready click **OK**.
5.	You need to set up the design support for the FlexPendant GUI controls before starting programming. How this is done is detailed in *Setting up design support for FlexPendant controls on page 83*. **NOTE!** If the SDK references seem to be missing in your FlexPendant project, see the following section for instructions about how to solve the problem.

Add any missing references

The SDK references needed by the FlexPendant project are added automatically when the Project wizard is completed. However, there is a possibility that Visual Studio cannot provide the path to these dlls. If this is the case you should add the references path manually. Follow these steps:

Step	Action
1.	Look at the **Solution Explorer** of Visual Studio. If you have a C# project expand the **References** node. Warning triangles mean that the path to the referenced dlls is missing.  6.1.1_2 For a VB project the **Solution Explorer** looks the same, except that there is no **References** node.

Continues on next page

5.2.1. Using the project template in Visual Studio

Continued

Step	Action
2.	C#: Select the project icon, right-click and select **Properties**. Click the **Reference Paths** tab and add the path to the SDK assemblies. VB: Select the project icon, right-click and select **Properties**. Click the **References** tab. If the path to the SDK references is missing add it by browsing to the directory where they are located.  **NOTE!** The default path is *C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder 5.xx\FlexPendant SDK \bin*.
3.	Save the project. You will notice that any warning triangles in the **Solution Explorer** **References** node will disappear.

5.2.2. Setting up design support for FlexPendant controls

Overview

This section describes how to make the FlexPendant GUI controls accessible in Visual Studio.

Procedure

Follow these steps to add the FlexPendant controls to the Visual Studio **Toolbox**:

Step	Action
1.	On the **View** menu, select **Toolbox.**
2.	Right click in the **Toolbox** area and select **Add Tab**.
3.	Name the new toolbox tab, e.g. *FlexPendant Controls*.
4.	Right click in the area of the new tab and select **Choose Items**.
5.	In the **Choose Toolbox Items** dialog, browse to the directory where the FlexPendant SDK assemblies are located and import the following assemblies: • ABB.Robotics.Tps.Windows.Forms.dll • ABB.Robotics.GTPU.Windows.Forms.dll • ABB.Robotics.DataBinding.dll The default location is *C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder\FlexPendant SDK 5.xx\bin*.
6.	In the Solution Explorer right-click view.cs (view.vb if you have a VB project) and select **View Designer** if you are not already in design mode. As you see, the FlexPendant specific controls are now accessible in the **Toolbox**. See *Introduction to visual design support on page 84* for information about how to use them.

NOTE!

The way you use the Visual Studio Designer to implement FlexPendant controls is very similar to implementing ordinary .NET controls. In this manual useful information which may not be obvious for all users is provided. But oftentimes, it is the general Visual Studio Help that will answer any questions you may have about control properties and the like.

5.3 Building the user interface

5.3.1. Introduction to visual design support

What is visual design support?

Design Support for Compact Framework User Controls was a new feature of Visual Studio 2005. It enabled design support for FlexPendant SDK controls to be included in RAB 5.08.

From RAB 5.08 onwards you visually design the FlexPendant application user interface in the Visual Studio Designer. FlexPendant controls are dragged from the toolbox to the designer area, moved and resized by clicking and dragging. By applying different settings in the **Properties** window of a control, you refine its appearance and behavior.

To be able to use the visual design support you must add the FlexPendant controls to the Visual Studio toolbox. How to do this is detailed in *Setting up design support for FlexPendant controls on page 83*.

NOTE!

Design support for FlexPendant controls has long been on RAB users' wish list. It is indeed a time-saving feature, as most of the code supporting the graphical user interface is now auto generated.

Why special controls for the FlexPendant?

You may wonder why the standard Microsoft Windows controls have not been considered good enough to be used for the FlexPendant touch screen.

The answer is that some Windows controls may very well be used. Other Windows controls are however not so well suited for the FlexPendant touch screen. To navigate using your finger controls in particular need to be large enough. In some other cases the Windows controls simply do not look very good on the touch screen.

NOTE!

In the *FP SDK Reference*, click the **Contents** tab and the **ABB.Robotics.Tps.Windows.Forms** node to get an overview and a short description of all ABB controls you may use to create the user interface of a FlexPendant application.

TIP!

How to program these ABB controls is very similar to the equivalent Windows controls. If you need code examples the best source of information is usually MSDN. You may try http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemwindowsformslistviewclasstopic.asp, for example, to find out how to program a `ListView`.

Illustration

The figure below shows the Visual Studio **Toolbox** with all of the FlexPendant controls to the left. In the **Designer** area to the right, a FlexPendant application is being developed. Part of the container control, along with a number of ABB and standard Windows controls can be

Continues on next page

seen.

6.3.1_1

The following features are worth noting:

- A Windows **PictureBox** control is used as the container of a photo.
- The FlexPendant button has a property called **BackgroundImage**, used to display an image. The ABB image library located in the *Resources* folder, which comes with the installation of Robot Application Builder, has numerous icons, which can be used by custom applications. You can of course also use photos and icons of your own.
- Some of the ABB controls, such as the **SignalBindingSource**, have no graphical representation in design-time. As you see in the figure, they are placed in the components pane under the main form. Code is of course generated, just like for the controls that you see on the form.
- Usually a mix of Windows and ABB controls are used for a FlexPendant SDK application. For example, as there is no ABB **RadioButton** or **CheckBox** the equivalent Windows controls are used. In the figure, the standard Windows **CheckBox** is used.

Continues on next page

Continued

CAUTION!

Do not forget about the limited amount of memory available for custom applications when adding images or photos! See *Technical overview of the FlexPendant device on page 182*.

CAUTION!

Auto generated code for controls is located in the `method InitializeComponent`. You should not tamper with the code inside this method. Any modifications or additions to auto generated code is usually best located in the constructor, after the call to `InitializeComponent`.

Hands on - Hello world

Are you ready to program and test your first FlexPendant application? If you have not created a FlexPendant project and added the FlexPendant controls to the Visual Studio Toolbox you need to do that first. See *Using the project template in Visual Studio on page 80* and *Setting up design support for FlexPendant controls on page 83*.

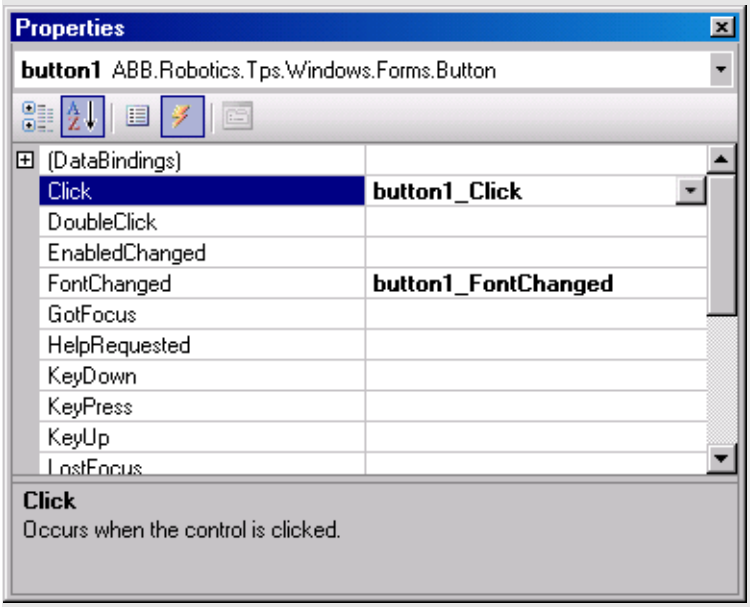
Follow these steps to create and test a simple custom application:

Step	Action
1.	Drag a FlexPendant button from the **Toolbox** to the **Designer**.
2.	Double-click the button in the **Designer**, this opens the code editor.

3.	As you see an event handler for the click event has been created. Add the code to launch the Hello World message in it:
	`private void button1_Click(object sender, EventArgs e)`
	`{`
	`  GTPUMessageBox.Show(this.Parent`
	`  , null`
	`  , "Hello world!"`
	`  , "Application Message"`
	`  , System.Windows.Forms.MessageBoxIcon.Asterisk`
	`  , System.Windows.Forms.MessageBoxButtons.OK);`
	`}`
	An alternative way of adding an event handler is shown in the next step.
	
	NOTE!
	The above code sample has no exception handling. A FlexPendant application to be used in industry must have exception handling in all event handlers. The FlexPendant has only ONE GUI thread, which all applications running on the FlexPendant share. If your application breaks the only GUI thread, all of the applications will die and the FlexPendant must be restarted manually. See *Exception handling on page 72* to find out more about this important issue.

Continues on next page

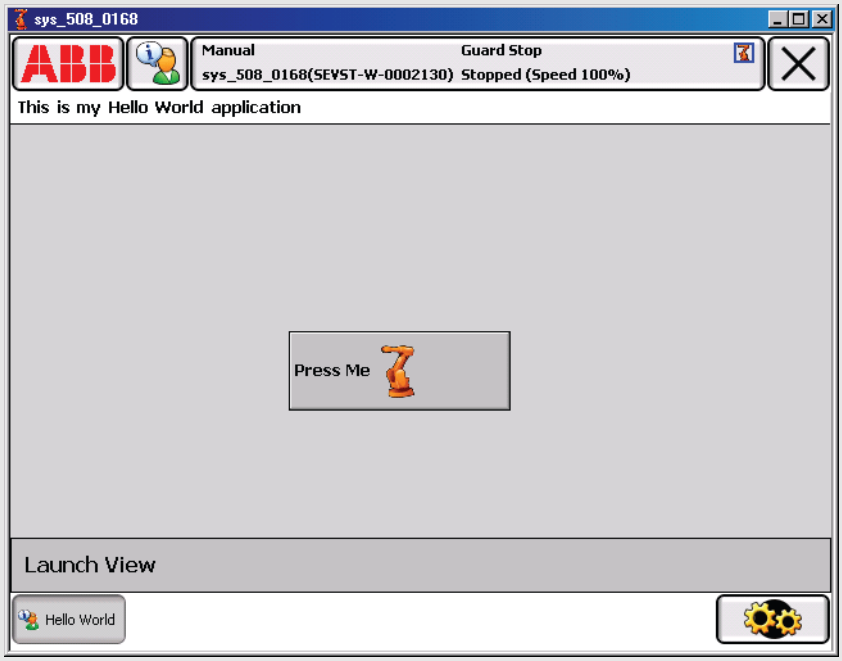
Continued

Step	Action
4.	Bring up the **Properties** window for the button by right clicking it in the **Designer** and selecting **Properties**. Click the events button (yellow flashing lightening) to see the events available for the ABB button. As shown by the figure, you have already connected an event handler to the click event.  6.3.1_1B **NOTE!** It is the default event of the control that you get when you double-click it in the Designer. To add an event handler for another button event, bring up the Properties window and double-click the event you want to generate a handler for, e.g. a `FontChanged` event handler. You will enter the code editor, the cursor inside the generated event handler.
5.	On the **Build** menu, click **Build Solution** and check that you did not get any compilation errors.
6.	To test the application you need to deploy it to a robot system. If there is no robot system available, you must create a robot system by using the *System Builder* of RobotStudio.
7.	Copy the assembly (*.dll) and the proxy assembly (*.gtpu.dll) from the bin\Debug (or bin\Release) directory of your Visual Studio project to the HOME directory created by RobotStudio when your robot system was created. Example showing default paths for copy/paste: *C:\Data\Visual Studio 2005\Projects\TpsViewMyApp\TpsViewMyApp\bin\Debug* *C:\Systems\sys1_508\HOME* **Note**! If you have already started the virtual FlexPendant you need to close it before pasting the dlls.
8.	Start the virtual FlexPendant.
9.	On the ABB menu, find your application and open it.

Continues on next page

Continued

Step	Action
10.	Click the button. The *Hello World* message will be displayed.



6.3.1_2

NOTE!

See *Hands on - step 2 on page 89* below if you wish to know how to implement the additional features shown above: application title, button background image, command bar with a menu item and an event handler to launch a standard view.

TIP!

When using the Virtual FlexPendant for test and debug you can automate the copy/paste procedure described in step seven above. This is how you do it:

1. Right click the project icon in the **Solution Explorer** and select **Properties**.
2. For a C# project, select the **Build Events** tab. (For a VB project click the **Compile** tab and then the **Build Events** button.)
3. Press the **Edit Post-build** button and add two commands which will copy the produced dlls to the directory of the robot system to run. Example:

Continues on next page

Continued

```
copy "$(TargetDir)$(TargetName).dll" "C:\Systems\sys1"
copy "$(TargetDir)$(TargetName).gtpu.dll" "C:\Systems\sys1"
```

Note! Do not remove the default post build command lines, which create the assembly proxy (*gtpu.dll).

Hands on - step 2

This section details how to program the remaining features of the *Hello World* application shown in step nine above: application title, button background image, command bar with a menu item. You will also learn how to implement an event handler for the **Launch View** command.

When you create a FlexPendant project you choose **Empty** or **Form** in the **FlexPendant SDK Project Wizard**. See *Empty or Form? on page 97* to understand the difference between them. In our example **Form** is selected as the container control.

Step	Action

1.	If you selected **Empty** for your project you can change it to **Form** by exchanging TpsControl with `TpsForm` directly in the source code, like this:
	
	C#:
	`public class TpsViewIRC5App14 : TpsForm, ITpsViewSetup, ITpsViewActivation`
	
	VB:
	`Public Class TpsViewIRC5App14 Inherits TpsForm '(was TpsControl before) Implements ITpsViewSetup, ITpsViewActivation`
	
	Notice that the **Designer** now shows the title bar on top of the container control.

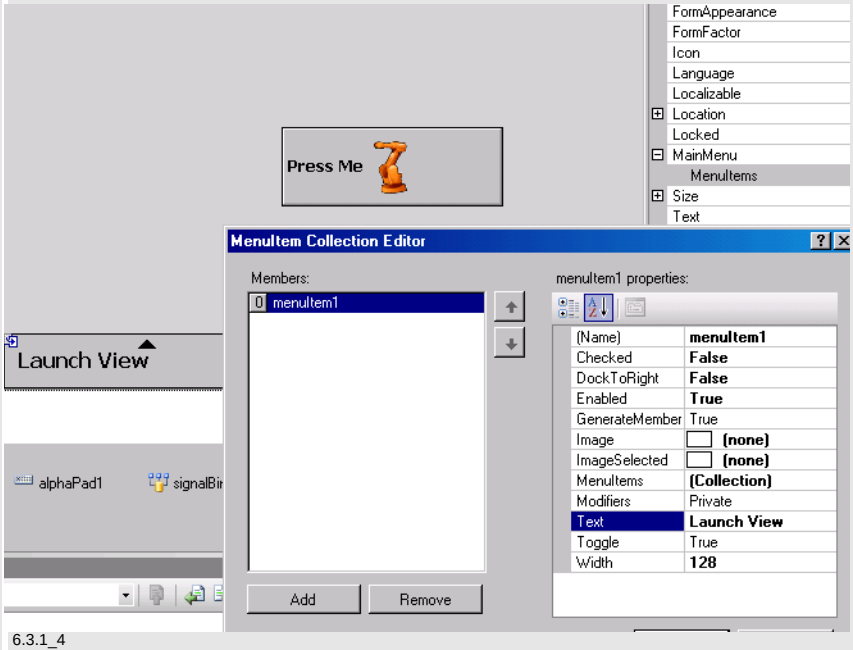
2.	Right-click somewhere in the middle of the container control and select **Properties**.
3.	Enter “This is my Hello World application” at the **Text** property. See the tip of *Empty or Form? on page 97* if the **Text** property is not available.
4.	Still using the **TpsForm Properties** window, expand the **MainMenu** property node.
5.	Click the **MenuItems** property and the browse button which appears once the property is selected.

Continues on next page

5.3.1. Introduction to visual design support

Continued

Step	Action
6.	In the **MenuItem Collection Editor**, which is now launched, click the **Add** button. Then enter “Launch View” at the **Text** property. Click **OK**.



6.3.1_4

NOTE!

The added menuItem has its own **MenuItems** property. You use it if the “Launch View” command is to present a menu instead of working as a button on the command bar. How this can be done is further detailed in *How to add menu items on page 101*.

Continues on next page

Continued

Step	Action

7.	Add an event handler with code that will launch the standard *Jogging* view when the Launch View button is pressed.
	Adding event handlers is done differently in Visual Studio depending on which programming language you are using.
	For C#:
	In the constructor, after the call to `InitializeComponent`, add a subscription to the click event of the “Launch View” command. Write:
	`menuItem1.Click +=`
	then TAB twice. As you will notice, Visual Studio intellisense auto generates an event handler, and you only need to write the code for launching the Jogging view:
	`this._launchService.LaunchView(FpStandardView.Jogging,null,false,out this._cookieJog)`
	
	For VB:
	Add an event handler using the drop down boxes above the Visual Studio Code Viewer. Find and select **MenuItem1** in the left drop down box and the **Click** event in the right drop down box. The event handler is now auto generated for you, and the only thing you need to do is to write the code for launching the Jogging view (see VB code in *Launching standard views on page 129*).
	
	NOTE!
	There is no possibility to attach an event handler in the properties window, like you do for a `Button` for example.

8.	Declare the _launchService and the _cookieJog objects: `private ABB.Robotics.Tps.Taf.ITpsViewLaunchServices _launchService;private object _cookieJog;`
9.	Retrieve the `_launchService` object in the `Install` method of your class: 6.3.1_5 See *ITpsViewSetup Install on page 128* for VB code example.
10.	In the designer, open the **Properties** window of the button.
11.	Enter “Press Me” at the **Text** property.
12.	Select the **BackgroundImage** property and browse to the *Resources* folder of the RAB installation and import “IRB140.gif”. (Default path: *C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder\FlexPendant SDK 5.08\Resources*)
13.	Build, deploy and start the application, that is repeat step five to eight in *Hands on - Hello world on page 86*.

```
/// <summary>
/// This method is called by TAF when the control is installed in the framework.
/// </summary>
bool ITpsViewSetup.Install(object sender, object data)
{
    if (sender is ITpsViewLaunchServices)
    {
        // Save the sender object for later use
        this._launchService = sender as ITpsViewLaunchServices;
        return true;
    }
    return false;
}
```

Continues on next page

5.3.1. Introduction to visual design support

Continued

Step	Action
14.	Press the **Launch VIew** command. The Jogging view should open up and get focus.

TIP!

See *Using launch service on page 128* and *Using standard dialogs to modify data on page 131* if you want to know more about the possibilities to use already existing views for your application

Visual design and user experience

There is another definition of the term *Visual design*, which is not in the scope of this manual. It has to do with how the design of the software user interface affects user experience. This is nonetheless an important topic for a FlexPendant application developer. Knowing what makes a user interface intuitive and easy to use is essential, as this is exactly what is expected from a custom operator interface.

TIP!

A FlexPendant style guide comes with the installation of RAB. It will help you to present application functionality in an intuitive way, teaching best practices of visual design and user experience from the FlexPendant perspective. It is a preliminary draft, but still very useful.

You may also want to study the Microsoft standard for visual design at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/ch14a.asp

5.3.2. GUI controls and memory management

Overview

All of the FlexPendant SDK controls belong to the `ABB.Robotics.Tps.Windows.Forms` namespace. This namespace thus includes well over 40 classes and subclasses used to develop the user interface of a FlexPendant application.

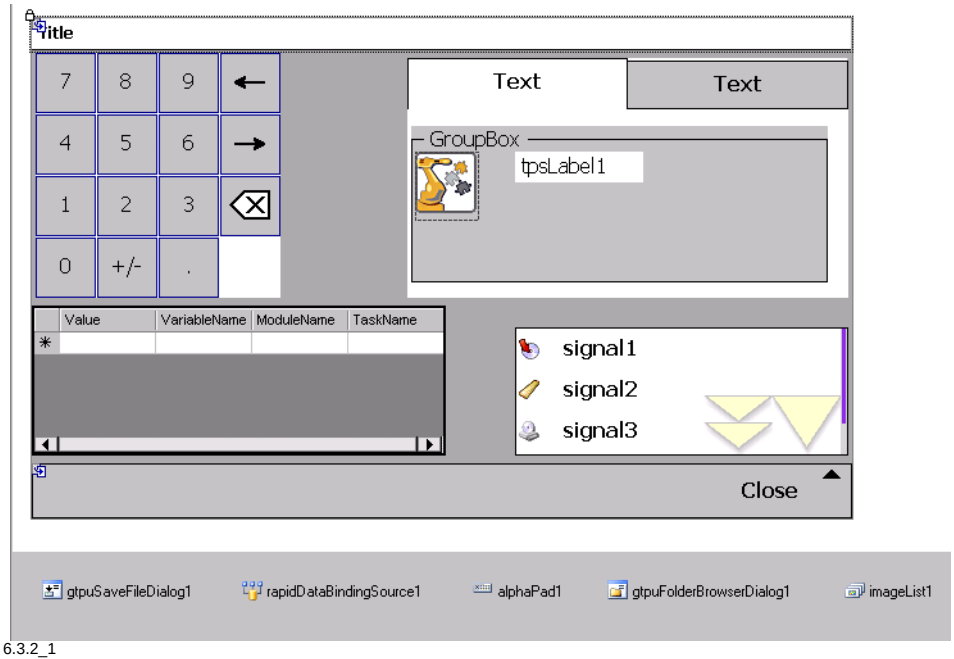
`ABB.Robotics.Tps.Windows.Forms.TpsControl` is the base type of all FlexPendant SDK controls. `TpsControl` in turn extends `System.Windows.Forms.UserControl`

`TpsControl` may be used as a container control of a FlexPendant view. It also has the default implementation of `Dispose`, which is called in order to free allocated memory for a control which should no longer be used.

You may wonder why this is necessary, as the .NET framework has a garbage collector, which should release the developer of the duty to free allocated memory. The answer is the garbage collector does not always reclaim memory which is no longer used. Therefore, if you program an application meant to execute around the clock on a device with limited memory, you are still responsible for freeing memory which is no longer used. Neglecting to do so will result in permanent memory leaks.

How to avoid memory leaks

Look at this figure showing a number of controls (both ABB and Microsoft) and learn the basics about memory management for GUI controls. It is not so complicated, the most important thing is not to forget that cleaning up is your responsibility!



6.3.2_1

Continues on next page

Continued

Learn and apply these the general rules for memory management of GUI controls.

- Controls with a graphical representation, e.g the ABB `Numpad`, `TabControl`, `GroupBox`, `TpsLabel`, `ListView` and the Microsoft `PicureBox` and `DataGrid` in the figure, are automatically added to the controls collection in `InitializeComponent. It may look like this:`
  `this.Controls.Add(this.numPad1);`
- If the figure above represents the first view of your application, controls with graphical representation will be disposed of by the base class of your view class when your application is shut down and the `Dispose` method is called by TAF. This happens when the following statement in your `Dispose` method is executed:
  `base.Dispose(disposing);`
- If, however, it represents a secondary view of your application (which is actually the case here, as you can tell from the close button on the command bar), you must call its `Dispose` method from the first view when it is closed. Its base class will then remove all controls that are part of its controls collection, like in the previous case.
- GUI controls that have no graphical representation, but are located in the *Components pane* under the form, e.g. `GTPUSaveFileDialog`, `RapidDataBindingSource`, `AlphaPad` etcetera, are NOT added to the controls collection by default. These are the ones that you need to be especially careful to remove, as no garbage collector will ever gather them. If you forget to explicitly call `Dispose` on such controls you will have caused a permanent memory leak. Carefully study the code example in the next section.

NOTE!

Microsoft and ABB controls behave in the same way. The Microsoft `ImageList` for example, which is commonly used in FlexPendant applications, has no graphical representation and must thus be explicitly removed by the application programmer.

Coding the Dispose method

The code example below shows how the `Dispose` method of the view shown in the figure above can be coded. All controls located in the components pane in the Designer must be explicitly removed by the programmer. Controls with a graphical representation will be removed when `Dispose` of the base class is executed.

```
protected override void Dispose(bool disposing)
{
  if (!IsDisposed)
  {
     try
     {
        if (disposing)
        {
              //Removes SaveFile dialog
           if(this.gtpuSaveFileDialog1 != null)
           {
              this.gtpuSaveFileDialog1.Dispose();
              this.gtpuSaveFileDialog1 = null;
```

Continues on next page

Continued

```
            }
               //Removes RapidDataBindingSource
            if(this.rapidDataBindingSource1 != null)
            {
               this.rapidDataBindingSource1.Dispose();
               this.rapidDataBindingSource1 = null
            }
               //Removes Alphapad
            if(this.alphaPad1 != null)
            {
               this.alphaPad1.Dispose();
               this.alphaPad1 = null
            }
               //Removes FolderBrowserDialog
            if(this.gtpuFolderBrowserDialog1 != null)
            {
               this.gtpuFolderBrowserDialog1.Dispose();
               this.gtpuFolderBrowserDialog1 = null
            }
               //Removes ImageList
               if(this.imageList1 != null)
            {
               this.imageList1.Dispose();
               this.imageList1 = null
            }
         }
      }
      finally
      {
          //Removes all controls added to the controls collection
          base.Dispose(disposing);
      }
    }
  }
```

Finally, as this is a secondary view, we should call its `Dispose` method from the first view when it is closed down.

```
//This code is executed by the first view when the secondary view
     is closed
void form2_Closed(object sender, EventArgs e)
{
  if(form2 != null)
  {
     form2.Dispose();
     form2 = null;
  }
}
```

Continues on next page

Continued

CAUTION!

If you forget to call `Dispose` on controls that are not part of the control collection of the class there will be memory leaks. This may cause the FlexPendant to run completely out of memory and crash. Usually, this will not happen when you try out the functionality of your application, but when it is executed and used continuously during production. To verify that a GUI control is really disposed of, you may set up a subscription to its Disposed event for example, and verify that it is triggered when you close down the view.

CAUTION!

All objects accessing robot controller services, i.e. unmanaged resources, must also be removed by the application programmer. See *Memory management on page 184* for further information.

Freeing allocated memory for a GUI control

You are recommended to remove a GUI control in the`Dispose` method of the class that created it. If the control belongs to the first view of your application, it will be disposed of when TAF calls `Dispose` at application shut down. If it belongs to a secondary view, you are responsible for disposing of the secondary view and its controls.

```
C#:
if (this.controlX != null)
{
   controlX.Dispose();
   controlX = null;
 }
base.Dispose(disposing);

VB:
  If disposing Then
     If Not controlX Is Nothing Then
        controlX.Dispose()
        controlX = Nothing
     End If
  End If
  MyBase.Dispose(disposing)
```

NOTE!

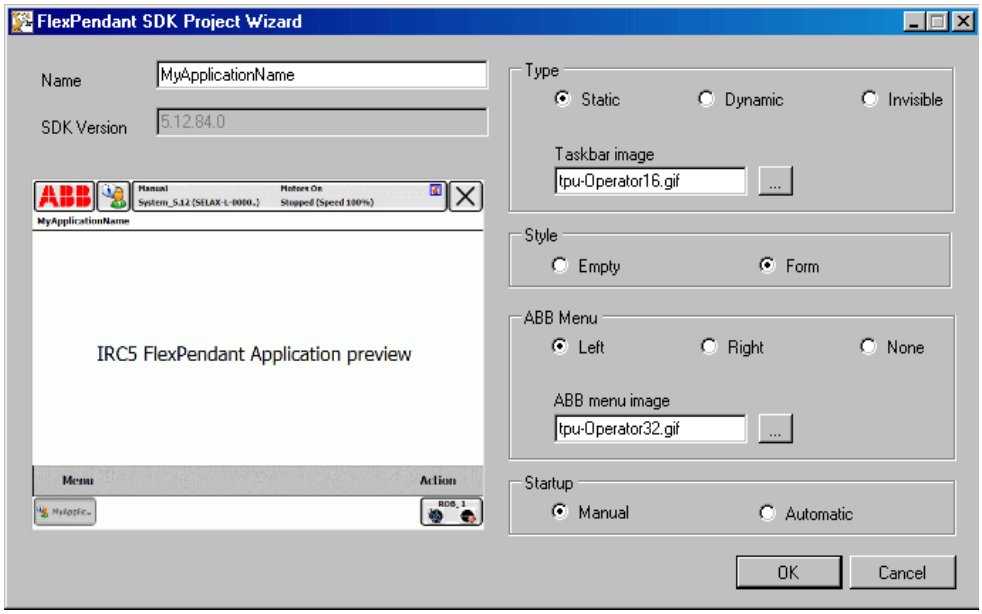
When the last statement in the code example above is executed the base class will call `Dispose` on all controls added to the controls collection in `InitializeComponent`. This means that you do not need to call `Dispose` on such controls.

5.3.3. Container style

Overview

The container control of a FlexPendant application is 640x390 pixels, which is the exact size of the FlexPendant screen dedicated for a custom view. This section details the two different styles that can be used for a FlexPendant view: **Empty** and **Form**.

You select which one to use in the **Style** area of the **FlexPendant SDK Project Wizard**. To the left in the wizard, a preview helps you see the difference between the two styles.



6.2.2_1

Here **Form** is selected. The preview illustration shows a `TpsForm`. It has a `CommandBar` at the bottom and a `TpsCtrlTitleBar` at the top.

NOTE!

When adding a new view to an existing FlexPendant project, you choose which control to use in the **Add New Item** dialog box. See *Adding a view to a custom application on page 99* for information on how to add a new view to an existing project.

Empty or Form?

Your choice in the wizard determines the type that your view class will inherit. If you change your mind about this, it is easy to make the change directly in the auto generated code.

Empty

This is a plain `TpsControl`. It has neither a `CommandBar`, nor a `TpsCtrlTitleBar`.

The code definition of the class will look like this:

VB:

```
Public Class TpsViewMyApp    Inherits TpsControl    Implements
ITpsViewSetup, ITpsViewActivation
```

Continues on next page

5.3.3. Container style

Continued

Form

This is a `TpsForm`, which is a specialized `TpsControl`.

The code definition of the class will look like this:

VB:

```
Public Class TpsViewMyApp    Inherits TpsForm    Implements
ITpsViewSetup, ITpsViewActivation
```


Continues on next page

Continued

NOTE!

Often **Empty** is chosen as the first view of an FP SDK application. For any secondary view, which is to be opened from the first one, **Form** is the recommended container control.

NOTE!

When selecting **Form** you might wonder why the command bar is not visible in the designer.

The answer is that it remains invisible until a `MenuItem` has been added to it. This is done in the **Properties** window of the `TpsForm`, by applying the property **MainMenu**.

TIP!

If you select the `TitlePanel` or the `CommandBar` of a `TpsForm` you will see that the **Properties** window is disabled. If you want to change the text of the title panel, or add menu items to the command bar, for example, you must select `TpsForm` by clicking somewhere in the middle and then modify its **Text** or **MainMenu** property.

How to build the command bar

The command bar is either ready-made, as for `TpsForm`, or used as a separate control available from the toolbox. If the command bar is ready-made, the design of it is done by modifying the **MainMenu** property of `TpsForm`. If used as a separate control, you design it by modifying its own properties. Apart from this, there is no difference in how you control the design.

An important thing to remember is that you have to manually add the code controlling what will happen when the user presses a menu item, as there are no event properties available in its **Properties** window.

See *Command bar on page 101* for further information on how to implement the control and its event handlers.

CAUTION!

The command bar of the *first* view of a custom application should not have a **Close** button. The reason is that all custom application must be closed down by TAF, which happens when the user presses the close [x] button in the upper right corner of the FlexPendant display. (See*Definitions on page 18* for a definition of TAF or *Understanding FlexPendant application life cycle on page 52* for further information on TAF.)

Adding a view to a custom application

To add another view to the FlexPendant project, you do not need to start a new project. This is how to proceed:

Step	Action
1	Right-click the project node in the **Solution Explorer**, point to **Add** and select **New Item**.
2	Select one of the FlexPendant container controls available in the dialog box. Normally it will be a *Form*. **Note!** If you select **Empty** the code you write to open it from the first view is a bit different than if you are using **Form** as a secondary view.

Continues on next page

Continued

This way all of your views make up a single dll, which is convenient. Most of the time there is no point in dividing the application into several dlls.

Launching the view

The code for launching the secondary view is simple. It may look like this:

```
//declare the secondary view as a private member
private View2 _view2;


//create the view and add subscription to its closed event, then launch view
private void button1_Click(object sender, EventArgs e)
{
    this._view2 = new View2();
    this._view2.Closed += new EventHandler(view2_Closed);
    this._view2.ShowMe(this);
}
//dispose of the view when it has been closed
void view2_Closed(object sender, EventArgs e)
{
    this._view2.Closed -= new EventHandler(view2_Closed);
    this._view2.Dispose();
    this._view2 = null;
}
```

NOTE!

Make sure that there is a natural way for the user to get back to the preceding view. The most intuitive way of doing this is to press a **Close** button or an **OK** or **Cancel** button on the command bar. If the secondary view is a `TpsForm`, this event handler closes it:

```
void menuItem1_Click(object sender, EventArgs e){ this.CloseMe();}
```

NOTE!

If you are using **Empty** as a secondary container control the code you write to launch it is:

```
this._view2.Show();
```

It must be added to the first view's control collection before it can be shown, just like an ordinary .NET control.

5.3.4. Command bar

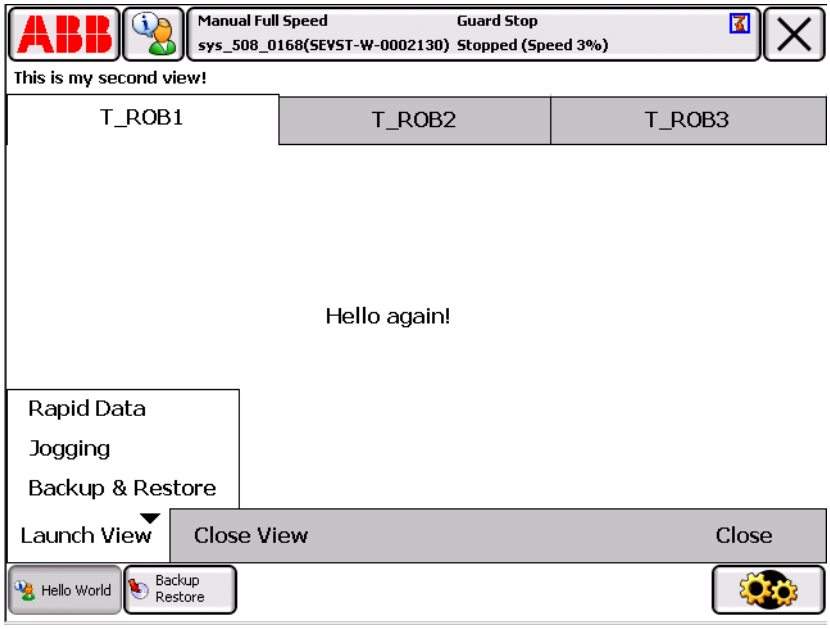
Overview

For the end-user the command bar is a central FlexPendant control and an important means of interacting with the system. As described in *How to build the command bar on page 99* it can be ready-made or added to the container control from the VS Toolbox.

How to add menu items

If you use the `Form` container a command bar is built-in, but not visible until you add a MenuItem collection to the **MainMenu** property of `TpsForm`. How to do this is shown in step 5 - 6 of *Hands on - step 2 on page 89*.

If you use **Empty** as container the command bar needs to be added from the Toolbox. In this case too, the commands are added as a collection of menu items.



6.3.4_1

The command bar in this figure has a collection of three menu items: Launch View, *Close View* and *Close*. Moreover, the *Launch View* menu item has its own collection of menu items: *Rapid Data*, *Jogging* and *Backup Restore*.

In the figure, as shown by the task bar, the *Backup Restore* view has been opened. It has been done by clicking the *Backup Restore* menu item. The user has then returned to the **Hello World** application by using the task bar. If *Close View* is now pressed, the *Backup Restore* view will close down. The next section shows how event handlers for this command bar can be implemented.

Continues on next page

Continued

How to add menu item event handling

Subscriptions to menu item events and event handling cannot be added using the design support, but have to be implemented by hand in the source file. The code example below shows how the command bar in the figure above can be implemented.

```
private ABB.Robotics.Tps.Taf.ITpsViewLaunchServices _launchService;
private object _cookie;

//Constructor, takes ITpsViewLaunchServices object as in parameter
public View2(ABB.Robotics.Tps.Taf.ITpsViewLaunchServices lS)
{
    _launchService = lS;
    InitializeComponent();
    this.menuItem3.Click += new EventHandler(menuItem3_Click);
    this.menuItem4.Click += new EventHandler(menuItem4_Click);
    this.menuItem5.Click += new EventHandler(menuItem5_Click);
    this.menuItem6.Click += new EventHandler(menuItem6_Click);
    this.menuItem7.Click += new EventHandler(menuItem7_Click);
}
//Event handlers for all menu item commands
void menuItem3_Click(object sender, EventArgs e)
{   //opens RapidData view, may open several instances as specified by 3:rd argument
    this._launchService.LaunchView(FpStandardView.RapidData, null, true, out this._cookie);
}
void menuItem4_Click(object sender, EventArgs e)
{  //opens Jogging view
   this._launchService.LaunchView(FpStandardView.Jogging, null, false, out this._cookie);
}
void menuItem5_Click(object sender, EventArgs e)
{  //opens BackUp & Restore view
    this._launchService.LaunchView(FpStandardView.BackUpRestore, null, false, out this._cookie
}
void menuItem6_Click(object sender, EventArgs e)
{
    if (_cookie != null) //if this app. has opened a standard view, this code closes it
    {
        this._launchService.CloseView(this._cookie);
        this._cookie = null;
    }
}
void menuItem7_Click(object sender, EventArgs e)
{
    this.CloseMe();  //closes secondary view
}
```

6.3.4_2

NOTE!

To launch a standard view the `ITpsViewLaunchServices` object, which can be saved by the `TpsView` class in its `Install` method, is needed. See *Using launch service on page 128*.

5.3.5. FlexPendant fonts

Overview

On the FlexPendant, `TpsFont` is used instead of standard Windows fonts. By using TpsFont you save limited device memory, as static references instead of new font instances will be used.

TpsFont

`Tahoma` is the default font of the FlexPendant. It is also the font usually used internally by `TpsFont`. A number of different font sizes are pre-allocated and can be reused. You are recommended to use TpsFont instead of creating new font instances for both ABB and .NET UI controls.

NOTE!

To be able to use Chinese or another language with non-western characters, you must use the FlexPendant font, `TpsFont`, for any UI controls. It internally checks what language is currently active on the FlexPendant and uses the correct font for that language.

NOTE!

If you use other fonts than available on the FlexPendant, i.e. Tahoma and Courier New, the application will use Tahoma instead of the intended one.

5.3.6. The use of icons

Overview

It is easy to display icons, images and photos on the FlexPendant touch screen. Utilizing this possibility is recommended, as many people find it easier to read pictures than text. Intuitive pictures can also be widely understood, which is a real advantage in a multinational setting.

NOTE!

The operating system of the first generation FlexPendant device (SX TPU 1) does not support images of more than 256 colors.

CAUTION!

Be aware of the limited memory resources of the FlexPendant device. Do not use larger images than necessary. Also see *How large can a custom application be? on page 182*

FlexPendant controls with images

Several FlexPendant controls support images, which are imported as bmp, jpg, gif or ico files when an **Image**, **Icon** or **BackgroundImage** property is set. These are some controls to be used with pictures:

- `MenuItem`
- `Button`
- `TpsCtrlTitlePanel`
- `TabPage`
- `ListViewItem`
- `TpsForm`
- `FPRapidData`, `FpToolCalibration`, `FpWorkObjectCalibration`
- `GTPUSaveFileDialog`

PictureBox and ImageList

There are also some Windows controls that can be used to display images.
`Windows.Forms.PictureBox` is one such control. It can display graphics from a bitmap (.bmp), icon (.ico), JPEG or GIF file. You set the **Image** property to the preferred image either at design time or at run time.

`Windows.Forms.ImageList` can manage a collection of **Image** objects. The ImageList is typically used by another control, such as the `ListView` or the `TabControl`. You add bitmaps or icons to the `ImageList`, and the other control can choose an image from the `ImageList` index. How to use images for a `TabControl` is detailed in *How to add tab images on page 106*.

Continues on next page

The TpsIcon class

The `TpsIcon` class offers static references to a number of icons used in the standard FlexPendant applications. You may want to use some of these icons if they suit the application need. `TpsIcon` belongs to the namespace `ABB.Robotics.Tps.Drawing`.

TIP!

In the *FP SDK Reference*, click the **Search** tab and search for *TpsIcon Members* to get a short description of the ABB icons that are available as static properties.

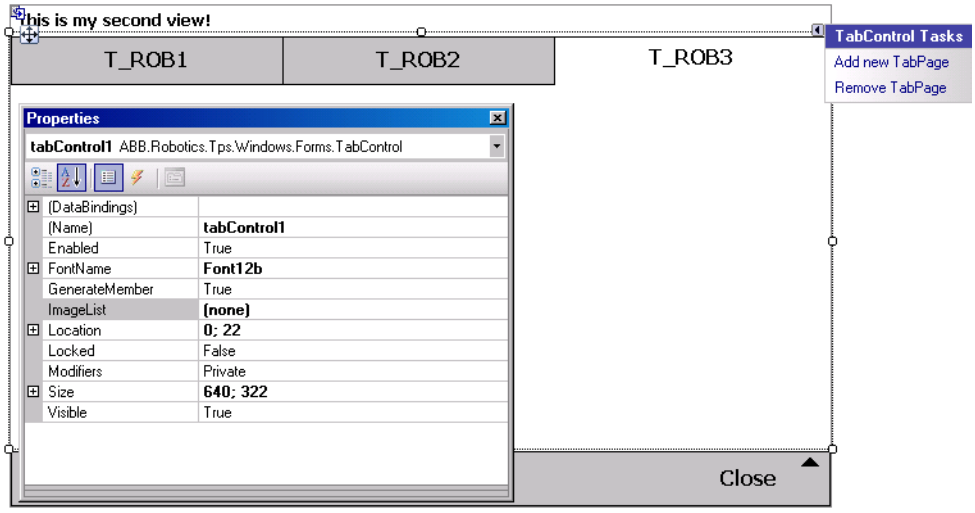
5.3.7. TabControl

Overview

The `ABB.Robotics.Tps.TabControl` is used and implemented much in the same way as a regular `System.Windows.Forms.TabControl`. The possibility to use tab icons, however, is an additional feature of the ABB `TabControl`.

Illustration

The figure below shows a `TabControl` and its **Properties** window. To add or remove tab pages you click the little arrow in the top right corner of the control or click **Add new TabPage** at the bottom of the **Properties** window.



6.2.4_1

NOTE!

The ABB `TabControl` has no **TabPages** property. Depending on where in the TabControl you click, you select either the entire `TabControl` or one of the `TabPages`. To view the **Properties** window for the T_ROB1 `TabPage` in the figure, you would have to click the T_ROB1 tab and then the T_ROB1 page below the tab.

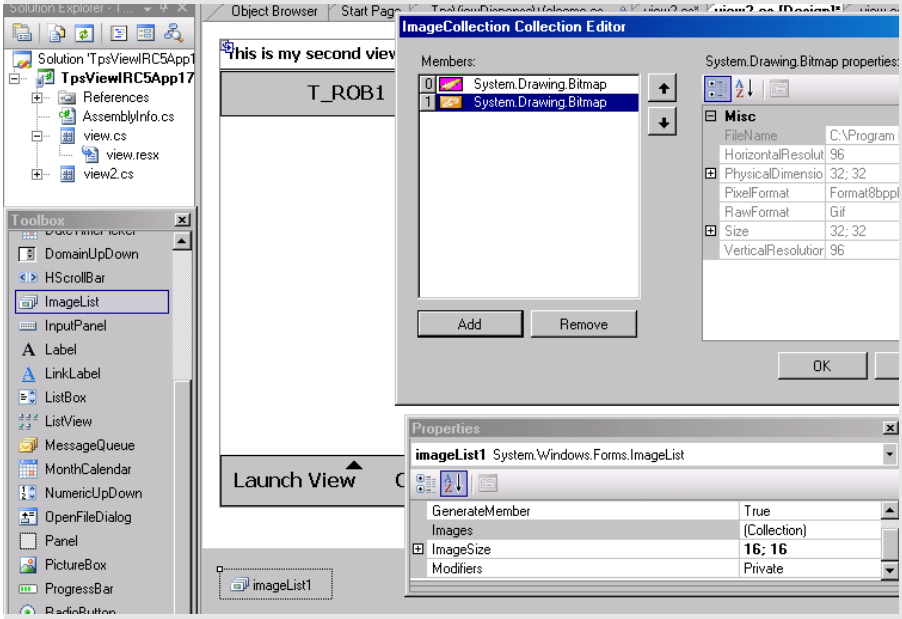
How to add tab images

This is how you use the **ImageList** property to add icons to the tabs of the `TabControl`:

Step	Action
1	Drag a `System.Windows.Forms.ImageList` to the **Designer** area. As you will notice it has no visual representation, but is placed on the components pane below the container control.
2	Display the **Properties** window of the ImageList.
3	Click the property **Images** and then the browse button, which will appear.

Continues on next page

Continued

Step	Action
4	Click **Add** and import an image. Repeat until you have added the icons your TabControl needs.  6.2.4_2
5	Now display the **Properties** window of the `TabControl` and select your `ImageList` as the **ImageList** property.
6	Select one of the tab pages and set the property **ImageIndex**, which defines which of the images in the `ImageList` is to be used for the tab. Repeat this procedure for all tab pages.

How to add an event handler using the Properties window

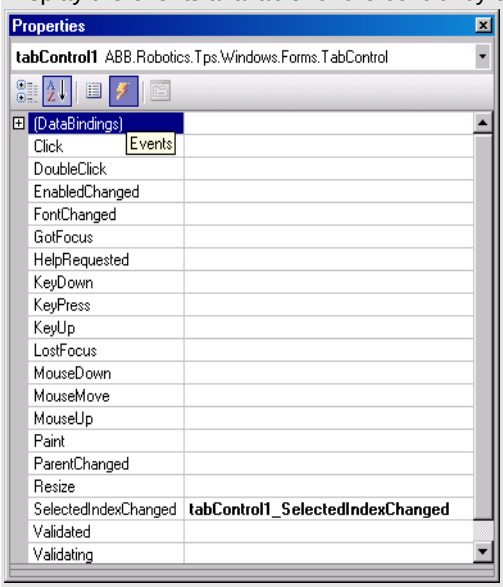
It is easy to attach an event handler to a control by using the Properties window. This procedure shows how to make something happen when the user clicks a tab to select another tab page:

Step	Action
1	In the **Designer**, select the `TabControl` and display the **Properties** window.

Continues on next page

5.3.7. TabControl

Continued

Step	Action
2	Display the events available for the control by clicking the **Events** button.  6.3.4_3
3	Select the **SelectedIndexChanged** event and double click. A method name is now auto generated for the **SelectedIndexChanged** event. The code view takes focus, the cursor placed inside the generated event handler.
4	Add code to make something happen. For now this code will do: `this.Text = "TabControl notifies change of tab pages";` This will change the title text of the TpsForm when a tab is clicked.
5	Test the functionality by building, deploying and starting the application. See step 5-8 in *Hands on - Hello world on page 86*.

NOTE!

The subscription to the event has been done under the hood, and you do not need to bother about it. If you use C# you will notice that some code adding the subscription has been inserted in the `InitializeComponent` method:

```
this.tabControl1.SelectedIndexChanged += new
System.EventHandler(this.tabControl1_SelectedIndexChanged);
```

Disposing TabControl

You do not need to explicitly call `Dispose` on the `TabControl` object. The reason is that `InitializeComponent` adds the tab pages to the controls collection of the `TabControl`, and the `TabControl` itself to `the control collection of the container class.` `this.Controls.Add(this.tabControl1);`

The `TabControl` is thus removed when the `Dispose` method of your container class calls `Dispose` on its base class like this: `base.Dispose(disposing);`

5.3.8. Button, TextBox and ComboBox

Overview

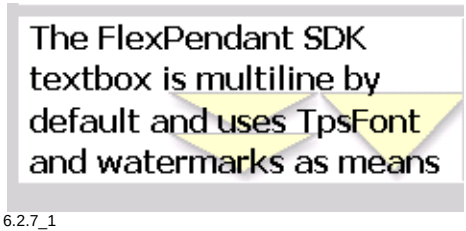
The FlexPendant button, text box and combo box controls are not very different from the equivalent Windows controls.

Using Button

A common size of a FlexPendant `Button` is 120 * 45 pixels; it can be a lot smaller than its default size. As opposed to a Windows button it can display an image. As pointed out in *Illustration on page 106* you use the property **BackgroundImage** and browse for the image to be displayed.

Using TextBox

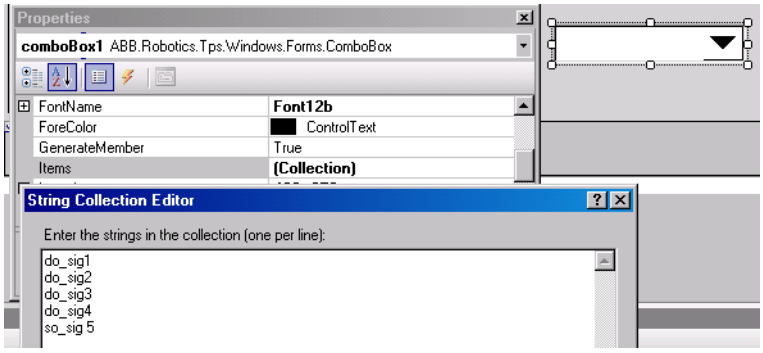
The `TextBox` control is multiline by default and uses the typical FlexPendant scroll bars as shown in the figure below.



6.2.7_1

Using ComboBox

The `ComboBox` is very similar to the `System.Windows.Forms.ComboBox`, but is visually improved to suit the FlexPendant touch screen. It can be statically populated using the **Items** property as illustrated by the figure. If you wish to populate it dynamically, you need to write code for it.



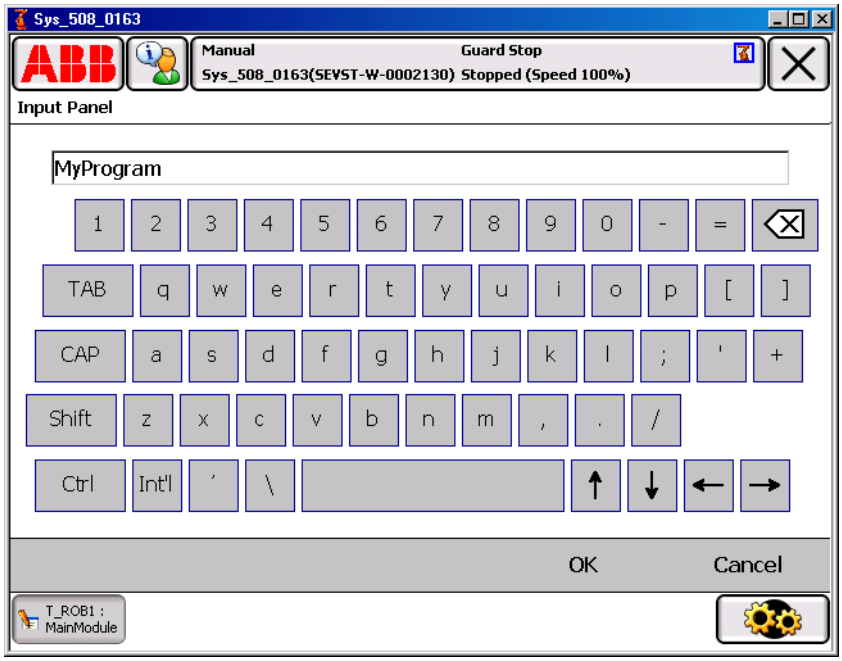
6.2.7_2

`SelectedIndexChanged` is the most commonly used among the events. It occurs when the user selects another item from the list.

5.3.9. AlphaPad

Overview

The `AlphaPad` control is a virtual keyboard, which enables the user to enter strings. It covers the whole FlexPendant display. The figure shows its appearance in run time.



6.2.10_1

The `AlphaPad` has no graphical representation in design-time. When dragged from the **Toolbox** to the **Designer** it is placed in the components pane, as shown by the figure in the next section. Code is of course generated for it, just like for controls that are visually laid out on the form in design-time.

Continues on next page

Launching the AlphaPad

You need to write some code to launch the virtual keyboard. This is how you do it:

C#:

```
this.alphaPad1.ShowMe(this);
```

VB:

```
Me.AlphaPad1.ShowMe(Me)
```

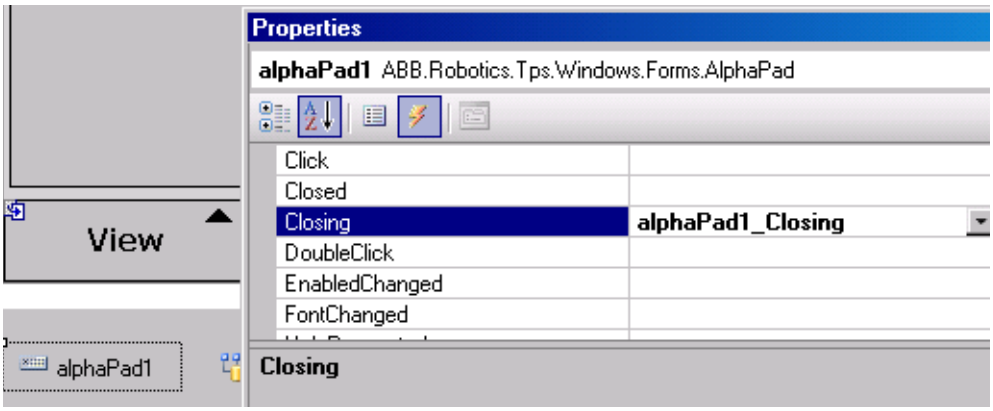
NOTE!

The code for launching the `AlphaPad` should be executed by an appropriate event handler.

Adding event handlers

The **Properties** window can be used to program `AlphaPad` event handlers. You double click the event you want to respond to. This takes you to the code editor, the cursor placed inside the auto generated event handler, where you can write code to save user input to a member variable for example:

```
if (alphaPad1.SelectedText != string.Empty)
{
  _userInput = alphaPad1.SelectedText;
}
```



6.2.10_2

Validating the result at the Closing event

The `Closing` event is generated when the **OK** or **Cancel** button of the `AlphaPad` is pressed. You can use this event to validate the string entered by the user. You can set the **Cancel** property of the event argument to true if you want the AlphaPad to remain open until a valid input value has been entered:

VB:

```
Private Sub NamePad_Closing(sender As Object, e As
     System.ComponentModel.CancelEventArgs) Handles
     NamePad.Closing
  If NamePad.SelectedText.CompareTo("No Name") = 0 &&
       NamePad.DialogResult =
       System.Windows.Forms.DialogResult.OK Then
    e.Cancel = True
```

Continues on next page

5.3.9. AlphaPad

Continued

```
    End If
  End Sub
```

C#:

```
  private void namePad_Closing(object sender,
       System.ComponentModel.CancelEventArgs e)
  {
    if ((this.namePad.SelectedText.CompareTo("No Name") == 0) &&
         (this.namePad.DialogResult ==
         System.Windows.Forms.DialogResult.OK))
    {
      e.Cancel = true;
    }
  }
```

Using the result at the Closed event

The Closed event has to be caught by an event handler, as the object cannot be disposed of until it has been closed. The result may be retrieved in this event handler or in the Closing event handler. First check that OK and not Cancel was pressed, then retrieve user input. Finally the AlphaPad should be disposed of.VB:

```
  Private Sub NamePad_Closed(sender As Object, e As EventArgs)
       Handles NamePad.Closed
    If NamePad.DialogResult = Windows.Forms.DialogResult.OK Then
      Me.answer = NamePad.SelectedText
    End IfNamePad.Dispose()
  End Sub
```

C#:

```
  private void namePad_Closed(object sender, EventArgs e)
  {
    if (this.namePad.DialogResult ==
         System.Windows.Forms.DialogResult.OK)
    {
      this.answer = this.namePad.SelectedText;
    }
    this.alphaPad1.Dispose();
  }
```


Continues on next page

Removing the AlphaPad control

The `AlphaPad` is NOT added to the control collection, and will therefore NOT be disposed of by the base class of its container.

You are responsible for explicitly calling its `Dispose` method when it is no longer used. In the example above, this is done at the `Closed` event. This implies that a new `AlphaPad` instance is created the next time its launch event is triggered.

Another way of dealing with this is to let the instance created by the Designer live until its container view is closed. This alternative means destroying it in the `Dispose` method of the container class:

```
this.alphaPad1.Dispose();
```

CAUTION!

If you forget to call `Dispose` on an AlphaPad control you will have a memory leak. For further information see *GUI controls and memory management on page 93*.

5.3.10. ListView

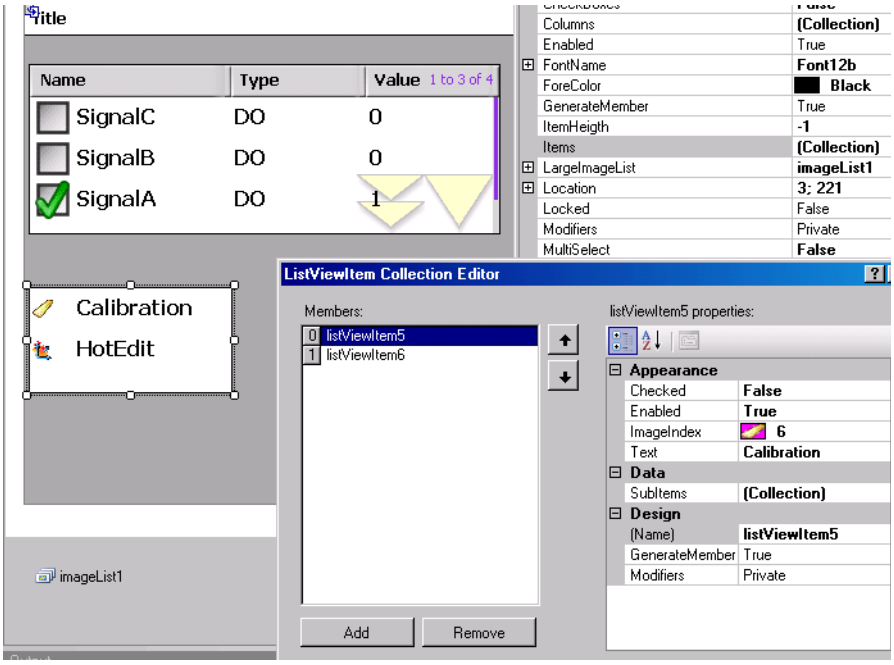
Overview

The `ABB.Robotics.Tps.Windows.Forms.ListView` is very similar to a standard .Net `ListView`. The main difference is that its appearance is somewhat adapted to the use of a touch screen, as can be seen in the figure.

You can use the `ListView` control in a variety of ways. Usually you use it to display a list of items with item text and, optionally, an icon to identify the type of item. By using the **CheckBoxes** property it is also possible to have a check box appear next to each item in the control, allowing users to check the items they want to perform an action on.

Illustration

The figure shows the Designer. You see two different ABB `ListView` lists, a standard .Net `ImageList` in the components pane under the lists and the **Properties** window for the *Calibration, HotEdit* list.



6.3.10_1

Using properties to control appearance

The first list has **CheckBoxes** set to true. To have columns displayed **View** must be set to **Details**. The columns are created by selecting **Columns** and adding any number of columns in the **ColumnHeader Collection Editor**. **Scrollable** has been set to true to enable the user to scroll the list to see all items. To have the current number of list items displayed

Continues on next page

Continued

ShowNumberOfItems has been set to true. By using **Sorting** the list items can be alphabetically sorted, in this case in **Descending** order. The list is statically populated, using the **Items** property.

The **View** property of the other list is set to **LargeIcon** instead of **Details**. To display icons a standard .Net **ImageList** is used. You first launch the **Properties** window for the **ImageList** and add the images you want to use. At the **LargeImageList** property of the `ListView` you select the image list, and for each **Item** you select the **ImageIndex** to use, as is shown in the **ListViewItem Collection Editor**.

Using the **Properties** window you can also add event handling. The most commonly used event is **SelectedIndexChanged**, which is triggered when a new list item is selected.

The **Properties** window can be used to statically populate the list. Usually, however, lists are populated dynamically. In *Getting signals using SignalFilter on page 167* there is a code example, which shows how I/O signals are dynamically displayed in an ABB `ListView`.

ABB specific properties

These properties are specific for the ABB `ListView`:

Property	Usage
MultiSelect	Specifies whether multiple items in the control can be selected. The default is false.
Scrollable	Specifies whether a scroll bar is added to the control when there is not enough room to display all items. The default is true. **Note!** The typical FlexPendant scroll bars are used by the first list in the figure above.
SelectionEnabledOver-Scrollbuttons	Specifies whether a touch inside the scroll region should select an item or scroll the list. The default is false.
ShowNumberOfItems	Specifies whether the current number of items in the list is displayed. The default is true. Scroll bars should be added to the control to allow the user to see all the items. If the list is not scrollable the default value is false.
ShowSelection	Specifies whether selection is shown or not. The default is false.

5.3.11. CompactAlphaPad and NumPad

Using CompactAlphaPad

The `CompactAlphaPad` control is used to enable user input. The input can be in the form of capital letters or numbers depending on the property `ActiveCharPanel`, which can be set to `Characters` or `Numeric`. It has a fixed size, big enough for the use of a finger to press its keys. There are properties available to define whether the numerical panel should support numerical operands, special characters, space and comma. All these properties are set to true in the figure:

6.3.11_1

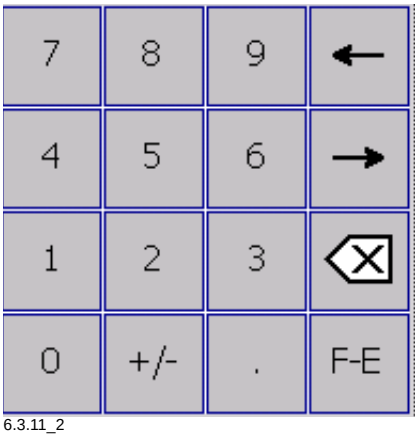
In order for your application to use the user input, you must connect to a text box, which displays the text or figures entered. This connection has to be coded manually, for example in the constructor, after the call to `InitializeComponent`, e.g.:

```
this.compactAlphaPad1.Target = this.textBox1;
```

Using NumPad

The `NumPad` is very similar to the CompactAlphaPad. You have to create a `Target` to be able to use the input, e.g.:

```
this.numPad1.Target = this.textBox2;
```



6.3.11_2

5.3.12. GTPUMessageBox

Overview

`GTPUMessageBox` is the message box control to be used by FlexPendant applications. You should not use the standard `.NET MessageBox`.

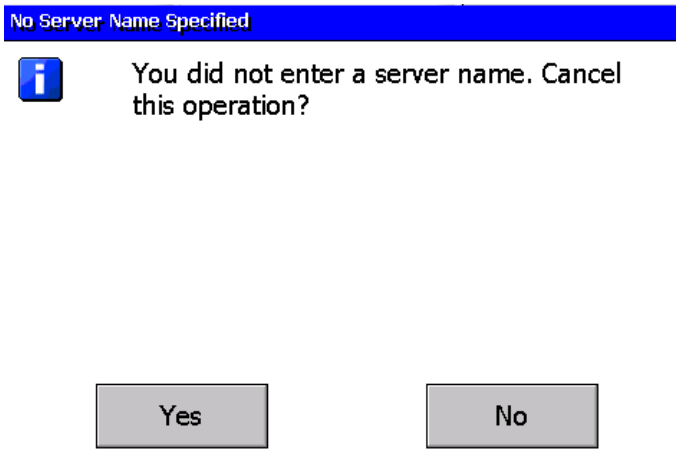
Design issues

The `GTPUMessageBox.Show` arguments include an owner argument, which is the control that displays the `MessageBox`, and a callback, which is a `MessageBoxEventHandler` called when the MessageBox is closed.

Except this, it is used in the same way as the regular Windows `MessageBox`. It is used together with the regular Windows `MessageBoxButtons` and `MessageBoxIcon`.

Simple code example

The figure and code below show how to display a simple message on the FlexPendant using the `GTPUMessageBox`.



6.2.6_1

```
string message = "You did not enter a server name. Cancel this
     operation?";
string caption = "No Server Name Specified";
GTPUMessageBox.Show(this, null, message, caption,
     System.Windows.Forms.MessageBoxIcon.Question,
     System.Windows.Forms.MessageBoxButtons.YesNo);
```


Continues on next page

Continued

NOTE!

Sometimes it can be a bit tricky to get the first argument right. The owner might be this, `this.Parent` or even `this.Parent.Parent`.

Using a callback

The logics in your program determines if there is any need for a callback. In the previous example the callback argument was `null`.The message box will just close down after the user has answered the question, no matter if the answer is **Yes** or **No**.

If we think about the message, it seems likely however, that something should happen if the user presses **No**. Let us change the implementation and use a callback for this purpose:

```
GTPUMessageBox.Show(this, new
      MessageBoxEventHandler(OnServerMessageClosed), message,
      caption, System.Windows.Forms.MessageBoxIcon.Question,
      System.Windows.Forms.MessageBoxButtons.YesNo);


//implement callback
private void OnServerMessageClosed(object sender,
      ABB.Robotics.Tps.Windows.Forms.MessageBoxEventArgs e)
{
  if(e.DialogResult == System.Windows.Forms.DialogResult.No)
  {
     //Use default server...
  }
}
```

If we really do not want to take action depending on how the user responds, it makes more sense to use **OK** instead of **Yes** and **No** as message box buttons, e.g:

```
GTPUMessageBox.Show(this, null, "You are not allowed to perform this
operation, talk to your system administrator if you need access",
"User Authorization System", MessageBoxIcon.Hand,
MessageBoxButtons.OK);
```

NOTE!

The last example uses a better message box title then the previous ones. The title preferably should tell the user from which part of the system the message originates.

5.3.13. GTPUFileDialog

Overview

The FlexPendant SDK provides a number of file dialogs used by the end-user to interact with the file system of the robot controller. These controls all inherit the abstract class `GTPUFileDialog` and belong to the namespace `ABB.Robotics.Tps.Windows.Forms`.

File dialog types

There are three different types of file dialog controls:

Use...	when you want to...
`GTPUOpenFileDialog`	Enable the user to specify a file to be retrieved from the controller file system.
`GTPUSaveFileDialog`	Enable the user to specify a file to be saved to the controller file system.
`GTPUFolderBrowserDialog`	Enable the user to specify a folder on the controller file system.

NOTE!

The Open/Save/Browse file dialogs represent a convenient way for the user to specify folder and filename for a file operation. You should know, however, that the dialogs themselves do not perform any file operation, they only provide the controller file system path to be used by your application.

CAUTION!

When added to the VS Designer from the Toolbox, these controls will be placed in the components pane. They must be explicitly removed by a call to their `Dispose` method, e.g. in the `Dispose` method of the class that created them.

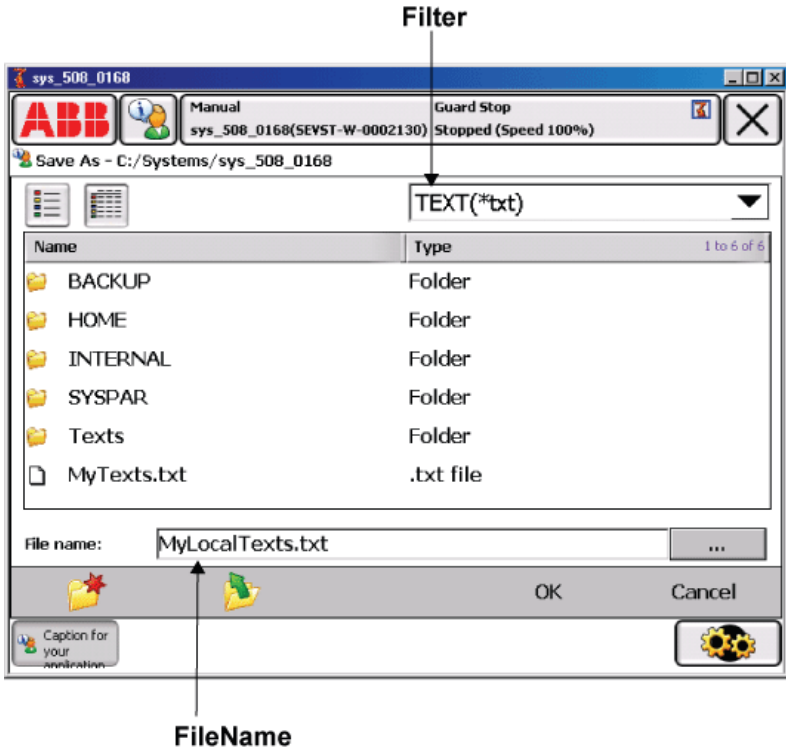
Illustration

Below is an illustration of the `GTPUSaveFileDialog`. The other file dialogs have almost the same appearance and the way they work is very similar. Using the icons of the command bar you create new folders and browse the controller file system. Together with the list and the

Continues on next page

Continued

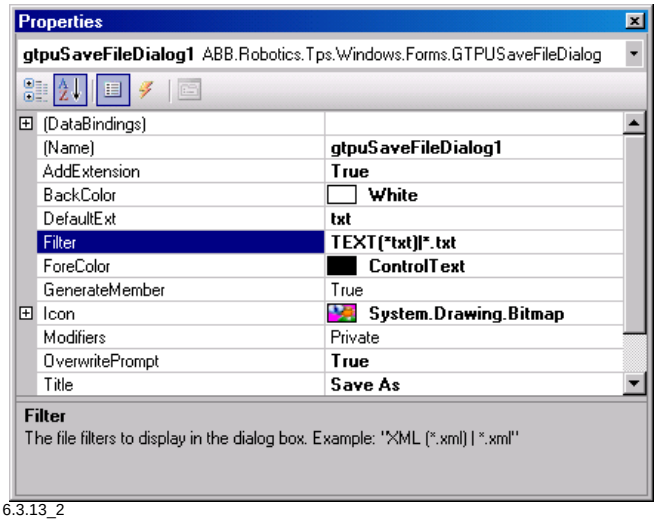
textbox, you specify a remote path. The **FileName** property should be read to retrieve this path. It returns the complete path to the controller file system, even though only the file name is visible in the **File name** textbox



6.3.13_1

Note! The button after the **File name** textbox opens the virtual keyboard, enabling the user to change the name of the file to be saved. The **Filter** property of the control is set to *TEXT(*.txt)|*.txt*. The first part of the setting is displayed in the filter textbox.

This is the Properties window of this dialog:



6.3.13_2

Continues on next page

Implementation details

The **Properties** window gives a clear description of a selected property, as can be seen in the figure above. Here only a few important properties will be detailed:

Property	Details
`Filter`	Carefully observe the string format when you set this property, e.g.: Program Files (*.pgf)|*.pgf The first part is displayed in the combo box and the second part is used by the SDK to retrieve the correct files to be displayed in the list. You can also specify several filters, which the user can choose from. Program Files (*.pgf)|*.pgf|All Files (*.*)|*.*;
FileName	Cannot be accessed in design-time by using the **Properties** window, but should be manually coded when the file dialog is launched if it is a save file dialog. When the dialog is closed, you should read this property. It holds the remote path and file name of the file to be opened, or the remote path and file name of the file to be saved. **Note!** Remember that *remote* refers to the controller file system and *local* to the FlexPendant file system.
`InitialDirectory`	Cannot be accessed in design-time by using the **Properties** window. Specifies the initial directory to be displayed by the file dialog box.

NOTE!

For your program to be able to use the controller path selected by the end user, you need to read the **FileName** property at the `Closing/Closed` event of the file dialog.

Example

This piece of code sets `InitialDirectory` and `FileName`, then sets up a subscription to the `Closed` event and finally displays the `GTPUSaveFileDialog`

```
saveFileDialog.InitialDirectory = initialDir;
saveFileDialog.FileName = programName;
saveFileDialog.Closed += new
     EventHandler(SaveProgram_FileDialog_EventHandler);
saveFileDialog.ShowMe(_parent);
```

The code of the `SaveProgram_FileDialog_EventHandler` method retrieves the specified path of the remote file system, including the file name, by reading the `FileName` property:

```
string remotePath = saveFileDialog.Filename;
```

NOTE!

The file has not yet been saved to the robot controller. To do that you should call `FileSystem.PutFile` using the retrieved path as the *remoteFile* argument. Likewise, to load a specified remote file to the FlexPendant file system you should use the retrieved path in the call to `FileSystem.GetFile`.

5.3.14. DataBinding of RAPID data and IO signals

What is databinding?

Databinding is the process of binding a property of a GUI control to a data source, so that the property automatically reflects the value of the data source. In .NET CF 2.0 this functionality was simplified thanks to the new `BindingSource` class. This class encapsulates the complexity related to setting up and managing databinding.

FlexPendant SDK classes to be used as binding sources

In the FlexPendant SDK there are two classes that inherit .NET `BindingSource`: `RapidDataBindingSource` and `SignalBindingSource`. These classes enable binding to RAPID data and IO signals respectively. They belong to the `ABB.Robotics.DataBinding` namespace and the assembly you need to reference is ABB.Robotics.DataBinding.dll.

RapidDataBindingSource

By using `RapidDataBindingSource` an automatic update of the bound GUI control takes place when the value of the specified RAPID data source in the controller has changed, or when the control is repainted.

SignalBindingSource

By using `SignalBindingSource` an automatic update of the bound GUI control takes place when the value of the specified IO signal in the controller has changed, or when the control is repainted.

NOTE!

It is only possible to use persistent RAPID data (PERS) as data source.

NOTE!

If you want to let users modify RAPID data, launching the standard FlexPendant application *Program Data* from your application is probably the best alternative. See *Using standard dialogs to modify data on page 131* for information about how to do it.

GUI example

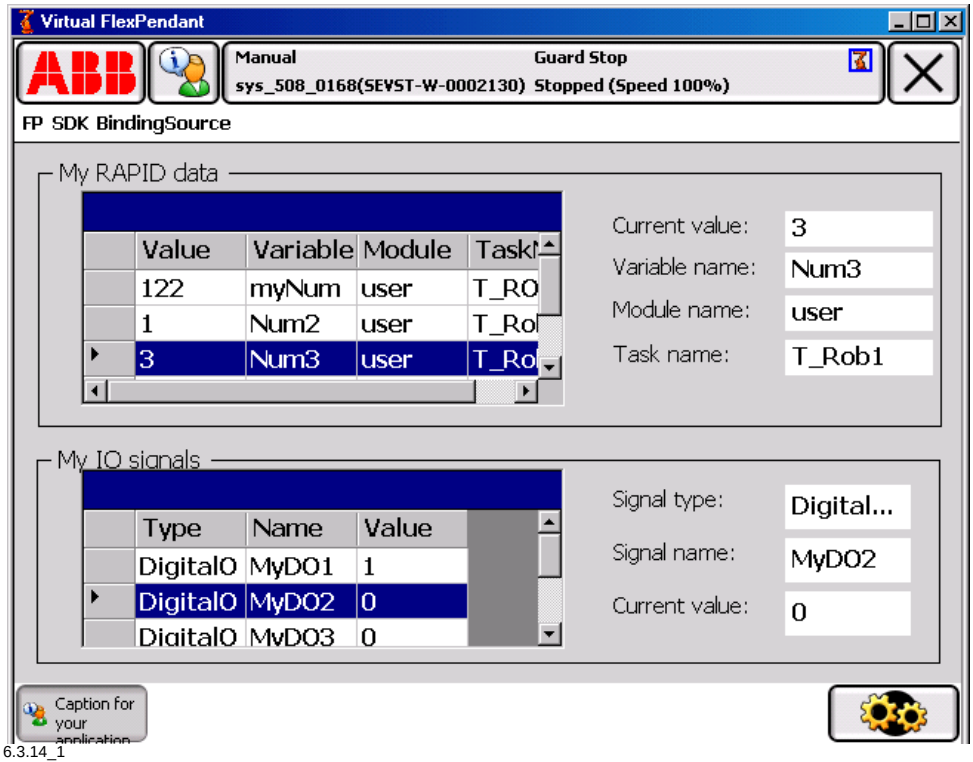
Many different controls can be used with binding sources, e.g. `TpsLabel`, `TextBox`, `ListView` etc. The figure below shows two `System.Windows.Forms.DataGrid` controls that each bind to several objects defined in a `RapidDataBindingSource` and a

Continues on next page

Continued

`SignalBindingSource` . The labels in each `GroupBox` are bound to the same sources. As you see the grid displays all objects defined in the `BindingSource` control, whereas each label displays the currently selected grid row.

When a signal or a data value changes this GUI is automatically updated with the new value. You do not have to write any code make this happen, as setting up subscriptions and updating the user interface is done by the underlying implementation.



6.3.14_1

Continues on next page

Continued

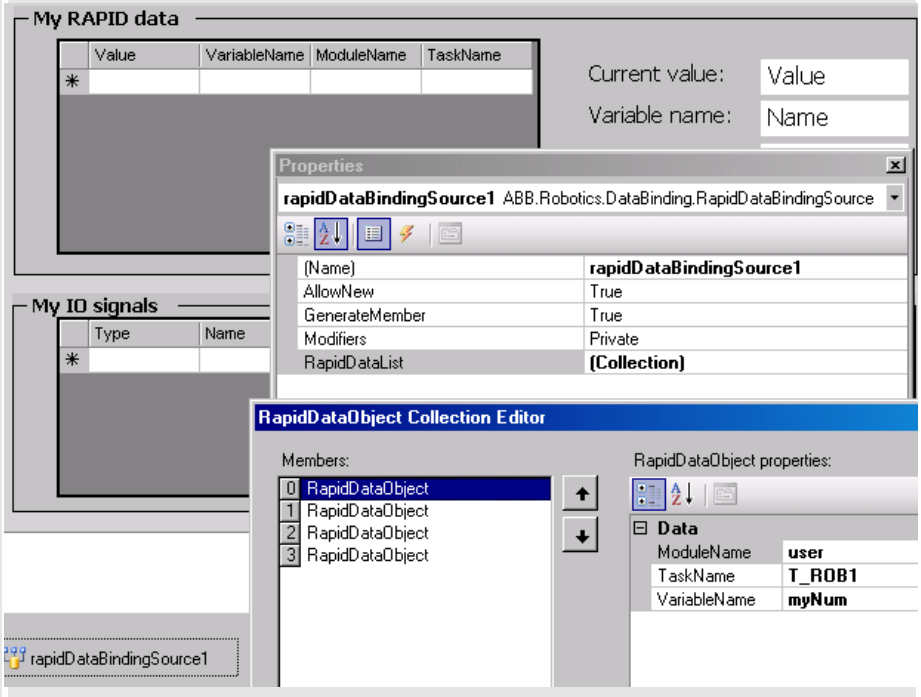
CAUTION!

To avoid any memory leaks an explicit call to the `Dispose` method of `BindingSource` controls must be made. However, the wrapper-objects `SignalObject` and `RapidDataObject` created for you are disposed of under the hood, so you do not need to worry about it.

How to use the VS designer for data binding

This section explains how to create the FlexPendant view shown in the previous section. First a `RapidDataBindingSource` control with bindings to specified RAPID data is created.Then the **DataBindings** property of a `TpsLabel` is used to bind the label to the binding source. Finally a standard .NET `DataGrid` is bound to the same binding source.

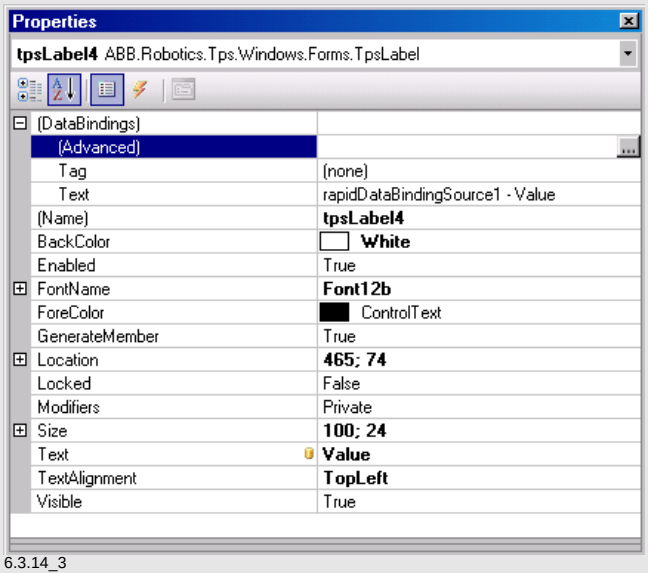
Step	Action
1.	Start by dragging a `RapidDataBindingSource` from the Toolbox to the Designer. It will be placed in the Components pane under the form. Open the **Properties** window and select the **RapidDataList** property to add the RAPID data you are interested in. For each new **RapidDataObject** member you must specify module name, task name and name of the persistent RAPID data to bind.



Continues on next page

Continued

Step	Action
2.	The next step is to open the **Properties** window for the label that is to display the value of the RAPID data. Expand the **DataBindings** node and select *Advanced*.

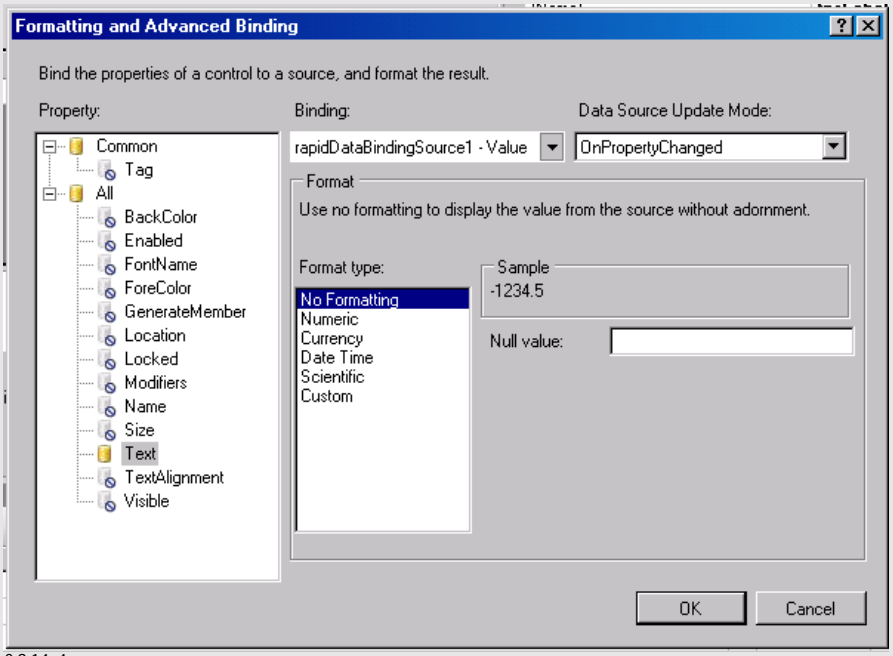


6.3.14_3

Continues on next page

5.3.14. DataBinding of RAPID data and IO signals

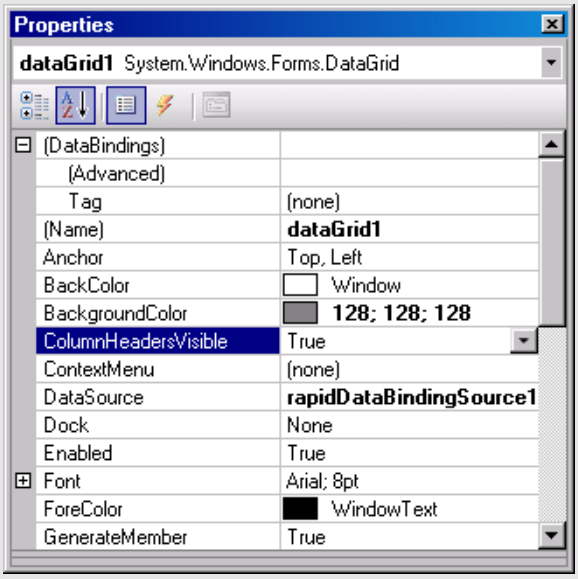
Continued

Step	Action
3.	In the **Advanced Binding** dialog box that appears, choose the already created `RapidDataBindingSource` in the **Binding** combo box, at the same time specifying which one of the **RapidDataObject** properties you want to bind to, in this case **Value**. (The other properties available are variable name, task name and module name, as can be seen in the figure in step 1.) You also select your preferred **Data Source Update Mode**, usually *OnProperty-Changed*. The yellow marking in the list to the left shows that the binding has been done to the **Text** property of the label. When a control has been bound to a data source you will see the same yellow marking in its **Properties** window, at the bound property. See figure of step 2.  6.3.14_4 **NOTE!** If a label has been bound to a data source with several member objects, the first one in the list is the one by default displayed by the label. If the selected index of the list changes (if the user selects a row in the grid for example) the label is updated automatically to show the value of the selected index. See the figure in *GUI example on page 122*.

Continues on next page

Continued

Step	Action
4.	Now launch the **Properties** window of the `DataGrid` control. Set **ColumnHeaderVisible** to true and select your data source at the **DataSource** property.



6.3.14_5

NOTE!

The `DataGrid` control displays only one row in design-time (See the figure of step 1). In run-time, however, the entire collection of **RapidDataObject** members is displayed (see the figure in *GUI example on page 122*).

SuspendBinding/ResumeBinding

`SuspendBinding()` and `ResumeBinding()` have extended functionality compared to the methods of the inherited `BindingSource` class. Removing/adding subscriptions have been added, as subscribing to signals and RAPID data can be resource consuming. These methods can be used in the `Deactivate/Activate` methods, which are executed each time the user switches back and forth between different applications using the FlexPendant task bar. See *Application Framework usage - ITpsViewActivation on page 190* for further information.

If you suspend the binding you no longer get any updates if the data source is changed. However, the binding still exists, so if **DataSourceUpdateMode** is set to *OnValidation* the value will be updated when you repaint the control, as the value from the controller will be read before it is repainted.

Method	Description
SuspendBinding	Suspends the binding and value change event.
ResumeBinding	Resumes the binding and value change event.

5.4 Launching other views

5.4.1. Using launch service

Overview

The FlexPendant SDK provides a launch service, which enables a RAB application to start a standard ABB application, such as the program editor or the jogging view.

It is also possible to start another custom application by specifying proxy assembly name and the application view class.

For information about how to add and launch another view of your own application, see *Adding a view to a custom application on page 99*.

NOTE!

To launch a view in order to edit a specified rapid data instance, another mechanism is available. See *Using standard dialogs to modify data on page 131* for details.

ITpsViewSetup Install

It is only classes that inherit the `ITpsViewSetup` interface that can launch other applications. The`Install` method, which is called when your custom application is launched, has a sender argument. Your application needs to save this object to be able to use the `ITpsViewLaunchServices` interface. It is used by the launch service to call `LaunchView` and `CloseView`.

```
VB:
//declaration
Private iTpsSite As ITpsViewLaunchServices
.....
//Install method of the TpsView class
Function Install(ByVal sender As System.Object, ByVal data As
     System.Object) As Boolean Implements ITpsViewSetup.Install
  If TypeOf sender Is ITpsViewLaunchServices Then
   // Save the sender object for later use
     Me.iTpsSite = DirectCast(sender, ITpsViewLaunchServices)
     Return True
  End If
  Return False
End Function
```


Continues on next page

Continued

```
C#:
//declaration
private ITpsViewLaunchServices iTpsSite;
......
//Install method of the TpsView class
bool ITpsViewSetup.Install(object sender,object data)
{
  if (sender is ITpsViewLaunchServices) {
     // Save the sender object for later use
     this.iTpsSite = sender as ITpsViewLaunchServices;
     return true;
  }
  return false;
}
```

Launching standard views

Using the `FpStandardView` enumerator, the following standard views can be started using the launch services:

- Program editor
- Program data
- Jogging
- Logoff
- Backup/Restore

TIP!

Launching the Program data view is the best way to let end-users create new RAPID data.

NOTE!

To launch the program editor an initialization object of `RapidEditorInitData` type can be used as argument. It specifies the task, module and row that the Program editor should display when it opens. For the other standard views no `InitData` can be used.

`LaunchView / CloseView example`

In the example below, the program editor is launched at a specified routine. First `initData` is created. Then the reference to the sender object, retrieved by the `Install` method in the previous example, is used to call `LaunchView`.

The cookie out argument is later used to specify the view to be closed by the `CloseView` method.

VB:

```
Dim initData As RapidEditorInitData = New RapidEditorInitData
      (ATask, AModule, ARoutine.TextRange.Begin.Row)
If Not (Me.iTpsSite.LaunchView(FpStandardView.RapidEditor,
      initData, True, Me.cookieRapidEd) = True) Then
  GTPUMessageBox.Show(Me, Nothing, "Could not start RapidEditor
       application")
  Return
```

Continues on next page

Continued

```
    End If
    ......
    Me.iTpsSite.CloseView(Me.cookieRapidEd)
```

C#:

```
    RapidEditorInitData initData = new RapidEditorInitData(task,
         module,routine.TextRange.Begin.Row;
    if(this.iTpsSite.LaunchView(FpStandardView.RapidEditor,initData,
         true,out this.cookieRapidEd)!= true) {
      GTPUMessageBox.Show(this, null,"Could not start RapidEditor
           application");
      return;
    }
    .....
    this.iTpsSite.CloseView(this.cookieRapidEd);
```

What happens if the specified view is already open when the call is made? The third argument specifies application behavior in this case. If `true`, another instance of the view is launched. If `false`, the already opened view gets focus. Notice, however, that creating a new instance is not possible for all standard views. The Jogging view, for example, is not allowed multiple instances.

Launching custom applications

The launch service can also be used to launch another RAB application. The name of the proxy assembly along with the fully qualified class name should be used as arguments.

VB:

```
    If Not (Me.iTpsSite.LaunchView("TpsViewCustomApp.gtpu.dll",
         "ABB.Robotics.SDK.Views.TpsViewCustomApp", Nothing, True,
         Me.cookieUser) = True) Then
      GTPUMessageBox.Show(Me, Nothing, "Could not start User
           application")
      Return
    End If
```

C#:

```
    if (this.iTpsSite.LaunchView("TpsViewCustomApp.gtpu.dll",
         "ABB.Robotics.SDK.Views.TpsViewCustomApp", null, true, out
         this.cookieUser)!= true) {
      GTPUMessageBox.Show(this, null,"Could not start User
           application");
      return;
    }
```

NOTE!

It is the namespace of the *proxy* assembly, ABB.Robotics.SDK.Views,which should be used to specify the `TpsView` class.

NOTE!

By using the argument cookieUser your application can close the launched custom application in the same way as in the *Launching standard views on page 129* example above.

5.4.2. Using standard dialogs to modify data

Overview

The `FpRapidData`, `FpToolCalibration` and `FpWorkObjectCalibration` dialogs take a `RapidData` reference as argument. When the dialog is opened it allows the user to directly operate on the referenced RapidData object, for example modifying a RAPID data variable or calibrate a work object. These dialogs are the ones used by the standard ABB applications. They are ready-made and cannot be modified, except for the title.

Creating the dialog

As with all secondary dialogs, the reference to the dialog should be declared as a class variable. This is to make sure that the reference is available for `Dispose`. The `RapidData` reference can be declared locally if it is disposed immediately after use, or else as a class variable. When the dialog has been created the title can be set using the `Text` property. A `Closed` event handler should be added to dispose the dialog. All three dialogs are created in the same way:

VB:

```
Private ARapidData As RapidData
Friend WithEvents FpRD As FpRapidData
.....
Me.ARapidData = Me.AController.Rapid.GetRapidData(ATaskName,
     AModuleName, AVariableName)
Me.FpRD = New FpRapidData(Me.ARapidData)
Me.FpRD.Text = Me.ARapidData.Name
AddHandler Me.FpRD.Closed, AddressOf Me.FpRD_Closed
Me.FpRD.ShowMe(Me)
```

C#:

```
private FpRapidData fpRD;
private RapidData aRapidData;
......
this.aRapidData = this.aController.Rapid.GetRapidData(taskName,
     moduleName, variableName);
this.fpRD = new FpRapidData(this.aRapidData);
this.fpRD.Text = this.aRapidData.Name;
this.fpRD.Closed += new EventHandler( fpRD_Closed);
this.fpRD.ShowMe(this);
```


Continues on next page

Continued

NOTE!

Verify that the `RapidData` is of correct `RapidDataType` for the dialog before using it, i.e. `tooldata` for the FpToolCalibration dialog and `wobjdata` for the FpWorkObjectCalibration dialog. See *Type checking on page 132* below.

NOTE!

The FpRapidData dialog is created with a RapidData as in argument. If the RapidData is an array, an index argument is used to specify which element should be displayed (the first element is 1).

Type checking

When calling the `FpToolCalibration` or `FpWorkObjectCalibration` constructor the `RapidData` value should be type checked before use:

VB:

```
If TypeOf Me.ARapidData.Value Is ToolData Then
  Me.FpTC = New FpToolCalibration(Me.ARapidData)
  .....
End If
```

C#:

```
if (this.aRapidData.Value is ToolData)
{
  this.fpTC = new FpToolCalibration(this.aRapidData);
  ....
}
```

5.5 Using the Controller API

5.5.1. ABB.Robotics.Controllers

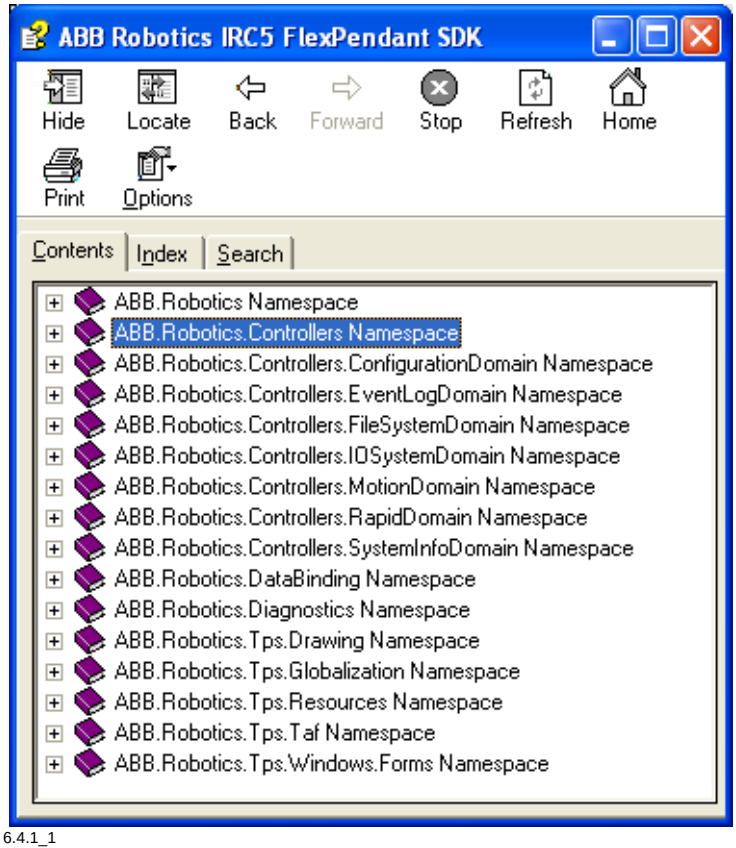
Controller API

To access the functions of the robot controller you utilize the class libraries of the FP SDK called *Controller Application Programming Interface* or *CAPI*. The assembly you need to reference to use controller functionality is *ABB.Robotics.Controllers.dll*.

CAPI domains

The top CAPI object is the `ABB.Robotics.Controllers.Controller`, which has to be created before any access to the robot controller.

The class libraries are organized in different domains (namespaces), as shown by the contents tab of the *FP SDK Reference* below. The name of a domain tells you something about the kind of services you can expect from it.



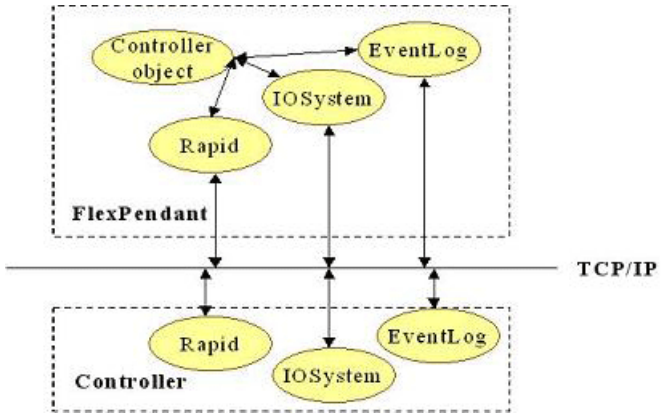
6.4.1_1

Continues on next page

Continued

CAPI and controller domains

This is a simplified illustration of how some `ABB.Robotics.Controllers` domains communicate with their respective domains in the robot controller:



6.4.1_2

NOTE!

In the normal case, a single `Controller` object is created and used throughout the application, its lifetime corresponding to that of the application.

What controller functionality is provided?

Plenty of robot controller functionality is offered by the classes and methods of the FlexPendant SDK.

This table presents short descriptions of the kind of services that the different CAPI domains provide:

CAPI domain	Services
Controllers	Information about the controller, such as IP address, current user, Mac address, operating mode, controller state etc. Notification when operating mode, state or mastership has changed. Backup and restore. Check if the user has the required UAS grant etc.
ConfigurationDomain	Read or write the value of a configuration parameter to the configuration database of the controller.
EventLogDomain	Notification when a new event log message has been written to the controller. Title, message, category, sequence number and time stamp of the message.
FileSystemDomain	Create, rename or remove files and directories in the controller file system. Retrieve a file from the controller and store it on the FlexPendant and vice versa.
IOSystemDomain	Read and modify I/O signals. Notify when a signal value has changed.
MotionDomain	Get/set the coordinate system and motion mode for jogging of the active mechanical unit. Information whether the mechanical unit is calibrated or not. Provide name, task, number of axes, active tool and work object etc. of the mechanical unit. Notify when data of the mechanical unit has changed. Send a start jogging or a stop jogging request to the robot controller.

Continues on next page

Continued

CAPI domain	Services
RapidDomain	Notification when execution status has changed. Start and stop RAPID execution. Load Rapid programs. Create, read and write RAPID data. Notification when RAPID data or RAPID program has changed. Notification when program pointer has changed. Search RAPID symbols etc.
SystemInfoDomain	Information about the active system of the robot controller, e.g. RobotWare version, system name, release and system paths, existing system options and installed additional options.

Releasing memory

Using CAPI you will create objects that reference unmanaged resources. (See *Definitions on page 18*.) It is necessary to explicitly deallocate the memory of such objects by calling their `Dispose` method when they are no longer needed (at application shut down at the latest). Otherwise your application will leak memory, which is a scarce resource on the FlexPendant platform.

NOTE!

You may prevent memory leaks and other pitfalls, by studying the chapter *Robust FlexPendant applications on page 181*.

FP SDK Reference

Although this manual covers a great deal of the FP SDK functionality, it is by no means complete. The *FP SDK Reference*, which you open from Windows **Start** button, is the complete reference to the functionality offered by the class libraries.

It also gives valuable code samples and remarks about methods requiring different UAS grants etc., which is not included in this manual.

5.5.2. Accessing the controller

Overview

Making use of controller functionality requires special attention. Disposal of CAPI objects when closing a view or the entire application is vital to save memory. Another pitfall that must be avoided is updating the user interface on a worker thread.

Controller instance

Before accessing the controller functionality you need to instantiate the `Controller` object. The best place to do this is normally in the `Install` method, which is called by TAF after the constructor of your view class has executed.

The `Controller` declaration should be done on class scope level:

VB:

```
Private AController As Controller
```

C#:

```
private Controller aController;
```

The recommendation is to instantiate the `Controller` object in the `Install` method:

VB:

```
AController = New Controller
```

C#:

```
aController = new Controller();
```


Continues on next page

Using the `Controller` object you can access IO signals, RAPID data variables, event log messages etc. Most of these objects will be created explicitly and should be disposed of explicitly.

NOTE!

Avoid placing any code that may result in an exception before the method `InitializeComponent`. Then at least you have a user interface, where you can display a `MessageBox` with information about what went wrong.

NOTE!

It is recommended that if several classes need to access the controller, they all reference the same `Controller` object.

Subscribing to controller events

The `Controller` object provides several public events, which enable you to listen to operating mode changes, controller state changes, mastership changes etc.

VB:

```
AddHandler AController.OperatingModeChanged, AddressOf UpdateOP
AddHandler AController.StateChanged, AddressOf UpdateState
AddHandler AController.MastershipChanged, AddressOf UpdateMast
AddHandler AController.BackupFinished, AddressOf UpdateBack
```

C#:

```
AController.OperatingModeChanged += new
     OperatingModeChangedEventHandler(UpdateOP);
AController.MastershipChanged += new
     MastershipChangedEventHandler(UpdateMast);
Controller.BackupFinished += new
     BackupFinishedEventHandler(UpdateBack);
Controller.StateChanged += new
     StateChangedEventHandler(UpdateState);
```


Continues on next page

Continued

NOTE!

Controller events use their own threads. Study *Controller events and threads on page 67* to find out how to avoid threading conflicts.

CAUTION!

Do not rely on receiving an initial event when setting up/activating a controller event. There is no guarantee an event will be triggered, so you had better read the initial state from the controller.

Create a backup

Using the `Controller` object you can call the `Backup` method. The argument is a string describing the directory path on the controller where the backup should be stored. As the backup process is performed asynchronously you can add an event handler to receive a `BackupCompleted` event when the backup is completed.

VB:

```
Dim BackupDir As String = "(BACKUP)$"+BackupDirName
AddHandler Me.AController.BackupCompleted, AddressOf
     AController_BackupCompleted)
Me.AController.Backup(BackupDir)
```

C#:

```
string backupDir = "(BACKUP)$"+backupDirName;
this.aController.BackupCompleted += new
     BackupEventHandler(controller_BackupCompleted);
this.aController.Backup(backupDir);
```

NOTE!

There is also a `Restore` method available. The FP SDK reference is the complete FlexPendant SDK programming guide and is more detailed than this manual. For the `Backup` and `Restore` methods, for example, there are parameter descriptions, remarks, code examples etc.

Dispose

The disposal of the `Controller` object should be done in the `Uninstall or Dispose` method of the application view class.

Make a check first that disposal has not already been done. Do not forget to remove any subscriptions to controller events before the `Dispose()` call:

VB:

```
If Not AController Is Nothing Then
  RemoveHandler AController.OperatingModeChanged, AddressOf
       OpMChange
  AController.Dispose()
  AController = Nothing
End If
```

Continues on next page

Continued

C#:

```
if (aController != null)
{
  aController.OperatingModeChanged -= new
       OperatingModeChangedEventHandler(OpMChange);
  aController.Dispose();
  aController = null;
}
```

CAUTION!

VB programmers should be aware that it is a bit tricky to use `WithEvents` together with the Dispose pattern on the .NET platform. When you use `WithEvents` the .NET framework automatically removes any subscriptions when the object is set to `Nothing`. If you look at the code sample above, which does NOT use `WithEvents`, you will understand why such behavior causes problems. When the controller reference is set to `Nothing` and the .NET framework tries to remove its subscription, the internal controller object has already been removed by the `Dispose` call in the preceding statement, and a `NullReferenceException` is thrown. This is not specific to the FlexPendant SDK, but a Microsoft issue. To avoid it you are advised to use `AddHandler` and `RemoveHandler` like in the example.

5.5.3. Rapid domain

5.5.3.1. Working with RAPID data

Overview

The `RapidDomain` namespace enables access to RAPID data in the robot system. There are numerous FP SDK classes representing the different RAPID data types. There is also a `UserDefined` class used to reference RECORD structures in RAPID.

The `ValueChanged` event enables notification from the controller when persistent RAPID data has changed.

TIP!

A convenient and user-friendly way to enable the end-user of your application to read and write to specific RAPID data is using the standard FlexPendant Program Data view. See *Using standard dialogs to modify data on page 131* for details about how this can be done.

TIP!

Using databinding for RAPID data is a quick way of implementing access to RAPID data. See how this works in *DataBinding of RAPID data and IO signals on page 122*.

Providing the path to the RAPID data

To read or write to RAPID data you must first create a `RapidData` object. The path to the declaration of the data in the controller is passed as argument. If you don't know the path you need to search for the RAPID data by using the `SearchRapidSymbol` functionality. See *SearchRapidSymbol method on page 157*.

Direct access

Direct access requires less memory and is faster, and is therefore recommended if you do not need to use the task and module objects afterwards.

The example below shows how to create a `RapidData` object that refers to the RAPID data instance "reg1" in the USER module.

VB:

```
Dim Rd As RapidData = Me.AController.Rapid.GetRapidData(
    "T_ROB1", "USER", "reg1")
```

C#:

```
RapidData rd = aController.Rapid.GetRapidData("T_ROB1", "USER",
    "reg1");
```

Hierarchical access

If you need the task and module objects hierarchical access can be more efficient. `GetRapidData` exists in the Rapid, Task and Module class.

VB:

```
Rd = AController.Rapid.GetTask("T_ROB1").GetModule("USER").
    GetRapidData("reg1")
```

C#:

```
rd = aController.Rapid.GetTask("T_ROB1").GetModule("USER").
    GetRapidData("reg1");
```

Continues on next page

Continued

Accessing data declared in a shared module

If your application is to be used with a multi-move system (a system with one controller and several motion tasks/robots), it may happen that the RAPID instance you need to access is declared in a -*Shared* RAPID module. Such a module can be used by all tasks, T_ROB1, T_ROB2 etc.

This example shows how to create a `RapidData` object that refers to the instance “reg100”, which is declared in a shared module.

VB:

```
Dim Rd As RapidData = Me.AController.Rapid.GetRapidData("reg100")
```

C#:

```
RapidData rd = aController.Rapid.GetRapidData("reg100");
```

Another possibility is using the `Task` object to access the RAPID instance, like this:

```
Task tRob1 = aController.Rapid.GetTask("T_ROB1");
RapidData rData = tRob1.GetRapidData("reg100");
```


Continues on next page

Continued

NOTE!

If `GetRapidData` is called from `Rapid` the RAPID data will be found even if the *-Shared* module is configured to be hidden.

NOTE!

If the RAPID data does not exist, the return value is `Nothing/null` and an `ArgumentNullException` is thrown. A null check should be performed before trying to use the object.

Creating an object representing the RAPID data value

The `RapidData` object stores the path to the RAPID data. But this is not enough if you want to access its value (at least not if you want to *modify* it). To do that you need to create another object, which represents the *value* of the RAPID data.

In the `RapidDomain` namespace there are types representing the different RAPID data types. To create the object needed to represent the RAPID data value you use the `RapidData` property `Value` and cast it to the corresponding type, e.g. `Num`, `Bool` or `Tooldata`.

This is how this is done if you want to access the value of a RAPID data of the RAPID data type `bool`:

VB:

```
'declare a variable of data type RapidDomain.Bool
Dim rapidBool As RapidDomain.Bool
Dim rd As RapidData = Me.AController.Rapid.GetRapidData( "T_ROB1",
     "MainModule", "flag" )
'test that data type is correct before cast
If TypeOf rd.Value Is RapidDomain.Bool Then
  rapidBool = DirectCast(rd.Value, RapidDomain.Bool)
  'check if the value of the RAPID data is true
  If (rapidBool.Value) Then
     ' Do something...
  End If
EndIf
```

C#:

```
//declare a variable of data type RapidDomain.Bool
RapidDomain.Bool rapidBool;
RapidDomain.RapidData rd =
     aController.Rapid.GetRapidData("T_ROB1", "MainModule",
     "flag");
//test that data type is correct before cast
if (rd.Value is ABB.Robotics.Controllers.RapidDomain.Bool)
{
  rapidBool =
       (ABB.Robotics.Controllers.RapidDomain.Bool)rd.Value;
  //assign the value of the RAPID data to a local variable
  bool boolValue = rapidBool.Value;
}
```


Continues on next page

Continued

If you just want to *read* this variable you can use this technique instead of creating a RapidDomain.`Bool` object:

VB:

```
Dim b As Boolean = Convert.ToBoolean(rd.Value.ToString)
```

C#:

```
bool b = Convert.ToBoolean(rd.Value.ToString());
```

The `ToolData` type (representing the RAPID data type `tooldata`) can be created like this:

VB:

```
Dim ATool As ToolData
If Rd.Value Is ToolData Then
  ATool = DirectCast(Rd.Value, ToolData)
End If
```

C#:

```
ToolData aTool;
if (rd.Value is ToolData)
{
  aTool = (ToolData) rd.Value;
}
```

IRapidData.ToString method

All RapidDomain structures representing RAPID data types implement the `IRapidData` interface. It has a `ToString` method, which returns the value of the RAPID data in the form of a string. This is a simple example:

```
string boolValue = rapidBool.ToString();
```

The string is formatted according to the same principle as described in *IRapidData.FillFromString method on page 144* below.

Here is an example of a more complex data type. The `ToolData` `Tframe` property is of the `Pose` type. Its `Trans` value is displayed in a label in the format [x, y, z].

VB:

```
Me.Label1.Text = ATool.Tframe.Trans.ToString()
```

C#:

```
this.label1.Text = aTool.Tframe.Trans.ToString();
```


Continues on next page

Continued

IRapidData.FillFromString method

The `IRapidData` interface also has a `FillFromString` method, which fills the object with a valid RAPID string representation. The method can always be used when you need to modify RAPID data. Using the method with the `RapidDomain.Bool` variable used earlier in the chapter will look like this:

```
rapidBool.FillFromString("True")
```

Using it for a `RapidDomain.Num` variable is similar:

```
rapidNum.FillFromString("10")
```

String format

The format is constructed recursively. An example is the easiest way of illustrating this.

`Example:`

The `RapidDomain.Pose` structure corresponds to the RAPID data type `pose`, which describes how a coordinate system is displaced and rotated around another coordinate system.

```
public struct Pose : IRapidData
{  public Pos trans;  public Orient rot;
}
```

This is an example in RAPID:

```
VAR pose frame1;
...
frame1.trans := [50, 0, 40];
frame1.rot := [1, 0, 0, 0];
```

The frame1 coordinate transformation is assigned a value that corresponds to a displacement in position where X=50 mm, Y=0 mm and Z=40 mm. There is no rotation.

As you see, the `RapidDomain.Pose` structure consists of two other structures, *trans* and *rot*. The *trans* structure consists of three floats and the and *rot* structure consists of four doubles. The `FillFromString` format for a `Pose` object is “[[1.0, 0.0, 0.0, 0.0][10.0, 20.0, 30.0]]”. This piece of code shows how to write a new value to a RAPID `pose` variable:

VB:

```
If TypeOf rd.Value Is Pose Then
  Dim rapidPose As Pose = DirectCast(rd.Value, Pose)
  rapidPose.FillFromString("[[1.0, 0.0, 0.0, 0.0][10, 20, 30]]")
  rd.Value = rapidPose
End If
```

C#:

```
if (rd.Value is Pose)
{
  Pose rapidPose = (Pose) rd.Value;
  rapidPose.FillFromString("[[1.0, 0.5, 0.0, 0.0][10, 15, 10]]");
  rd.Value = rapidPose;
}
```


Continues on next page

NOTE!

Using the same principle arbitrarily long RAPID data types can be represented.

NOTE!

The string format must be carefully observed. If the string argument has the wrong format, a `RapidDataFormatException` is thrown.

Writing to RAPID data

Writing to RAPID data is only possible using the type cast `RapidData` value, to which the new value is assigned. To transfer the new value to the RAPID data in the controller you must finally assign the .NET object to the `Value` property of the RapidData object. This example uses the rapidBool object created in *Creating an object representing the RAPID data value on page 142*.

VB:

```
'Assign new value to .Net variable
rapidBool.Value = False
'Write the new value to the data in the controller
rd.Value = rapidBool
```

C#:

```
//Assign new value to .Net variable
rapidBool.Value = false;
//Write to new value to the data in the controller
rd.Value = rapidBool;
```

This was an easy example, as the value to change was a simple `bool`. Often, however, RAPID uses complex structures. By using the `FillFromString` method you can assign a new value to any `RapidData` and write it to the controller.

The string must be formatted according to the principle described in the previous section. The following example shows how to write a new value to the `pos` structure (x, y, z) of a RAPID tooldata:

```
VB:
Dim APos As Pos = New Pos
APos.FillFromString("[2,3,3]")
Me.ATool.Tframe.Trans = APos
Me.Rd.Value = Me.ATool

C#:
Pos aPos = new Pos();
aPos.FillFromString("[2,3,3]");
this.aTool.Tframe.Trans = aPos;
this.rd.Value = this.aTool;
```


Continues on next page

Continued

NOTE!

The new value is not written to the controller until the last statement is executed.

Letting the user know that RAPID data has changed

In order to be notified that RAPID data has changed you need to add a subscription to the `ValueChanged` event of the `RapidData` instance. Note, however, that this only works for **persistent** RAPID data.

Add subscription

This is how you add a subscription to the `ValueChanged` event:

VB:

```
Addhandler Rd.ValueChanged, AddressOf Rd_ValueChanged
```

C#:

```
this.rd.ValueChanged += rd_ValueChanged;
```

Handle event

Implement the event handler. Remember that controller events use their own threads, and avoid Winforms threading problems by the use of `Control.Invoke`, which forces the execution from the background thread to the GUI thread.

VB:

```
Private Sub Rd_ValueChanged(ByVal sender As Object, ByVal e As
      DataValueChangedEventArgs)
  Me.Invoke(New EventHandler (AddressOf UpdateGUI), sender, e)
End Sub
```

C#

```
private void rd_ValueChanged(object sender,
      DataValueChangedEventArgs e)
{
  this.Invoke(new EventHandler (UpdateGUI), sender, e);
}
```

See *Controller events and threads on page 67* to learn more about potential threading conflicts in RAB applications.

Read new value from controlller

Update the user interface with the new value. As the value is not part of the event argument, you must use the `RapidData Value` property to retrieve the new value:

VB:

```
Private Sub UpdateGUI(ByVal sender As Object, ByVal e As
      System.EventArgs)
  Dim Tool1 As ToolData = DirectCast(Me.Rd.Value, ToolData)
  Me.Label1.Text = Tool1.Tframe.Trans.ToString()
End Sub
```


Continues on next page

Continued

C#

```
private void UpdateGUI(object sender, System.EventArgs e)
{
  ToolData tool1= (ToolData)this.rd.Value;
  this.label1.Text = tool1.Tframe.Trans.ToString();
}
```

NOTE!

Subscriptions work only for RAPID data declared as PERS.

RapidData disposal

Always dispose of `RapidData` objects when they are no longer needed. If you want to reuse a `RapidData` object, you should make sure that you dispose of the current object first.

VB:

```
If Not Rd Is Nothing Then
  Rd.Dispose()
  Rd = Nothing
End If
```

C#:

```
if (rd != null)
{
  rd.Dispose();
  rd = null;
}
```

5.5.3.2. Handling RAPID arrays

Overview

In RAPID you can have up to three dimensional arrays. These are accessible by using a `RapidData` object like any other RAPID data.

There are mainly two ways of accessing each individual element of an array: by indexers or by an enumerator.

TIP!

A convenient and user-friendly way of reading and writing array elements is using the standard Program Data view of the FlexPendant. You provide the element you want to have displayed as argument, and the user can view or manipulate the item the way it is usually done on the FlexPendant. See *Using standard dialogs to modify data on page 131*.

ArrayData object

If the `RapidData` references a RAPID array is `Value` property returns an object of `ArrayData` type. Before making a cast, check the type using the `is` operator or by using the `IsArray` property on the `RapidData` object.

VB:

```
Dim RD As RapidData = AController.Rapid.GetRapidData("T_ROB1",
     "User", "string_array")
If RD.IsArray Then
  Dim AD As ArrayData = DirectCast( RD.Value,ArrayData)
  .....
End If
```

C#:

```
RapidData rd = aController.Rapid.GetRapidData("T_ROB1", "User",
     "string_array");
if (rd.IsArray)
{
  ArrayData ad = (ArrayData)rd.Value;
  .....
}
```

Array dimensions

The dimension of the array is returned by the Rank property. If you need to check the length of the individual arrays you can use the `GetLength` method on the `ArrayData` object passing the dimension index as argument.

VB:

```
Dim ARank As Integer = AD.Rank
Dim Len As Integer = AD.GetLength(ARank)
```

C#:

```
int aRank = ad.Rank;
int len = ad.GetLength(aRank);
```

Continues on next page

Array item access by using indexers

By the use of indexers you can access each array element, even in three dimensional arrays. A combination of the `GetLength` method and `For` loops makes it possible to access any item:

VB:

```
Dim ASum As Double = 0R
Dim ANum As Num
If AD.Rank = 1 Then
  For I As Integer = 1 To AD.Length
     ANum = DirectCast(ad.[I], Num)
     ASum +=  DirectCast(ANum, Double)
  Next
ElseIf AD.Rank = 2 Then
  For I As Integer = 1 To AD.GetLength(1)
     For J As Integer = 1 To AD.GetLength(2)
        ANum = DirectCast(ad[I, J], Num)
        ASum +=  DirectCast(ANum, Double)
     Next
  Next
Else
  For I As Integer = 1 To AD.GetLength(1)
     For J As Integer = 1 To AD.GetLength(2)
        For K As Integer = 1 To AD.GetLength(3)
           ANum = DirectCast(ad[I, J, K], Num)
           ASum +=  DirectCast(ANum, Double)
        Next
     Next
  Next
End If
```

C#:

```
double sum = 0d;
Num aNum;
if (ad.Rank == 1) {
  for (int i = 1; i <= ad.Length; i++)
  {
     aNum = (Num)ad.[i];
     aSum +=  (double)ANum;
  }
```


Continues on next page

Continued

```
}
elseif (ad.Rank == 2)
{
  for(int i = 1; i< ad.GetLength(1); i++)
  {
     for (int j = 1; j <= ad.Length; j++)
     {
        aNum = (Num)ad.[i,j];
        aSum +=  (double)ANum;
     }
  }
}
else {
  for(int i = 1; i< ad.GetLength(1); i++)
  {
     for(int j = 1; j< ad.GetLength(2); j++)
     {
        for (int k = 1; k <= ad.GetLength(3); k++)
        {
           aNum = (Num)ad.[i, j, k];
           aSum +=  (double)ANum;
        }
     }
  }
}
```

Array item access using enumerator

You can also use the enumerator operation (`foreach`) like it is used by collections in .NET. Notice that it can be used for both one dimension and multi-dimensional arrays to access each individual element. The previous example is a lot simpler this way:

VB:

```
Dim ASum As Double = 0R
Dim ANum As Num
For Each ANum As Num In AD
  ASum +=  DirectCast(ANum, Double)
Next
```

C#:

```
double sum = 0d;
Num aNum;
foreach(Num aNum in ad)
{
  aSum +=  (double)ANum;
}
```

5.5.3.3. ReadItem and WriteItem methods

Overview

An alternative way of accessing RAPID data stored in an array are the `ReadItem` and `WriteItem` methods.

ReadItem method

Using the `ReadItem` method you can directly access a RAPID data item in an array, e.g. an array with *RobTargets* or *Nums*. The index to the item is explicitly specified in the `ReadItem` call. The first item is in position 1, i.e. the array is 1-based as in RAPID.

This example retrieves the second *Num* value in the first array of the RAPID data variable referenced by rd.

VB:

```
Dim ANum As Num
aNum = DirectCast(rd.ReadItem(1, 2), Num)
```

C#:

```
Num aNum = (Num)rd.ReadItem(1, 2);
```

WriteItem method

In the same manner it is possible to use the `WriteItem` method to write to an individual RAPID data item in an array. This example shows how to write the result of an individual robot operation into an array representing a total robot program with several operations:

VB:

```
Dim ANum As Num = New Num(OPERATION_OK)
rd.WriteItem(ANum, 1, 2)
```

C#:

```
Num aNum = new Num(OPERATION_OK);
rd.WriteItem(aNum, 1, 2);
```

NOTE!

If the index is out of bounds an `IndexOutOfRangeException` will be thrown.

5.5.3.4. UserDefined data

Overview

You often work with RECORD structures in RAPID code. To handle these unique data types a `UserDefined` class has been implemented. This class has properties and methods to handle individual components of a RECORD.

In some cases implementing your own .NET structure can improve application design and code maintenance.

Creating UserDefined object

The `UserDefined` constructor takes a `RapidDataType` object as argument. To retrieve a `RapidDataType` object the path to the declaration of the RAPID data type is passed as argument.

This example creates a `UserDefined` object referencing the RAPID RECORD *processdata*:

VB:

```
Dim rdt As RapidDataType
rdt = Me.controller.Rapid.GetRapidDataType("T_ROB1", "MyModule",
      "processdata")
Dim processdata As UserDefined = New UserDefined(rdt)
```

C#

```
RapidDataType rdt;
rdt = this.controller.Rapid.GetRapidDataType("T_ROB1",
      "MyModule", "processdata");
UserDefined processdata = new UserDefined(rdt);
```

NOTE!

If the module where the RECORD is defined is configured as -*Shared* you only provide the name of the data type as argument, like this:

```
rdt = this.controller.Rapid.GetRapidDataType("processdata");
```

Reading UserDefined data

You can use a `UserDefined` object to read any kind of RECORD variable from the controller. The individual components of the RECORD are accessible using the `Components` property and an index. Each `Component` can be read as a string.

VB:

```
Dim processdata As UserDefined = DirectCast(rd.Value, UserDefined)
Dim No1 As String = processdata.Components(0).ToString()
Dim No2 AS String = processdata.Components(1),ToString()
```

C#:

```
UserDefined processdata = (UserDefined) rd.Value;
string no1 = processdata.Components[0].ToString();
string no2 = processdata.Components[1].ToString();
```


Continues on next page

Continued

Each individual string can then be used in a `FillFromString` method to convert the component into a specific data type, e.g. `RobTarget` or `ToolData`. See *IRapidData.FillFromString method on page 144* for details.

Writing to UserDefined data

If you want to modify `UserDefined` data and write it to the RECORD in the controller you must first read the `UserDefined` object and the apply new values using the `FillFromString` method. Then you perform a write operation using the `RapidData.Value` property.

VB:

```
processdata.Components(0).FillFromString("[0,0,0]")
processdata.Components(1).FillFromString("10")
rd.Value = ud
```

C#:

```
processdata.Components[0].FillFromString("[0,0,0]");
processdata.Components[1].FillFromString("10");
rd.Value = ud;
```

See *IRapidData.FillFromString method on page 144* and *Writing to RAPID data on page 223* for further information and code samples.

Implement your own struct representing a RECORD

This example shows how you can create your own .NET data type representing a RECORD in the controller instead of using the `UsefDefined` type.

Creating ProcessData type

VB:

```
Dim rdt As RapidDataType = Me.ARapid.GetRapidDataType("T_ROB1",
     "MyModule", "processdata")
Dim p As ProcessData = New ProcessData(rdt)
p.FillFromString(rd.Value.ToString())
```

C#

```
RapidDataType rdt = this.aRapid.GetRapidDataType("T_ROB1",
     "MyModule", "processdata");
ProcessData p = new ProcessData(rdt);
p.FillFromString(rd.Value.ToString());
```


Continues on next page

5.5.3.4. UserDefined data

Continued

Implementing ProcessData struct

This example shows how the new data type `ProcessData` may be implemented. As you see, this is done by using a .NET struct and letting ProcessData wrap the `UserDefined` object.

The struct implementation should include a `FillFromString` and `ToString` method, i.e. inherit the `IRapidData` interface. Any properties and methods may also be implemented.

VB:

```
Public Structure ProcessData
  Implements IRapidData
  Private data As UserDefined

  Public Sub New(ByVal rdt As RapidDataType)
     data = New UserDefined(rdt)
  End Sub

  Private Property IntData() As UserDefined
  Get
     Return data
  End Get

  Set(ByVal Value As UserDefined)
     data = Value
  End Set
  End Property
  .....
End Structure
```

```
C#:
  public struct ProcessData: IRapidData
  {
     private UserDefined data;

     public ProcessData(RapidDataType rdt)
     {
        data = new UserDefined(rdt);
     }
     private UserDefined IntData
     {
        get { return data; }
        set { data = value; }
```


Continues on next page

Continued

```
        }

        public int StepOne
        {
           get
           {
              int res =
                   Convert.ToInt32(IntData.Components[0].ToString())
                   ;
              return res;
           }
           set
           {
              IntData.Components[0] = new Num(value);
           }
        }
     }
```

Implementing IRapidData methods

This piece of code shows how the two `IRapidData` methods `ToString` and `FillFromString` can be implemented.

VB:

```
Public Sub FillFromString(ByVal newValue As String) Implements
     ABB.Robotics.Controllers.RapidDomain.IRapidData.FillFromStr
     ing
  IntData.FillFromString(newValue)
End Sub

Public Overrides Function ToString() As String Implements
     ABB.Robotics.Controllers.RapidDomain.IRapidData.ToString
  Return IntData.ToString()
End Function
```

C#:

```
public void FillFromString(string newValue)
{
  IntData.FillFromString(newValue);
}

public override string ToString()
{
  return IntData.ToString();
}
```

NOTE! The `ToString` method has to use the `Overrides` keyword in Visual Basic and the `override` keyword in C#.

Continues on next page

5.5.3.4. UserDefined data

Continued

Property implementation

Each item in the RECORD structure should have a corresponding property in the extended .NET data type. The get and set methods have to implement the conversion from/to controller data type to .NET data type.

VB:

```
Public Property Step() As Integer
  Get
     Dim res As Integer =
          Convert.ToInt32(IntData.Components(0).ToString())
     Return res
  End Get
  Set(ByVal Value As Integer)
     Dim tmp As Num = New Num
     tmp.FillFromNum(Value)
     IntData.Components(0) = tmp
  End Set
End Property
```

C#:

```
  public int Step
  {
     get
     {
        int res =
             Convert.ToInt32(IntData.Components[0].ToString());
        return res;
     }
     set
     {
        Num tmp = new Num();
        tmp.FillFromNum(value);
        IntData.Components[0] = tmp;
     }
  }
```

5.5.3.5. RAPID symbol search

Overview

Most RAPID elements (variables, modules, tasks, records etc.) are members of a symbol table, in which their names are stored as part of a program tree structure.

It is possible to search this table and get a collection of `RapidSymbol` objects, each one including the RAPID object name, location and type.

SearchRapidSymbol method

The search must be configured carefully, due to the large amount of RAPID symbols in a system. To define a query you need to consider from where in the program tree the search should be performed, which symbols are of interest and what information you need for the symbols of interest.To enable search from different levels the `SearchRapidSymbol` method is a member of several different SDK classes, e.g. `Task`, `Module` and `Routine` . This example shows a search performed with `Task` as the starting point:

VB:

```
Dim RSCol As RapidSymbol()
RSCol = ATask.SearchRapidSymbol(SProp, "num", string.Empty)
```

C#:

```
RapidSymbol[] rsCol;
rsCol = aTask.SearchRapidSymbol(sProp, "num", string.Empty);
```

The `SearchRapidSymbol` method has three arguments. The first argument, of data type `RapidSymbolSearchProperties`, is detailed in the next section. The second and third arguments are detailed in the following sections.

Search properties

The `RapidSymbolSearchProperties` type is rather complex and requires some knowledge about RAPID concepts.

It is used to specify search method, type of RAPID symbol to search for, whether the search should be recursive, whether the symbols are local and/or global and whether or not the search result should include only symbols currently used by a program. If a property is not valid for a particular symbol, it will just be discarded and will not exclude the symbol from the search result.

The table describes the different properties of `RapidSymbolSearchProperties`.

Property	Description
SearchMethod	Specifies the direction of the search, which can be `Block` (down) or `Scope` (up). Example: If the starting point of the search is a routine, a block-search will return the symbols declared within the routine, whereas a scope-search will return the symbols accessible from the routine.

Continues on next page

Continued

Property	Description
SymbolType	Specifies which RAPID type(s) you want to search for. The `SymbolTypes` enumeration includes `Constant`, `Variable`, `Persistent`, `Function`, `Procedure`, `Trap`, `Module`, `Task`, `Routine`, `RapidData`. etc. (`Routine` includes `Function`, `Procedure` and `Trap`. `RapidData` includes `Constant`, `Variable` and `Persistent`.)
Recursive	For both block and scope search it is possible to choose if the search should stop at the next scope or block level or recursively continue until the root (or leaf) of the symbol table tree is reached.
GlobalRapidSymbol	Specifies whether global symbols should be included.
LocalRapidSymbol	Specifies whether local symbols should be included.
IsInUse	Specifies whether only symbols in use by the loaded RAPID program should be searched.

Default instance

`RapidSymbolSearchProperties` has a static method, which returns a default instance.

VB:

```
Dim SProp As RapidSymbolSearchProperties =
     RapidSymbolSearchProperties.CreateDefault()
```

C#:

```
RapidSymbolSearchProperties sProp =
     RapidSymbolSearchProperties.CreateDefault();
```

The default instance has the following values:.

Property	Description
SearchMethod	SymbolSearchMethod.Block
SymbolType	SymbolTypes.NoSymbol
Recursive	True
GlobalRapidSymbol	True
LocalRapidSymbol	True
IsInUse	True

Using this instance you can specify the search properties of the search you want to perform.

Example:

VB:

```
SProp.SearchMethod = SymbolSearchMethod.Scope
SProp.SymbolType = SymbolTypes.Constant Or SymbolTypes.Persistent
SProp.Recursive = False
```

C#:

```
sProp.SearchMethod = SymbolSearchMethod.Scope;
sProp.SymbolType = SymbolTypes.Constant | SymbolTypes.Persistent
sProp.Recursive = false;
```

Continues on next page

Continued

NOTE!

The default instance has the property `SymbolType` set to `NoSymbol`, which means you need to specify it in order to perform a meaningful search.

NOTE!

The `SymbolType` property allows you to combine several types in the search. See the example above.

Data type argument

The second argument of the `SearchRapidSymbol` method is the RAPID data type written as a string. The data type should be written with small letters, e.g. “num”, “string” or “robtarget”. It can also be specified as `string.Empty`.

NOTE!

To search for a `UserDefined` data type the *complete* path to the module that holds the RECORD definition must be passed, like this:

```
result = tRob1.SearchRapidSymbol(sProp,"RAPID/T_ROB1/MyModule/
MyDataType", string.Empty);
```

However, if `MyModule` is configured as -*Shared* the system sees its data types as installed, and the task or module should **not** be included in the path

```
result = tRob1.SearchRapidSymbol(sProp,"MyDataType", string.Empty);
```

Symbol name argument

The third argument is the name of the RAPID symbol. It can be specified as `string.Empty` if the name of the symbol to retrieve is not known, or if the purpose is to search ALL “num” instances in the system for example.

Instead of the name of the RAPID symbol a regular expression can be used. The search mechanism will then match the pattern of the regular expression with the symbols in the symbol table. The regular expression string is not case sensitive

A regular expression is a powerful mechanism. It may consist of ordinary characters and meta characters. A meta character is an operator used to represent one or several ordinary characters, and the purpose is to extend the search.

Within a regular expression, all alphanumeric characters match themselves, i.e. the pattern “abc” will only match a symbol named “abc”. To match all symbol names containing the character sequence “abc”, it is necessary to add some meta characters. The regular expression for this is “.*abc.*”.

The available meta character set is shown below:

Expression	Meaning
.	Any single character
^	Any symbol starting with
[s]	Any single character in the non-empty set s, where s is a sequence of characters. Ranges may be specified as c-c.
[^s]	Any single character not in the set s.

Continues on next page

5.5.3.5. RAPID symbol search

Continued

Expression	Meaning
r*	Zero or more occurrences of the regular expression r.
r+	One or more occurrences of the regular expression r.
r?	Zero or one occurrence of the regular expression r.
(r)	The regular expression r. Used for separate that regular expression from another.
r | r’	The regular expressions r or r’.
.*	Any character sequence (zero, one or several characters).

Example 1

"^c.*"

Returns all symbols starting with c or C.

Example 2

"^reg[1-3]"

Returns reg1, Reg1, REG1, reg2, Reg2, REG2, reg3, Reg3 and REG3.

Example 3

"^c.*|^reg[1,2]"

Returns all symbols starting with c or C as well as reg1, Reg1, REG1, reg2, Reg2 and REG2.

SearchRapidSymbol example

This example searches for VAR, PERS or CONST num data in a task and its modules. The search is limited to globally declared symbols. By default the search method is `Block`, so it does not have to be set.

VB:

```
Dim SProp As RapidSymbolSearchProperties =
     RapidSymbolSearchProperties.CreateDefault()
SProp.SymbolTypes = SymbolTypes.RapidData
SProp.LocalRapidSymbol = False
Dim RSCol As RapidSymbol()
RSCol = ATask.SearchRapidSymbol(SProp, "num", string.Empty)
```

C#:

```
RapidSymbolSearchProperties sProp =
     RapidSymbolSearchProperties.CreateDefault();
sProp.SymbolType = SymbolTypes.RapidData;
sProp.LocalRapidSymbol = false;
RapidSymbol[] rsCol;
rsCol = aTask.SearchRapidSymbol(sProp, "num", string.Empty);
```


Continues on next page

Search for UserDefined RAPID data - example

In this example a user defined RECORD data type (“mydata”) is declared in a module (“myModule”). Assuming that the end-user can declare and use data of this data type in any program module, the search method must be Block (default). A search for all “mydata” instances may look like this:

VB:

```
Dim SProp As RapidSymbolSearchProperties =
      RapidSymbolSearchProperties.CreateDefault()
SProp.SymbolType = SymbolTypes.RapidData
Dim RSCol As RapidSymbol()
RSCol = ATask.SearchRapidSymbol(SProp, "RAPID/T_ROB1/myModule/
      mydata", string.Empty)
```

C#:

```
RapidSymbolSearchProperties sProp =
      RapidSymbolSearchProperties.CreateDefault();
sProp.SymbolType = SymbolTypes.RapidData;
RapidSymbol[] rsCol;
rsCol = aTask.SearchRapidSymbol(sProp, "RAPID/T_ROB1/myModule/
      mydata", string.Empty);
```

NOTE!

If *myModule* is configured as -*Shared* and all *myData* instances are declared in *myModule* the search method must be set to `Scope` and the `SearchRapidSymbol` call should look like this:

```
rsCol = aTask.SearchRapidSymbol(sProp, "mydata", string.Empty);
```

5.5.3.6. RAPID execution

Start and Stop RAPID programs

You can start and stop RAPID execution using `Rapid.Start` and `Stop` methods. A `StartResult` from the `Start` method returns the result of the call. Start arguments can be used. `RegainMode` defines how the mechanical unit should handle path status at start. `ExecutionMode` specifies if the program should run continuously, step backward or go to the next Move instruction etc.

The `Stop` method can include a `StopMode` argument, specifying when the program should stop (after current cycle, after completed instruction or immediately).

VB:

```
AController.Rapid.Start(RegainMode.Regain,
      ExecutionMode.Continous)
.....
AController.Rapid.Stop(StopMode.Instruction)
```

C#:

```
aController.Rapid.Start(RegainMode.Regain,
      ExecutionMode.Continous);
.....
aController.Rapid.Stop(StopMode.Instruction);
```

NOTE!

It is also possible to start a service routine or an ordinary routine without any parameters as if it were a service routine. See `Task.CallRoutine` and `Task.CancelRoutine` in the FP SDK Reference Help for detailed information along with code samples.

RAPID execution change event

It is possible to subscribe to events that occur when a RAPID program starts to execute and when it stops. This is done on the `Rapid` property of the `Controller` object, like this:

VB:

```
AddHandler AController.Rapid.ExecutionStatusChanged, AddressOf
      UpdateUI
```

C#

```
aController.Rapid.ExecutionStatusChanged += new
      ExecutionStatusChangedEventHandler(UpdateUI);
```

See *Letting the user know that RAPID data has changed on page 224* for information and code example on how write the event handlers needed to update the GUI due to a controller event.

ResetProgramPointer method

The `ResetProgramPointer` method resets the program pointers of all tasks and sets them to the main entry point of the respective task.

VB:

```
AController.Rapid.ResetProgramPointer()
```

Continues on next page

Continued

C#:

```
aController.Rapid.ResetProgramPointer();
```

NOTE!

It is also possible to set the program pointer to a specified routine, row or position. See `Task.SetProgramPointer` in the *FP SDK Reference* for information along with code samples.

5.5.3.7. Modifying modules and programs

Overview

Using the `Task` object it is possible to load and save programs and individual modules. You can also unload programs, get the position of the motion pointer (MP) and the program pointer (PP) as well as modify a robot position.

Load modules and programs

To load a module or program file you need the path to the file in the file system of the controller. When the file is loaded into memory the `RapidLoadMode` enumeration argument, `Add` or `Replace`, specifies whether or not it should replace old modules or programs.

If the file extension is not a valid module (mod or sys) or program (pgf) extension an `ArgumentException` is thrown.

VB:

```
Try
  ATask.LoadProgramFromFile(APrgFileName, RapidLoadMode.Replace)
  ATask.LoadModuleFromFile(AModFileName, RapidLoadMode.Add)
Catch ex As ArgumentException
  Return
End Try
```

C#:

```
try
{
  aTask.LoadProgramFromFile(aPrgFileName, RapidLoadMode.Replace);
  aTask.LoadModuleFromFile(aModFileName, RapidLoadMode.Add);
}
catch (ArgumentException ex)
{
  return;
}
```

Save and unload RAPID program

You can save a program using the `SaveProgramToFile` method and unload it using the `DeleteProgram` method. These methods save and unload all modules in the task.

VB:

```
Dim TaskCol As Task() = AController.Rapid.GetTasks()
Dim AnObject As Object
For Each AnObject in TaskCol
   ATask = DirectCast(AnObject, Task)
   ATask.ProgramName = ATask.Name
   ATask.SaveProgramToFile(SaveDir)
   ATask.DeleteProgram()
Next
```

Continues on next page

Continued

C#:

```
Task[] taskCol = aController.Rapid.GetTasks();
foreach (Task aTask in taskCol)
{
   aTask.PrograamName = aTask.Name;
   aTask.SaveProgramToFile(saveDir);
   aTask.DeleteProgram();
}
```

Save module

You can save a module by using the `Module.SaveToFile` method. As argument you use a path to the controller file system.

VB:

```
AModule.SaveToFile(AFilePath)
```

C#

```
aModule.SaveToFile(aFilePath);
```

ProgramPointer and MotionPointer

The Task.`ProgramPointer` property returns the current location of the program pointer (module, routine and row number), i.e. where the program is currently executing. The same functionality is available for motion pointer by using the `MotionPointer` property.

VB:

```
Dim APP As ProgramPointer = ATask.ProgramPointer
If Not APP = ProgramPointer.Empty Then
  Dim AStartRow As Integer = APP.Start.Row
  .....
```

C#:

```
ProgramPointer pp = aTask.ProgramPointer;
if (pp != ProgramPointer.Empty)
{
  int aStartRow = pp.Start.Row;
  .....
}
```

ModifyPosition method

Using the ModifyPosition method of the `Task` object you can modify the position of a `RobTarget` instance in the currently loaded program. As arguments you supply a module `Name` as well as a `TextRange` object. The first `RobTarget` within the text range specified by the `TextRange` object will be changed using the current TCP of the active mechanical unit.

VB:

```
Me.ATask.ModifyPosition(AModule, ATextRange)
```

C#:

```
this.ATask.ModifyPosition(aModule, aTextRange)
```

TIP!

Learn more about `Task` methods and properties in the *FP SDK Reference*.

5.5.4. IO system domain

Overview

A robot system uses input and output signals to control processes. Signals can be of digital, analog or group signal type. Such IO signals are accessible using the SDK.

Signal changes in the robot system are often significant, and there are many scenarios where end-users of the system need notification of signal changes.

Accessing signals

Accessing signals is done through the `Controller` object and its property `IOSystem`, which represents the IO signal space in the robot controller.

To access a signal you need the system name of the signal. The object that is returned from the `IOSystem.GetSignal` method is of type `Signal`.

VB:

```
Dim Signal1 As Signal = AController.IOSystem.GetSignal("signal
     name")
```

C#:

```
Signal signal1 = aController.IOSystem.GetSignal("signal name");
```

The returned `Signal` object has to be typecast to digital, analog or group signal. This example shows a how a signal of type `DigitalSignal` is created:

VB:

```
Dim DISig As DigitalSignal = DirectCast(Signal1, DigitalSignal)
```

C#:

```
DigitalSignal diSig = (DigitalSignal) signal1;
```

This example shows a how an `AnalogSignal` is created:

VB:

```
Dim AISig As AnalogSignal = DirectCast(Signal1, AnalogSignal)
```

C#:

```
AnalogSignal aiSig = (AnalogSignal) signal1;
```

This example shows a how a `GroupSignal` is created:

VB:

```
Dim GISig As GroupSignal = DirectCast(Signal1, GroupSignal)
```

C#:

```
GroupSignal giSig = (GroupSignal) signal1;
```


Continues on next page

Continued

NOTE!

Remember to call the `Dispose` method of the signal when it should no longer be used.

Getting signals using SignalFilter

Instead of just getting one signal at a time you can use a filter and get a signal collection. Some of the `SignalFilter` flags are mutually exclusive, e.g. `SignalFilter.Analog` and `SignalFilter.Digital`. Others are inclusive, e.g. `SignalFilter.Digital` and `SignalFilter.Input`. You can combine the filter flags using the “|” character in C# and the `Or` operator in VB:

VB:

```
Dim ASigFilter As SignalFilter = SignalFilter.Digital Or
     SignalFilter.Input
Dim Signals As SignalCollection  =
     AController.IOSystem.GetSignals(ASigFilter)
```

C#:

```
SignalFilter aSigFilter = SignalFilter.Digital |
     SignalFilter.Input;
SignalCollection signals =
     aController.IOSystem.GetSignals(aSigFilter);
```

This piece of code iterates the signal collection and adds all signals to a `ListView` control. The list has three columns displaying signal name, type and value:

VB:

```
For Each ASignal As Signal In Signals
  Item = New ListViewItem(ASignal.Name)
  Item.SubItems.Add(ASignal.Type.ToString())
  Item.SubItems.Add(ASignal.Value.ToString())
  Me.ListView1.Items.Add(Item)
Next
```

C#:

```
foreach(Signal signal in signals)
{
  item = new ListViewItem(signal.Name);
  item.SubItems.Add(signal.Type.ToString());
  item.SubItems.Add(signal.Value.ToString());
  this.listView1.Items.Add(item);
}
```

If the signal objects are no longer needed they should be disposed of:

VB:

```
For Each ASignal As Signal In Signals
  ASignal.Dispose()
Next
```

Continues on next page

Continued

C#:

```
foreach(Signal signal in signals)
{
  signal.Dispose();
}
```

Reading IO signal values

These examples show how to read a digital and an analog signal.

Digital signal

This piece of code reads the digital signal DO1 and checks a checkbox if the signal value is 1 (ON):

VB:

```
Dim Sig As Signal = AController.IOSystem.GetSignal("DO1")
Dim DigitalSig As DigitalSignal = DirectCast(Sig, DigitalSignal)
Dim val As Integer = DigitalSig.Get
If val = 1 Then
  Me.CheckBox1.Checked = True
EndIf
```

C#:

```
Signal sig = aController.IOSystem.GetSignal("DO1");
DigitalSignal digitalSig = (DigitalSignal)sig;
int val = digitalSig.Get();
if (val == 1)
{
  this.checkBox1.Checked = true;
}
```

Analog signal

This piece of code reads the value of the analog signal AO1 and displays it in a textbox:

VB:

```
Dim Sig As Signal = AController.IOSystem.GetSignal("AO1")
Dim AnalogSig As AnalogSignal = DirectCast(Sig, AnalogSignal)
Dim AnalogSigVal As Single = AnalogSig.Value
Me.TextBox1.Text = AnalogSigVal.ToString()
```

C#:

```
Signal sig = aController.IOSystem.GetSignal("AO1");
AnalogSignal analogSig = (AnalogSignal)sig;
float analogSigVal = analogSig.Value;
this.textBox1.Text = analogSigVal.ToString();
```


Continues on next page

Writing IO signal values

In this example, new values for the IO signals that were retrieved in the previous example are written to the controller.

NOTE!

In manual mode a signal value can be modified only if the *Access Level* of the signal is *ALL*. If not, the controller has to be in auto mode.

Digital signal

This piece of code changes the value of a digital signal in the controller when the user checks/unchecks a checkbox:

VB:

```
Private Sub CheckBox1_Click(ByVal sender As Object, ByVal e As
     System.EventArgs) Handles CheckBox1.Click
  If Me.CheckBox1.Checked Then
     DigitalSig.Set()
  Else
     DigitalSig.Reset()
  End If
End Sub
```

C#:

```
private void checkBox1_Click(object sender, EventArgs e)
{
  if (this.checkBox1.Checked)
  {
     digitalSig.Set();
  }
  else
  {
     digitalSig.Reset();
  }
}
```

NOTE! You can also set the value using the `Value` property.

Analog signal

This piece of code writes the value entered in a text box to the analog signal AO1. The value is converted from string to a float before it is written to the controller:

VB:

```
Dim AnalogSigVal As Single = Convert.ToSingle(Me.TextBox1.Text)
AnalogSig.Value = AnalogSigVal
```

Continues on next page

Continued

C#:

```
float analogSigVal = Convert.ToSingle(this.textBox1.Text);
analogSig.Value = analogSigVal;
```

Listening to signal changes

Once a `Signal` object is available it is possible to add a subscription to its Changed event, which is triggered at a signal change such as changed value, changed simulated status or changed signal quality.

Visual Basic

```
Friend WithEvents AISig As AnalogSignal
...
AddHandler AISig.Changed, AddressOf AISig_Changed
...
Private Sub AISig_Changed(sender As Object, e As
     SignalChangeEventArgs) Handles AISig.Changed
End Sub
```

C#

```
this.aiSig.Changed +=new SignalChangeHandler(aiSig_Changed);
```

NOTE! The event handler skeleton is auto generated using the Tab key twice after "+=" in the above statement:

```
private void aiSig_Changed(object sender, SignalChangeEventArgs e)
{ }
```

Start and stop subscriptions

It is recommended that you activate and deactivate subscriptions to the `Changed` event if these are not necessary throughout the lifetime of the application:

VB:

```
AddHandler AISig.Changed, AddressOf AISig_Changed
RemoveHandler AISig.Changed, AddressOf AISig_Changed
```

C#:

```
this.aiSig.Changed += new SignalChangeHandler(aiSig_Changed);
this.aiSig.Changed -= new SignalChangeHandler(aiSig_Changed);
```


Continues on next page

Avoiding threading conflicts

It is important to keep in mind that all controller events use their own threads, which are different from the application GUI thread. This can cause problems if you want to display signal changes in the application GUI.

If an update of the user interface is not necessary, you do not need to take any special action, but can execute the event handler on the event thread. If, however, you need to show to the user that the signal has changed you should use the `Invoke` method. It forces execution to the window control thread and thus provides a solution to potential threading conflicts.

VB:

```
Me.Invoke(New ABB.Robotics.Controllers.IOSystemDomain.
      SignalChangedEventHandler(AddressOf UpdateUI), New Object()
      {sender, e})
```

C#:

```
this.Invoke(new ABB.Robotics.Controllers.IOSystemDomain.
      SignalChangedEventHandler(this.UpdateUI), new Object[]
      {sender, e});
```

See *Controller events and threads on page 67* for further information.

Finding out the new value

The `SignalChangedEventArgs` object has a `NewSignalState` property, which has information about signal value, signal quality and whether the signal is simulated or not:

VB:

```
Private Sub UpdateUI(ByVal Sender As Object, ByVal e As
      SignalChangedEventArgs)
  Dim State As SignalState = e.NewSignalState
  Dim val As Single
  Val = State.Value
  Me.TextBox1.Text = Val.ToString()
  ....
End Sub
```

C#:

```
private void UpdateUI(object sender, SignalChangedEventArgs e)
{
  SignalState state = e.NewSignalState;
  ....
  float val = state.Value
  this.textBox1.Text = val.ToString()
}
```

NOTE!

Do not count on receiving an initial event when setting up the subscription. To get initial information about the value of a signal you should read it using the `Value` property.

NOTE!

Make sure the subscription is removed before you dispose of the signal.

5.5.5. Event log domain

Overview

Event log messages may contain information about controller status, RAPID execution, the running processes of the controller etc.

Using the SDK it is possible to either read messages in the queue or to use an event handler that will receive a copy of each new log message. An event log message contains queue type, event type, event time, event title and message.

Access the controller event log

You access the event log domain through the `Controller` property `EventLog`.

VB:

```
Private Log As EventLog = AController.EventLog
```

C#:

```
private EventLog log = aController.EventLog;
```

Access event log categories

All event log messages are organized into categories. To search for an individual message you have to know what category it belongs to. The enumeration type, `CategoryType`, defines all available categories. You can get a category either by using the method `GetCategory` or by using the `Categories` property, which is an array of all available categories.

VB:

```
Dim Cat As EventLogCategory
Cat = Log.GetCategory(CategoryType.Program)
```

or

```
Cat = Log.Categories(4)
```

C#:

```
EventLogCategory cat;
cat = log.GetCategory(CategoryType.Program);
```

or

```
cat = log.GetCategory[4];
```

NOTE!

The `EventLogCategory` should be disposed of when it is no longer used.

Access event log messages

To access a message you use the `Messages` property of the `Category` object. A collection of messages is returned. The collection implements the `ICollection` and `IEnumerable` interfaces , which means you can use the common operations for collections. Access is done either using an index or by iterating using `foreach`.

VB:

```
Dim Msg As EventLogMessage = Cat.Messages(1)
```

Continues on next page

or

```
Dim Msg As EventLogMessage
For Each Msg In Cat.Messages
  Me.TextBox1.Text = Msg.Title
  .....
Next Item
```

C#:

```
EventLogMessage msg = cat.Messages[1];
```

or

```
foreach(EventLogMessage msg in cat.Messages)
{
  this.textBox1.Text = msg.Title;
  .....
}
```

MessageWritten event

It is possible to add an event handler that is notified when a new messages is written to the controller event log. This is done by subscribing to the EventLog event `MessageWritten`.

The event argument is of type `MessageWrittenEventArgs` and has a `Message` property, which holds the latest event log message.

VB:

```
Private Sub Log_MessageWritten(sender As Object, e As
     MessageWrittenEventArgs) Handles Log.MessageWritten
  Dim Msg As EventLogMessage = e.Message
End Sub
```

C#:

```
private void log_MessageWritten(object sender,
     MessageWrittenEventArgs e)
{
  EventLogMessage msg = e.Message;
}
```

NOTE!

If the application user interface needs to be updated as a result of the event, you must delegate this job to the GUI thread using the `Invoke` method. See *Invoke method on page 68* for further information and code samples.

5.5.6. Motion domain

Overview

The `MotionDomain` namespace lets you access the mechanical units of the robot system.

Using a `MotionSystem` object you can send jogging commands to an mechanical unit and get or set the incremental jogging mode. Using a `MechanicalUnit` object you can get a lot of information about the mechanical units of the robot system.

You can also subscribe to changes of the mechanical unit, e.g. changed tool, work object, coordinated system, motion mode or incremental step size.

MotionSystem object

You access the motion system by using a the `Controller` property `MotionSystem`.

VB:

```
Private AMotionSystem As MotionSystem
AMotionSystem = AController.MotionSystem
```

C#

```
private MotionSystem aMotionSystem;
aMotionSystem = aController.MotionSystem;
```

Accessing Mechanical units

The mechanical units can be of different types, e.g. a robot with a TCP, a multiple axes manipulator or a single axis unit. Mechanical units are available through the `MotionSystem` object. If only the active mechanical unit is of interest you may use the method `GetActiveMechanicalUnit`.

VB:

```
Dim AMechCol As MechanicalUnitCollection =
     AController.MotionSystem.GetMechanicalUnits()
Dim AMechUnit As MechanicalUnit =
     AController.MotionSystem.GetActiveMechanicalUnit();
```

C#:

```
MechanicalUnitCollection aMechCol =
     aController.MotionSystem.GetMechanicalUnits();
MechanicalUnit aMechUnit =
     aController.MotionSystem.GetActiveMechanicalUnit();
```

Jogging

It is possible to jog the active mechanical unit using the `SetJoggingCmd` method and the calls `JoggingStart` and `JoggingStop`. Depending on the selected `MotionMode` and `IncrementalMode` different joints and speeds are configured.

VB:

```
AController.MotionSystem.JoggingStop()
AMechUnit.MotionMode = MotionModeType.Linear
AController.MotionSystem.IncrementalMode =
     IncrementalModeType.Small
AController.MotionSystem.SetJoggingCmd(-50, 50, 0)
AController.MotionSystem.JoggingStart()
```

Continues on next page

Continued

C#:

```
aController.MotionSystem.JoggingStop();
aMechUnit.MotionMode = MotionModeType.Linear;
aController.MotionSystem.IncrementalMode =
     IncrementalModeType.Small;
aController.MotionSystem.SetJoggingCmd(-50, 50, 0);
aController.MotionSystem.JoggingStart();
```

Mechanical unit properties and methods

There are numerous properties available for the mechanical unit, e.g. `Name`, `Model`, `NumberOfAxes`, `SerialNumber`, `CoordinateSystem`, `MotionMode`, `IsCalibrated`, `Tool` and `WorkObject` etc. It is also possible to get the current position of a mechanical unit as a `RobTarget` or `JointTarget`.

VB:

```
Dim ARobTarget As RobTarget =
     AController.MotionSystem.GetActiveMechanicalUnit.GetPositio
     n(CoordinateSystemType.World)
Dim AJointTarget As JointTarget =
     AController.MotionSystem.ActiveMechanicalUnit.GetPosition()
```

C#:

```
RobTarget aRobTarget =
     aController.MotionSystem.GetActiveMechanicalUnit.GetPositio
     n(CoordinateSystemType.World);
JointTarget aJointTarget =
     aController.MotionSystem.ActiveMechanicalUnit.GetPosition()
     ;
```

DataChanged event

By subscribing to the `DataChanged` event of the `MechanicalUnit` object, you will be notified when a change of tool, work object, coordinated system, motion mode or incremental step size occurs.

VB:

```
AddHandler AMechUnit.DataChanged, AddressOf AMech_DataChanged
......
Private Sub AMech_DataChanged(sender As Object, e As
     MechanicalUnitDataEventArgs)
  Select e.Reason
     Case MechanicalUnitDataChangeReason.Tool
        ChangeOfTool(DirectCast(sender, MechanicalUnit))
     Case MechanicalUnitDataChangeReason.WorkObject
        ......
  End Select
End Sub
```


Continues on next page

5.5.6. Motion domain

Continued

C#:

```
aMechUnit.DataChanged += new
     MechanicalUnitDataEventHandler(aMech_DataChanged);
.....
private void aMech_DataChanged(object sender,
     MechanicalUnitDataEventArgs e) {
  switch (e.Reason) {
     case MechanicalUnitDataChangeReason.Tool:
        ChangeOfTool((MechanicalUnit)sender)
     case MechanicalUnitDataChangeReason.WorkObject:
        ......
  }
}
```

TIP!

Read more about the classes, methods and properties available in the `MotionDomain` in the FP SDK Reference help.

5.5.7. File system domain

Overview

Using the FlexPendant SDK FileSystemDomain you can create, save, load, rename and delete files on the controller. You can also create and delete directories.

Accessing files and directories

You access the file system domain through the `Controller` object property `FileSystem`.

VB:

```
Private AFileSystem As FileSystem = AController.FileSystem
```

C#:

```
FileSystem aFileSystem = aController.FileSystem;
```

Controller and FlexPendant file system

You can find and set the remote directory on the controller and the local directory on the FlexPendant device by using the `RemoteDirectory` and `LocalDirectory` properties.

VB:

```
Dim RemoteDir As String = AController.FileSystem.RemoteDirectory
Dim LocalDir As String = AController.FileSystem.LocalDirectory
```

C#:

```
string remoteDir = aController.FileSystem.RemoteDirectory;
string localDir = aController.FileSystem.LocalDirectory;
```

Loading controller files

You can load a file from the controller to the FlexPendant using the `GetFile` method. An exception is thrown if the operation fails. The arguments are complete paths including filenames.

VB:

```
AController.FileSystem.FileSystem.GetFile(RemoteFilePath,
     LocalFilePath)
```

C#:

```
aController.FileSystem.GetFile(remoteFilePath, localFilePath);
```

Saving files

You can save a file to the controller file system by using the `PutFile` method. An exception is thrown if the operation fails. The arguments are complete paths including filenames.

VB:

```
AController.FileSystem.PutFile(LocalFilePath, RemoteFilePath)
```

C#:

```
aController.FileSystem.PutFile(localFilePath, remoteFilePath);
```


Continues on next page

Continued

Getting multiple files and directories

The `FileSystem` class has a method called `GetFilesAndDirectories`. It can be used to retrieve an array of `ControllerFileSystemInfo` objects with information about individual files and directories. The `ControllerFileSystemInfo` object can then be cast to either a `ControllerFileInfo` object or a `ControllerDirectoryInfo` object.

This example uses search pattern to limit the search.

VB:

```
Dim AnArray As ControllerFileSystemInfo()
Dim info As ControllerFileSystemInfo
AnArray = AController.FileSystem.GetFilesAndDirectories("search
     pattern")
Dim I As Integer
For I = 0 To array.Length -1
  info = AnArray(I)
  ......
Next
```

C#:

```
ControllerFileSystemInfo[] anArray;
ControllerFileSystemInfo info;
anArray = aController.FileSystem.GetFilesAndDirectories("search
     pattern");
for (int i=0;i<anArray.Length;i++) {
  info = anArray[i];
  ......
}
```

Using search patterns

As seen in the example above, you can use search patterns to locate files and directories using the `GetFilesAndDirectories` method. The matching process uses the Wildcard pattern matching of Visual Studio. This is a brief summary:

Character in pattern	Matches in string
?	Any single character
*	Zero or more characters
#	Any single digit (0–9)
[*charlist*]	Any single character in *charlist*
[!*charlist*]	Any single character not in *charlist*

TIP!

Read more about the classes, methods and properties available in the `FileSystemDomain` in the FP SDK Reference help.

5.5.8. System info domain

Overview

The `SystemInfoDomain` provides information about the active robot system. This is mainly done through the static methods and properties of the `SystemInfo` class:

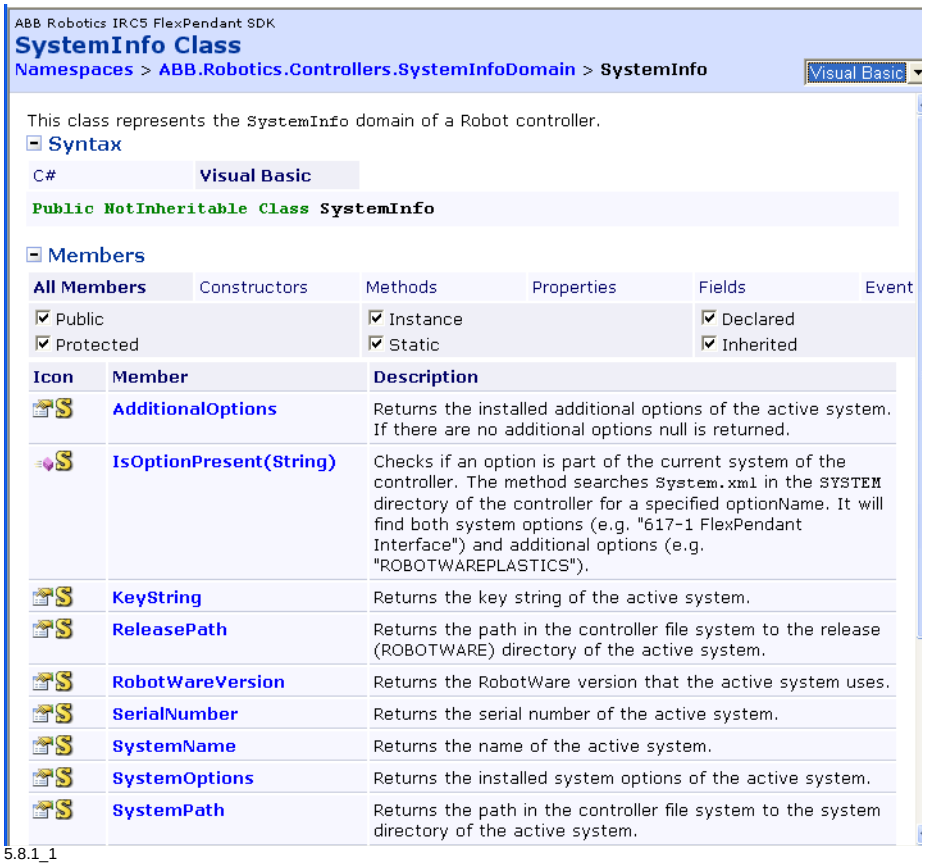


ABB Robotics IRC5 FlexPendant SDK

SystemInfo Class

Namespaces > ABB.Robotics.Controllers.SystemInfoDomain > SystemInfo

Visual Basic

This class represents the SystemInfo domain of a Robot controller.

Syntax

C# | Visual Basic

```
Public NotInheritable Class SystemInfo
```

Members

All Members | Constructors | Methods | Properties | Fields | Event

Public | Instance | Declared

Protected | Static | Inherited

Icon	Member	Description
	AdditionalOptions	Returns the installed additional options of the active system. If there are no additional options null is returned.
	IsOptionPresent(String)	Checks if an option is part of the current system of the controller. The method searches System.xml in the SYSTEM directory of the controller for a specified optionName. It will find both system options (e.g. "617-1 FlexPendant Interface") and additional options (e.g. "ROBOTWAREPLASTICS").
	KeyString	Returns the key string of the active system.
	ReleasePath	Returns the path in the controller file system to the release (ROBOTWARE) directory of the active system.
	RobotWareVersion	Returns the RobotWare version that the active system uses.
	SerialNumber	Returns the serial number of the active system.
	SystemName	Returns the name of the active system.
	SystemOptions	Returns the installed system options of the active system.
	SystemPath	Returns the path in the controller file system to the system directory of the active system.

5.8.1_1

SystemInfo class

The functionality of the `SystemInfoDomain` is accessed by calling the static methods of the `SystemInfo` class.This is how to retrieve the path in the controller file system to the release (ROBOTWARE) directory of the active system.

Example:

```
string rWDir =
     ABB.Robotics.Controllers.SystemInfoDomain.SystemInfo.Releas
     ePath
```

Likewise, the path to the active system directory can be retrieved:

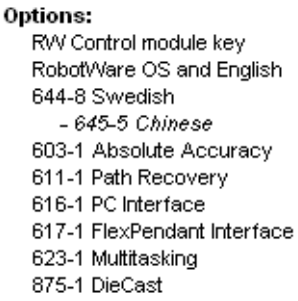
```
string sysDir =
     ABB.Robotics.Controllers.SystemInfoDomain.SystemInfo.System
     Path
```


Continues on next page

Continued

System options

Using the `SystemInfo.SystemOptions` property you can retrieve the system options of the currently active robot system. The result is an array of `SystemOption` objects. If you list the `Name` property of these objects you will get the same result as shown in the *SystemBuilder* of *RobotStudio*, e.g:



5.5.8_2

You can retrieve sub options of a system option by using the `SystemOption.SubOptions` property.

Additional options

Using the `SystemInfo.AdditionalOptions` property you can find out which additional options are installed in the robot system. The result is an array of `AdditionalOption` objects. The `AdditionalOption.Path` property returns the path to the installation directory of the additional option.

The following `AdditionalOption` properties are available:

- KeyString
- Name
- Path
- Type
- VersionInfo

NOTE!

You can find out more about the `SystemInfoDomain` in the *FP SDK Reference*, which also provides you with code examples.

6 Robust FlexPendant applications

6.1. Introduction

Overview

Developing an application for a device with limited resources, such as memory and process power, can be quite demanding. Moreover, to have an application executing around the clock will reveal weaknesses in design and implementation that may cause slow performance or FlexPendant hangings.

At worst, your application will drain the FlexPendant of all memory during production, and cause an out-of-memory crash. It can even slow down the performance of the robot controller due to excessive use of controller resources.

This chapter describes how to design and implement reliable and well performing applications for the FlexPendant. It presents some good practices to utilize, as well as some pitfalls that should be avoided.

Technical overview of the FlexPendant device

The FlexPendant device consists of both hardware and software and is a complete computer in itself, with its own memory, file system, operating system and processor.

It is an integral part of IRC5, connected to the controller by an integrated cable and connector. Using the hot plug button option, however, you can disconnect the FlexPendant in automatic mode and continue running without it.

There are ways to restart the FlexPendant without having to restart the controller (See *Restart the FlexPendant on page 49*). At a FlexPendant restart the assemblies and resources of FlexPendant SDK applications are downloaded to the FlexPendant file system from the robot controller.

FlexPendant applications run on Windows CE, a scalable embedded operating system, and the .NET Compact Framework, which is Microsoft's lightweight version of the .NET Framework, intended for small devices.

This is the size of the FlexPendant touch screen:

FlexPendant screen	Size
Total display	640 * 480 pixels
FP SDK Application display	640 * 390 pixels

The FlexPendant uses these kinds of memory:

Memory type	**Function**
Flash - 16 MB	Stores the FlexPendant standard software, the Windows CE operating system in compressed format and the registry.
RAM - 64 MB	At boot time the compressed image is copied to RAM. All execution of code uses RAM.
E^2PROM	Stores touch screen calibration values, joystick calibration values etc. Only used internally.

How large can a custom application be?

You may wonder about the maximum size of your custom application. There is no exact answer to that question, as there are many variables to take into account. For a rough estimation the table below can be used. As you see, the operating system uses about 8 MB

Continues on next page

Continued

and the ABB base software about 25 MB. This means that half of the available RAM memory is already used once the FlexPendant has started up. The standard applications of the ABB menu and the FlexPendant SDK applications will all share the memory that is left. As a rule of thumb, about 20 MB should be available for custom applications.

FlexPendant memory resources	
RAM	64 MB
Operating system	8 MB
ABB base software	25 MB
Custom applications	~20 MB

This is some advice to help you make sure your application does not exceed the memory limitation:

1. Do not allow more than ONE instance by setting the `TpsViewType` parameter of the `TpsView` attribute to `Static`. See *Application type on page 56* for detailed information.
2. Avoid excessive use of images. Do not use bigger images than necessary. Check the size of the images your application will use.
3. Use fpcmd “-memShow” to check the amount of memory in use when your application is active. Open a couple of Program Editors and start RAPID execution. See *Discover memory leaks on page 186* for further information.
4. Avoid `static` data and methods.
5. Release memory for objects that are not used by calling their `Dispose` method.

6.2. Memory management

Garbage collection and Dispose

An important feature of the .NET runtime environment is the garbage collector, which reclaims not referenced memory from the managed heap. Generally, this means that the programmer should not have to free memory which has been allocated by the use of `new`. A drawback, when memory is limited, is that the execution of the garbage collector is non-deterministic. There is no way of knowing exactly when garbage collection will be performed.

The `IDisposable` interface, however, represents a way to obtain deterministic deallocation of resources. You should therefore call `Dispose()` on all disposable object when they are no longer needed, as this will free up valuable resources as soon as possible.

Moreover, objects used to access robot controller resources, must be released by the custom application by an explicit call to their `Dispose` method. `SignalBindingSource` and `RapidDataBindingSouce` objects, as well as all other objects located in the components pane of the VS Designer must also be explicitly disposed of, or else your application will have a permanent memory leak.

NOTE!

The creator of an object implementing the `IDisposable` interface is responsible for its lifetime and for calling `Dispose`.

TIP!

You may wonder why the .NET garbage collector cannot ensure that all objects no longer referenced are finally destroyed? The FlexPendant development team have tried hard to remove any remaining objects of an SDK application at application shut down, for example implemented finalizers for the `TpsControl`, `RapidDataBindingSouce` and `SignalBindingSource` classes. Due to Microsoft's implementation, however, the .NET

Continues on next page

Continued

runtime nonetheless refuses to destroy these objects unless their `Dispose` method is called. This behavior is under debate. If you are curious to find out more about this, these community articles may be of interest.

Dispose, Finalization, and Resource Management (Joe Duffy):

http://www.bluebytesoftware.com/blog/PermaLink.aspx?guid=88e62cdf-5919-4ac7-bc33-20c06ae539ae

Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework (MSDN-magazine): ms-help://ms.msdnqtr.v80.en/ms.msdn.v80/dnmag00/html/GCI.htm

Finalization - cbrumme's WebLog: http://blogs.msdn.com/cbrumme/archive/2004/02/20/77460.aspx

Application Framework usage - ITpsViewSetup

The application framework TAF, which hosts the controls that make up a FlexPendant application, offers some mechanisms that should be used by client applications. See *Understanding FlexPendant application life cycle on page 52* to learn more about TAF.

Your application view class should implement `ITpsViewSetup` and `ITpsViewActivation`. See *ITpsViewSetup and ITpsViewActivation on page 58* for general information on these interfaces and *Application Framework usage - ITpsViewActivation on page 190* to learn how to use `ITpsViewActivation` to improve performance.

The `ITpsViewSetup` interface has two methods: `Install` and `Uninstall`. `Install` is called when the view is being created, right after the constructor has been executed.

`Uninstall` is called when the client view is closed down. After `Uninstall` has been called, TAF will also call the `Dispose` method of the view class. These methods thus offer the last opportunity for you to clean up and release memory and system resources held by the custom application.

NOTE!

The user should close down the application by using the close button, [x], in the upper right corner of the display, in the same way as the Program Data or other standard applications are closed. Never implement a close button on the first view. When the application is closed the correct way, first `Deactivate`, then `Uninstall` and finally `Dispose` will be called by TAF.

How to program the Dispose method - example

When starting a new FlexPendant project in Visual Studio a skeleton for the `Dispose` method is auto generated. You should use it to add code for cleaning up as shown by the figure below:

Continues on next page

Continued

```
/// Releases all resources used by this instance.
/// </summary>
/// <param name="disposing"><b>true</b>releases both managed and unmanaged resources (called by Dispos
///                         <b>false</b>releases only unmanaged resources (called by finalizer).</para
protected override void Dispose(bool disposing)
{
    if (!this.IsDisposed)
    {
        if (disposing == true)
        {
            /*TODO: Dispose of all CAPI objects.
             * Remember to remove any subscriptions first.*/
            if (_controller != null)
            {
                _controller.OperatingModeChanged -= new ABB.Robotics.Controllers.OperatingModeChangedE
                _controller.Dispose();
                _controller = null;
            }

            /*TODO: Add code to dispose of all GUI components that have not been added
             *to the controls collections like this: this.Controls.Add(this.comboBox1).
             * (These will be disposed of by the base class.)*/
            if (this.alphaPad1 != null)
            {
                this.alphaPad1.Dispose();
                this.alphaPad1 = null;
            }

            /*TODO: Add code to dispose of System.Drawing objects such as Bitmap, Pen, Brush etc*/
        }
    }
    /*Always call Dispose of base class*/
    base.Dispose(disposing);
}
```

7.2_1

NOTE!

Error handling should be added to the `Dispose` method (left out in the figure).

NOTE!

Ensure you have unsubscribed to any events from the robot controller before you call the `Dispose` method of a CAPI object. If it has been done in the `Deactivate` method, which is what is usually recommended, you should not do it again in the `Dispose` method. Also ensure you do not try to access an object after it has been disposed.

Discover memory leaks

When your application interacts with the robot controller, unmanaged objects are created under the hood. If you forget to call `Dispose` on such objects there will be memory leaks.

You are recommended to test the memory consumption of your application. Use a controller console window on your PC and the command `fpcmd_enable_console_output` 3 to enable printouts. Then use the `"-memShow"` command with a period argument that produces a printout every 500 second for example, like this:

-> period 500, fpcmd, “-memShow”

Result:

task spawned: id = 0xba7f3b8, name = t2value = 195556280 = 0xba7f3b8-> [fp]: Available memory: 20881408 (tot 40394752), disk: 737148 (tot 1728512) load: 54(261955)[fp]:

Test all functions of your application with several other FlexPendant views visible in the task bar and possibly one or several RAPID tasks executing. Observe how your implementation affects memory. The load component shows current memory load on the FlexPendant

Continues on next page

Continued

expressed in percentage and the figures in the parenthesis are number of ms after start-up. Close your application and make sure the same amount of memory is available as before opening it.

The procedure below shows yet another way of checking that your application cleans up correctly:

Step	Action
1	Log out from the FlexPendant by pressing **Log Off** on the ABB menu.
2	In the controller console window write `fpcmd_enable_console_output`.
3	Write `fpcmd "-a"`.
4	For each cpp class check *Number of instances*. It shall be 0, except for `AdaptController` (1) and `AdaptEventImpl` (1).
5	If the previous step shows that there are unmanaged objects left, you need to search your code for missing `Dispose` calls.

NOTE!

You must use the real FlexPendant device when verifying memory consumption and looking for memory leaks in your code.

6.3. Performance

About performance

A FlexPendant application cannot meet hard real time demands. There should however be no difference in performance between a FlexPendant SDK application and the standard applications of the device. If your application is slow, the advice in this section will be useful.

TIP!

Do not get overwhelmed by the number of restrictions presented in this section. Most of the time you will not notice any difference if you decide to neglect a few of them. Yet - it is good to know what can be done whenever performance does become an issue.

Less code means faster code

A general piece of advice (maybe too obvious a recommendation) is that less code normally means faster code. Remember that methods that are executed frequently, such as `OnPaint`, event handlers etc. must be efficient. Everything that can be computed once and stored for future use should be computed only once.

Fewer controller accesses means faster code

The best thing you can do to improve performance is probably to ensure that you do not access controller functionality more often than necessary. To achieve this you need to understand CAPI, the SDK class libraries used to access robot controller functionality.

It is very easy to use a property of a class and not realize that an access to the controller is actually made. A general recommendation is to try to use subscriptions to controller information (events) where applicable, and let your application store updated values instead of performing numerous reading operations toward the same controller resource.

TIP!

You can easily get an estimate of the number of controller accesses your application performs in different user scenarios by using a robot controller console. The console command `robdisp_watch 2` monitors and prints controller accesses made by the FlexPendant to the console. Press a button of your application for example, and study how the FlexPendant and the controller communicate in response to this action. Enter the command `robdisp_watch`

Continues on next page

Continued

`0` to turn this service off. (The monitor service slows down the response time of your application, due to printing to the console. So don't get worried if your application seems unusually slow!)

NOTE!

Keep in mind that an excessive number of controller accesses will slow down the performance of your application. At worst, it may even affect the performance of the robot controller.

NOTE!

Excessive use of subscriptions may also be a problem. If your application has to handle a continuous load of Rapid data and I/O signal events, and your event handlers need to manipulate the data before presenting it, GUI interaction may become slow. As a rule of thumb, do not estimate more than 20 events/sec. and keep the event handlers thin.

Fewer objects means faster code

Fewer objects means better performance. If you know that an object will be used again, create it once and keep a reference. Also, try not to create objects that may not be used.

Reusing existing objects instead of creating new ones is especially important when executing the same code repeatedly, e.g. in tight loops.

This example shows how this should be done:

```
object _o = new object();
for (int i = 0; i < 100000; i++)
{   ... }
```

TIP!

Do not create several `Controller` objects, but reuse it by sending it as a parameter when creating a new view, or share it between classes as a public property.

Transferring files

Transferring files between the device and the robot controller takes time and also occupies storage memory on the device. Write efficient and fault-tolerant code if it must be done.

Continues on next page

Continued

Application Framework usage - ITpsViewActivation

The `ITpsViewActivation` interface is used by TAF to hide and display your application. It has two methods: `Activate` and `Deactivate`. The former is called by TAF at the creation of the client view, right after the `ITpsViewSetup.Install` method has been executed. The latter is called at shut down of the client view, right before the `ITpsViewSetup.Uninstall` method is executed.

The interface is also used when the user selects another application on the task bar. TAF then calls `ITpsViewActivation.Deactivate` (but not `ITpsViewSetup.Uninstall`). Likewise, when the application regains focus via the task bar icon, TAF calls `ITpsViewActivation.Activate` (but not`ITpsViewSetup.Install`).

It is recommended to enable and disable any subscriptions to controller events in the `ITpsViewActivation` methods, as valuable resources should not be held when the application is not used. Note that current values should be read before enabling the subscription.

For the same reason, any timers can be activated and deactivated in these methods. That way timers will not run when other applications are in focus, thus saving processor power.

Excessive string manipulation is costly

The string class is an immutable type, i.e. once a string is created its value cannot be changed. This means that string methods that seem to modify the string in fact create new strings.

Look at a this string concatenation example:

```
string name = "192.168.126.1";
string str = "/" + name + "/" + "RAPID";
```

Four strings will be appended to one resulting string. No less than four additional strings will be created and allocated when the right hand side of the assignment is executed. A better idea is to use the `StringBuilder` class or `string.Format` (which uses the `StringBuilder` internally):

```
string str = string.Format("/{0}/RAPID", name);
```

As a rule, if you are going to do only one string operation on a particular string, you can use the appropriate string method. It is when you start doing numerous string operations that you need to use `StringBuilder` or `string.Format`.

Continues on next page

Continued

Avoid Control.Refresh

`Control.Refresh()` must only be used when an immediate update of the GUI is absolutely required. `Refresh` makes a direct call to the `OnPaint` method of the Control, and is therefore much more costly than `Control.Invalidate()`.

Several calls to `Invalidate` will not mean several calls to `OnPaint`. When the GUI thread processes the `Invalidate` message all queued up messages of the same control are handled at the same time, thus saving process power.

NOTE!

In the 2.0 version of .NET CF there are new methods to be used for GUI controls inheriting `System.Windows.Forms.Control`:

- `SuspendLayout()`
- `ResumeLayout()`
- `BeginUpdate()`
- `EndUpdate()`

These methods are used to control GUI updates while modifying a GUI element. The control will not be drawn when `SuspendLayout` has been called, for example, but is blocked until `ResumeLayout` is called.

Avoid boxing and unboxing

A common reason for slow code is unintentional boxing and unboxing. To avoid this, you need to be aware of the difference between *reference* and *value* types. Reference types (the keyword *class* is used) are always allocated on the heap, while value types are allocated on the stack; unless embedded into a reference type.

Boxing is the operation where a value type, is converted to a reference type. It is done automatically when a reference to a value type is required. Then a new object will be created, allocated on the heap with a copy of the original data.

Here are some examples of not so obvious boxing/unboxing:

- Using the `foreach` statement on an array that contains value types will cause the values to be boxed and unboxed.
- Accessing values of a Hashtable with a value type key, will cause the key value to be boxed when the table is accessed.
- Using an ArrayList with value types; this should be avoided - use typed arrays instead. Typed arrays are also better because of type safety, as type checking can be performed at compile time.
- The following methods should be overridden in order to avoid unnecessary boxing/ unboxing: Equals( ), GetHashCode( ).

Continues on next page

Continued

Foreach

Using a `for` loop is often faster than using the `foreach` statement, especially if a large number of iterations are made. The reason is that the JIT compiler (see *About terms and acronyms on page 18*) is prohibited to optimize the code execution when `foreach` is used.

However, `foreach` makes the code more readable and is therefore a better option when performance is not crucial.

Reflection is performance demanding

Reflection is a mechanism used to read the meta data of an assembly. The `typeof` operator, for example, uses reflection to determine the type of an object. Another example is `object.ToString()`, which also uses reflection.

As reflection is very performance demanding you are recommended to override or to avoid the `ToString()` method for reference types.

Efficiently parsing Xlm

`XmlTextReader` and `XmlDocument` can both be used to parse xml data. The `XmlTextReader` is the preferred option in most cases; it is more light weight (less memory footprint) and a lot faster to instantiate. The limitation of the `XmlTextReader` is that forward only reading is possible.

The xml structure may also have an impact on performance. If xml data is organized in non-flat way, search operations will be faster, as a large portion of the information can be skipped. This is achieved by using categories and sub categories.

6.4. Reliability

Overview

This section presents information on how to make the application robust. The most important mechanism related to robustness and reliability is error handling. Avoiding thread conflicts and memory leaks are however also important means of improving reliability.

Error handling in .NET applications

As already pointed out in this manual, Microsoft recommends that exceptions are used to discover and report anomalities in .NET applications. If you are not sure about when to use exceptions or how to do the implementation you should read *Exception handling on page 72* to understand the general idea before moving on to the FlexPendant specific information of this section.

Continues on next page

Continued

SDK exception classes

The `ABB.Robotics` namespace provides quite a few exception classes, used by the FlexPendant SDK and SDK applications. In addition, the SDK also uses `System` exceptions and may throw a `System.ArgumentException` object for example.

Whenever an exception is thrown, your application must catch it and take proper action. You catch an exception by using the `try-catch(-finally)` statements. See *Exception handling on page 72* for further information about how to implement these statements.

As you see in the figure below `GeneralException` derives from `BaseException`, which in turn derives from `System.ApplicationException` of the .NET Framework class library.

`ApplicationException` is thrown by user programs, such as the FlexPendant SDK, not by the Common Language Runtime (CLR, see *Definitions on page 18*). It therefore represents a way to differentiate between exceptions defined by applications versus exceptions defined by the system.

ABB Robotics IRC5 FlexPendant SDK

GeneralException Class

Common exceptions to be handled by caller.

For a list of all members of this type, see GeneralException Members.

System.Object
  System.Exception
    System.ApplicationException
      ABB.Robotics.BaseException
        ABB.Robotics.GeneralException
          Derived types

```
public class GeneralException : BaseException
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Requirements

Namespace: ABB.Robotics

Assembly: ABB.Robotics (in ABB.Robotics.dll)

See Also

GeneralException Members | ABB.Robotics Namespace

7.4_1

As you see there are derived types under `GeneralException`. These types are listed and briefly described below:

Continues on next page

Continued

ABB Robotics IRC5 FlexPendant SDK
ABB.Robotics Namespace

Namespace hierarchy

Classes

Class	Description
BaseException	ABB.Robotics exception base class.
FPBase	FlexPendant object base
GeneralException	Common exceptions to be handled by caller.
InternalException	Internal error exception.
MasterRejectException	The user does not have required mastership for the operation.
ModeRejectException	The operation is not allowed in current operation mode. For example, a remote user may not be allowed to perform the operation in manual mode.
ModPosException	The modify postion operation failed. The failing task name and reason are available.
RejectException	Operation rejected by the controller safety access restriction mechanism.
ResourceHeldException	Requested resource is held by someone else.
SysfailRejectException	The operation is not allowed in SysFail.
UasRejectException	The user is denied access to the operation.

7.4_2

Continues on next page

Continued

TIP!

Learn more about the FlexPendant SDK exception classes by using the *FlexPendant SDK Reference Documentation Help*.

Thread affinity

You should be aware that execution of code modifying the user interface, has to be done on the thread that created the GUI control, the so called GUI thread.

An application initially starts with a single thread. Normally all user interface controls are created by this thread. Windows CE user interface objects are characterized by thread affinity, which means that they are closely coupled with the thread that created them.

Interacting with the message queue of an interface control from a thread other then the creating thread may cause data corruption or other errors. This restriction applies to the thread pool as well as to explicitly created threads.

When executing on a secondary thread, a so called worker thread, an update of the user interface must therefore be done very carefully, enforcing a switch to the GUI thread. This is in fact a very common scenario, as controller events use their own threads and should often be communicated to the end user by a GUI update.

NOTE!

Thread affinity is especially relevant as for robot controller events, as these by default execute on a background thread. See *GUI and controller event threads in conflict on page 68* and *Invoke method on page 68* for further information along with code samples. The following section also deals with the same issue.

Invoke

In order to execute a method on the GUI thread `Control.Invoke` can be used. It should however be done carefully, as it makes a synchronous call to the specified event handler, which blocks execution until the GUI thread has finished executing the method. All concurrent calls to `Invoke` will be queued and executed in their queue order by the GUI thread. This could easily make the GUI less responsive.

There is now an asynchronous version of `Invoke`, which should be used instead whenever possible. `TpsControl.BeginInvoke` is a non blocking method, which lets the worker thread continue execution instead of waiting for the GUI thread to have processed the method

Remember that `Invoke` as well as BeginInvoke should only be used on code that modifies the user interface. You should keep the execution on the background thread as long as possible.

NOTE!

If your code tries to access a GUI control from a background thread the .NET common language runtime will throw a `System.NotSupportedException`.

Memory leaks

As FlexPendant applications are supposed to run without interruption, no memory leaks can be permitted. It is your responsibility to properly clean up and call `Dispose`. Take your time studying *How to avoid memory leaks on page 93*, *Memory management on page 184* and

Continues on next page

Continued

Technical overview of the FlexPendant device on page 182.

Utilizing multi-threading

Threading enables your program to perform concurrent processing so you can do more than one operation at a time. For example, you can use threading to monitor input from the user, and perform background tasks simultaneously. The `System.Threading` namespace provides classes and interfaces which enable you to easily perform tasks such as creating and starting new threads, synchronizing multiple threads, suspending threads and aborting threads.

The classes `Thread` and `ThreadPool` can be used to execute methods on a worker thread. Use `ThreadPool` for temporary usage of a background thread when a task is meant to terminate fairly soon. Use `Thread` only for background work that will persist, e.g. a thread that fetches data from the controller continuously.

There are two timers available in .NET CF that can be used to execute methods periodically at specified intervals. There is however, an important difference between these. The `System.Threading.Timer` executes the method on a background thread, while the `System.Windows.Forms.Timer` executes the method on the UI thread.

NOTE!

Use `System.Threading.Timer` if you want to poll data from the controller in order to reduce the load on the GUI thread.

Lock statement

The `lock` statement is used in multi-threaded applications to make sure access to a part of the code is made by one thread exclusively. If a second thread attempts to lock code which has already been locked by another thread, it must wait until the lock is released.

Remember to limit the code segment that you want to control, i.e. only lock what is absolutely necessary to make the code thread safe:

```
Object thisLock = new Object();
lock (thisLock)
{
 // Critical code section
}
```

Deadlocks must be avoided. They can occur if more than one lock is used. If more than one lock object must be held, they must always be locked and released in the same order, wherever they are used.

Continues on next page

Continued

NOTE!

If a deadlock occurs, the FlexPendant system will hang. If this happens you should attach the device to the Visual Studio debugger and study the call stack of the threads in the *Threads* window.

CAUTION!

Using the `lock` statement in combination with a call to `Invoke` is a potential cause to deadlock situations, since `Invoke` is a blocking call. In short, be careful if you use locks!

Multicast delegates

If a multicast delegate is executed, the execution of the delegates is terminated if an exception is thrown by one of the delegates. It means that the remaining delegates in the list will not be executed if an exception is thrown. This situation can cause erratic behavior, which may be tricky to trace and debug.

Therefore, if you have numerous delegates which are to execute as the result of the same event, you may want to implement the `GetInvocationList` method. It retrieves a list with a copy of the delegates. This list can be iterated and each delegate called directly:

```
private void OnChange()
{
    Delegate[] evtHandlers = null;

    lock (_lockobj)
    {
        if (_changeHandler != null)
            evtHandlers = _changeHandler.GetInvocationList();
    }

    if (evtHandlers != null)
    {
        for (int i = 0; i < evtHandlers.Length; i++)
        {
            Delegate d = evtHandlers[i];
            try
            {
                ((EventHandler)d)(this, new EventArgs());
            }
            catch (System.Exception e)
            {
                ABB.Robotics.Diagnostics.Debug.Assert(false,
                    string.Format("Exception in OnChange: {0}", e.ToString()));
                ABB.Robotics.Diagnostics.Trace.WriteLine
                    (string.Format("Exception in OnChange: {0}", e.ToString()));
            }
        }
    }
}
```

7.4_3

7 Using the PC SDK

7.1. Controller API

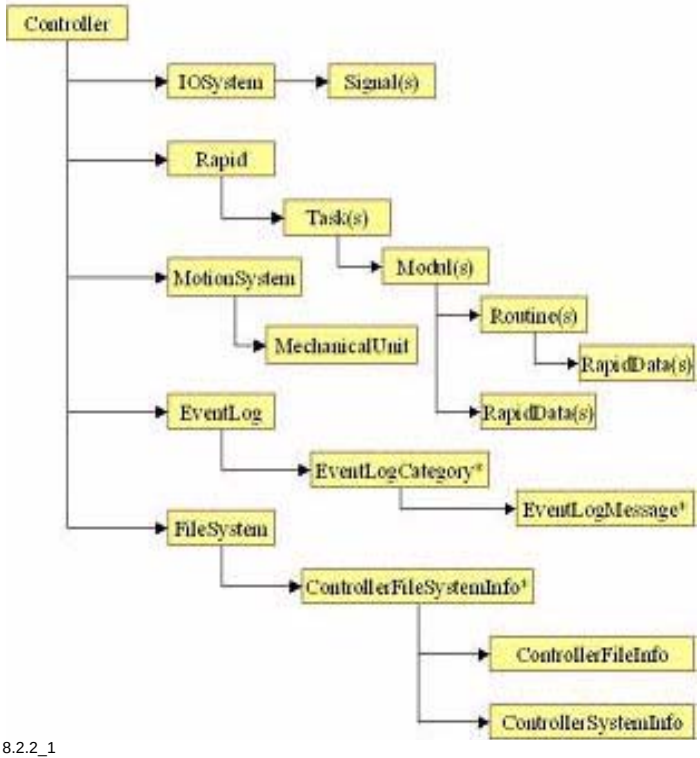
PC SDK domains

The PC SDK class libraries are organized in the following domains:

- Controllers
- ConfigurationDomain
- Discovery
- EventLogDomain
- FileSystemDomain
- Hosting
- IOSystemDomain
- Messaging
- MotionDomain
- RapidDomain
- UserAuthorizationManagement

CAPI illustration

The classes used to access robot controller functionality together make up the *Controller API* (CAPI). Part of the CAPI object model is illustrated below:



8.2.2_1

Continues on next page

PC SDK Reference

This User's Guide covers some of the PC SDK functionality, but is by no means a complete guide to the PC SDK.

The *PC SDK Reference* is the complete reference of the PC SDK class libraries. It should be your companion while programming.

It can be launched from Windows **Start** menu by pointing at *Programs - ABB Industrial IT - Robotics IT - Robot Application Builder 5.xx* and selecting *PC SDK Reference*.

7.2. Create a simple PC SDK application

Overview

To get started programming let us create a simple application that will display all virtual and real controllers on the network. It should then be possible to log on to a controller and start RAPID execution.

CAUTION!

Remote access to controllers must be handled carefully. Make sure you do not unintentionally disturb a system in production.

Set up the project

Follow these steps to set up a PC SDK project.

Step	Action
1.	On the **File** menu in Visual Studio, point to **New** and then click **Project**. Select a **Windows Application** project.
2.	Add the references to the PC SDK assemblies, *ABB.Robotics.dll* and *ABB.Robotics.Controllers.dll*, to the project. The assemblies are located in the installation directory, by default at C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder 5.xx\PC SDK.

3.	Open *Form1.cs* and add the needed namespace statements at the top of the source code page:
	VB:
	`Imports ABB.Robotics`
	`Imports ABB.Robotics.Controllers`
	`Imports ABB.Robotics.Controllers.Discovery`
	`Imports ABB.Robotics.Controllers.RapidDomain`
	
	C#:
	`using ABB.Robotics;`
	`using ABB.Robotics.Controllers;`
	`using ABB.Robotics.Controllers.Discovery;`
	`using ABB.Robotics.Controllers.RapidDomain;`

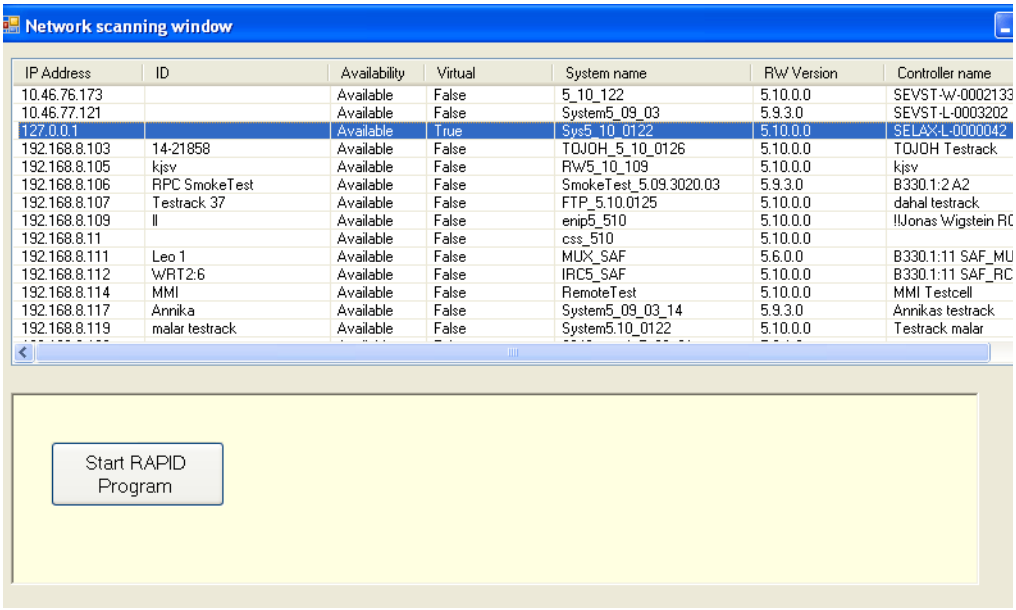
4.	In the **Solution Explorer** right-click *Form1.cs* and select **View Designer**. Create the Graphical User Interface according to the instruction in the next section.

Continues on next page

Continued

Create the user interface

This picture shows the running PC SDK application that we will create. As you see both virtual and real controllers on the network are included in a network scan.



7.1_1

Follow these steps to create the user interface of the application:

Step	Action
1.	Change the **Text** property of the form to “Network scanning window”.
2.	Change its **Size** to 850; 480.
3.	Add a `ListView` control to the form. Set the following properties to get a similar look as in the figure above: **FullRowSelect** - True **GridLines** - True **View** - Details
4.	Add the columns for *IPAdress*, *ID, Availability, Virtual, System name, RW Version* and *Controller name* and adjust the width of the columns.
5.	Add a `Panel` with a `Button` under the listview. Set the `Text` of the button.

Implement network scanning

To find all controllers on the network we start by declaring these member variables in the class `Form1`

VB:

```
Private scanner As NetworkScanner = Nothing
Private controller As Controller = Nothing
Private tasks As Task() = Nothing
Private networkWatcher As NetworkWatcher = Nothing
```


Continues on next page

Continued

C#:

```
private NetworkScanner scanner = null;
private Controller controller = null;
private Task[] tasks = null;
private NetworkWatcher networkwatcher = null;
```

As the application is supposed to scan the network as soon as it is started, we can put the code for it in the `Form1_Load` event handler, like this:

VB:

```
Me.scanner = New NetworkScanner
Me.scanner.Scan()
Dim controllers As ControllerInfoCollection =
     Me.scanner.Controllers
Dim controllerInfo As ControllerInfo = Nothing
Dim item As ListViewItem
For Each controllerInfo In controllers
  item = New ListViewItem(controllerInfo.IPAddress.ToString())
  item.SubItems.Add(controllerInfo.Id)
  item.SubItems.Add(controllerInfo.Availability.ToString())
  item.SubItems.Add(controllerInfo.IsVirtual.ToString())
  item.SubItems.Add(controllerInfo.SystemName)
  item.SubItems.Add(controllerInfo.Version.ToString())
  item.SubItems.Add(controllerInfo.ControllerName)
  Me.listView1.Items.Add(item)
  item.Tag = controllerInfo
Next
```

C#:

```
this.scanner = new NetworkScanner();
this.scanner.Scan();
ControllerInfoCollection controllers = scanner.Controllers;
ListViewItem item = null;
foreach (ControllerInfo controllerInfo in controllers)
{
   item = new ListViewItem(controllerInfo.IPAddress.ToString());
   item.SubItems.Add(controllerInfo.Id);
   item.SubItems.Add(controllerInfo.Availability.ToString());
   item.SubItems.Add(controllerInfo.IsVirtual.ToString());
   item.SubItems.Add(controllerInfo.SystemName);
   item.SubItems.Add(controllerInfo.Version.ToString());
   item.SubItems.Add(controllerInfo.ControllerName);
   this.listView1.Items.Add(item);
   item.Tag = controllerInfo;
 }
```


Continues on next page

Continued

Add a network watcher

By implementing a `NetworkWatcher` the application can supervise the network and detect when controllers are lost or added. This example shows how to program network supervision, and how to add a detected controller to the listview.

After having added a `NetworkWatcher` object to the `FormLoad` event handler, we add a subscription to its `Found` event.

VB:

```
Me.networkWatcher = New NetworkWatcher(Me.scanner.Controllers)
AddHandler Me.networkWatcher.Found, AddressOf Me.HandleFoundEvent
AddHandler Me.networkWatcher.Lost, AddressOf Me.HandleLostEvent
Me.networkWatcher.EnableRaisingEvents = True
```

C#:

```
this.networkwatcher = new NetworkWatcher(scanner.Controllers);
this.networkwatcher.Found += new EventHandler
     <NetworkWatcherEventArgs>(HandleFoundEvent);
this.networkwatcher.Lost += new EventHandler
     <NetworkWatcherEventArgs>(HandleLostEvent);
this.networkwatcher.EnableRaisingEvents = true;
```

NOTE!

In C# the event handler skeleton is auto generated using the Tab key twice after “+=” in the above statements. If you prefer, you can use a simplified syntax when using generic event handlers:

```
networkwatcher.Found += HandleFoundEvent;
```

Handle event

As the events will be received on a background thread and should result in an update of the user interface the `Invoke` method must be called in the event handler. See *Invoke method on page 68* about how to force execution from background to GUI thread.

VB:

```
Private Sub HandleFoundEvent(ByVal sender As Object, ByVal e As
     NetworkWatcherEventArgs)
  Me.Invoke(New NetworkWatcherEventHandler(AddressOf
       AddControllerToListView), New Object() {sender, e})
End Sub
```

C#:

```
void HandleFoundEvent(object sender, NetworkWatcherEventArgs e)
{
  this.Invoke(new
       EventHandler<NetworkWatcherEventArgs>(AddControllerToList
       View), new Object[] { sender, e });
}
```

Continues on next page

Continued

This event handler updates the user interface:

VB:

```
Private Sub AddControllerToListView(ByVal sender
  As Object, ByVal e As NetworkWatcherEventArgs)
  Dim controllerInfo As ControllerInfo = e.Controller
  Dim item As ListViewItem = New ListViewItem(
  controllerInfo.IPAddress.ToString())
  item.SubItems.Add(controllerInfo.Id)
  item.SubItems.Add(controllerInfo.Availability.ToString())
  item.SubItems.Add(controllerInfo.IsVirtual.ToString())
  item.SubItems.Add(controllerInfo.SystemName)
  item.SubItems.Add(controllerInfo.Version.ToString())
  item.SubItems.Add(controllerInfo.ControllerName)
  Me.listView1.Items.Add(item)
  item.Tag = controllerInfo
End Sub
```

C#:

```
private void AddControllerToListView(object sender,
     NetworkWatcherEventArgs e)
{
  ControllerInfo controllerInfo = e.Controller;
  ListViewItem item = new
       ListViewItem(controllerInfo.IPAddress.ToString());
  item.SubItems.Add(controllerInfo.Id);
  item.SubItems.Add(controllerInfo.Availability.ToString());
  item.SubItems.Add(controllerInfo.IsVirtual.ToString());
  item.SubItems.Add(controllerInfo.SystemName);
  item.SubItems.Add(controllerInfo.Version.ToString());
  item.SubItems.Add(controllerInfo.ControllerName);
  this.listView1.Items.Add(item);
  item.Tag = controllerInfo;
}
```

Establish connection to controller

When the user double-clicks a controller in the list a connection to that controller should be established and the user should be logged on. Follow these steps to implement this functionality:

Step	Action
1.	Generate the `DoubleClick` event of the `ListView`.
2.	In the event handler create a `Controller` object that represents the selected robot controller.
3.	Log on to the selected controller. See the code sample of *Implement event handler on page 207*.

Continues on next page

Implement event handler

This example shows the code of the `ListView.DoubleClick` event handler:

```
VB:
Dim item As ListViewItem = Me.listView1.SelectedItems(0)
If Not item.Tag Is Nothing Then
  Dim controllerInfo As ControllerInfo
  controllerInfo = DirectCast(item.Tag, ControllerInfo)
  If controllerInfo.Availability = Availability.Available Then
     If Not Me.controller Is Nothing Then
        Me.controller.Logoff()
        Me.controller.Dispose()
        Me.controller = Nothing
     End If
     Me.controller = ControllerFactory.CreateFrom(controllerInfo)
     Me.controller.Logon(UserInfo.DefaultUser)
  End If
Else
  MessageBox.Show("Selected controller not available.")
End If


C#:
ListViewItem item = this.listView1.SelectedItems[0];
if (item.Tag != null)
{
  ControllerInfo controllerInfo = (ControllerInfo) item.Tag;
  if (controllerInfo.Availability == Availability.Available)
  {
     if (this.controller != null)
     {
        this.controller.Logoff();
        this.controller.Dispose();
        this.controller = null;
     }
     this.controller =
          ControllerFactory.CreateFrom(controllerInfo);
     this.controller.Logon(UserInfo.DefaultUser);
  }
  else
  {
     MessageBox.Show("Selected controller not available.");
  }
}
```


Continues on next page

Continued

NOTE!

The check to see whether the `Controller` object already exists is important, as you should explicitly log off and dispose of any existing controller object before creating a new one. The reason is that a logon session allocates resources that should not be kept longer than necessary.

Start program execution

The `Click` event handler of the **Start RAPID Program** button should start program execution of the first RAPID task.

Starting RAPID execution in manual mode can only be done from the FlexPendant, so we need to check that the controller is in automatic mode before trying. We then need to request mastership of `Rapid` and call the `Start` method. If mastership is already held, by ourselves or another client, an `InvalidOperationException` will be thrown. For further information see *Mastership on page 41*.

It is necessary to release mastership whether or not the start operation succeeds. This can be done by calling `Release()` or `Dispose()` in a finally clause, as shown in the VB example, or by applying the `using` mechanism, as shown in the C# example.

VB:

```
Private m As Mastership = Nothing
...
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
     As System.EventArgs) Handles Button1.Click
  Dim m As ABB.Robotics.Controllers.Mastership = Nothing
  Try
     If controller.OperatingMode = ControllerOperatingMode.Auto
          Then
        tasks = controller.Rapid.GetTasks()
        m = Mastership.Request(controller.Rapid)
        'Perform operation
        tasks(0).Start()
     Else
        MessageBox.Show("Automatic mode is required to start
             execution from a remote client.")
     End IfCatch ex As InvalidOperationException
     MessageBox.Show("Mastership is held by another client.")
  Catch ex As System.Exception
     MessageBox.Show("Unexpected error occurred: " + ex.Message)
  Finally
     If Not m Is Nothing Then
        If (m.IsMaster()) Then
           m.Release()
        End If
     End If
  End Try
End Sub
```

Continues on next page

Continued

C#:

```
private void Button1_Click(object sender, EventArgs e)
{
  try
  {
     if (_controller.OperatingMode ==
          ControllerOperatingMode.Auto)
     {
        tasks = controller.Rapid.GetTasks();
        using (Mastership m =
             Mastership.Request(controller.Rapid))
        {
            //Perform operation
           tasks[0].Start();
        }
     }
     else
     {
        MessageBox.Show("Automatic mode is required to start
             execution from a remote client.");
     }
  }
  catch (System.InvalidOperationException ex)
  {
     MessageBox.Show("Mastership is held by another client.");
  }
  catch (System.Exception ex)
  {
     MessageBox.Show("Unexpected error occurred: " + ex.Message);
  }
}
```

7.3. Discovery domain

Overview

To create a connection to the controller from a PC SDK application it has to make use of the Netscan functionality of the `Discovery` namespace. A `NetworkScanner` object must be created and a `Scan` call must be performed.

For the PC SDK to be able to establish a connection either RobotStudio or Robot Communications Runtime must be installed on the PC hosting the PC SDK application. Robot Communications Runtime can be installed from C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder 5.xx\redistributable\RobotCommunicationRuntime if RobotStudio is not installed.

To find out what controllers are available on the network you use the `NetworkScanner` methods `Scan`, `Find`, `GetControllers` and `GetRemoteControllers`.

NetworkScanner

The `NetworkScanner` class can be declared and instantiated at class level. No scanning is done until the `Scan` method is called. When the `GetControllers` method is called a collection of `ControllerInfo` objects is returned. Each such object holds information about a particular controller connected to the local network. Both virtual and real controllers are detected this way.

VB:

```
Private AScanner As NetworkScanner = New NetworkScanner
...' Somewhere in the code
AScanner.Scan()
Dim ACollection As ControllerInfo() = AScanner.GetControllers()
```

C#:

```
private NetworkScanner aScanner = new NetworkScanner();
... // Somewhere in the code
aScanner.Scan();
ControllerInfo[] aCollection = aScanner.GetControllers();
```

See *Implement network scanning on page 203* for a complete code sample.

If only real controllers are of interest, you can first scan the network and then request only real controllers using the `NetworkScannerSearchCriterias` enumeration in the `GetControllers` method.

VB:

```
Dim ACollection As ControllerInfo() =
      AScanner.GetControllers(NetworkScannerSearchCriterias.Real)
```

C#:

```
ControllerInfo[] aCollection =
      aScanner.GetControllers(NetworkScannerSearchCriterias.Real)
      ;
```


Continues on next page

If you know which controller system you want to connect to you can call the `Find` method, which finds a specified controller on the network. It takes the system ID as a System.Guid data type as argument. The system's globally unique identifier (GUID) can be find in the *system.guid* file in the INTERNAL folder of the robot system file system.

ControllerInfo object

When a network scan is performed a collection of `ControllerInfo` objects is returned. The ControllerInfo object has information about availability. Remember that the ControllerInfo object is not updated when controller status changes. If you again need to find out if a controller is available, you need to perform a new network scan or use an existing `Controller` object and check the status directly.

Add controllers from outside local network

A network scan is done only on the local network. To detect controllers outside the local network you need to supply the IP address of the controller using the static `AddRemoteController` method or configuring it in the *App.config* file. See *PC application configuration on page 43* for further information.

If you supply the controller IP address you either use a string argument or a `System.Net.IPAddress` object.

VB:

```
Dim AnIPAddress As System.Net.IPAddress
Try
  AnIPAddress = System.Net.IPAddress.Parse(Me.TextBox1.Text)
  NetworkScanner.AddRemoteController(AnIPAddress)
Catch ex As FormatException
  Me.TextBox1.Text = "Wrong IP address format"
End Try
```

C#:

```
System.Net.IPAddress ipAddress;
try
{
  ipAddress = System.Net.IPAddress.Parse(Me.textBox1.Text);
  NetworkScanner.AddRemoteController(ipAddress);
}
catch (FormatException ex)
{
  Me.textBox1.Text = "Wrong IP address format";
}
```

NetworkWatcher

By using a `NetworkWatcher` object you can supervise network changes and find out when a new controller is found or when a controller is lost. See *Add a network watcher on page 205* for a complete code example.

7.4. Accessing the controller

Controller object

By using a `Controller` object you can get access to the different domains of the robot controller, e.g. IO signals, RAPID, file system and elog messages.

To create a `Controller` object you normally make a call to the `ControllerFactory`:

VB:

```
Private AController As Controller
AController = ControllerFactory.CreateFrom(info As
     ControllerInfo)
```

C#:

```
private Controller aController;
aController = ControllerFactory.CreateFrom(ControllerInfo info);
```


Continues on next page

The argument is a `ControllerInfo` object, which may have been retrieved during a network scan, see *NetworkScanner on page 210*. It is also possible to add an optional argument if the IP address of the controller or the system ID (guid) should be used.

If the PC application is supposed to work with a single controller it can be specified in an *app.config* file. The default constructor can then be used to create the controller object, e.g. `aController = new Controller()`. See *<defaultSystem> on page 44*for details.

If several classes in your application need to access the controller, it is recommended that they all reference the same `Controller` object. This is done either by passing the `Controller` object as an argument to the constructor or by using a `Controller` property.

NOTE!

You should be aware that the .NET objects created for operations toward the robot controller will access native resources (C++ and COM code). The .NET garbage collector does not collect such objects, but these must be disposed of explicitly by the application programmer. See *Memory management in PC applications on page 213* for further information.

Memory management in PC applications

An important feature of the .NET runtime environment is the garbage collector, which reclaims not referenced memory from the managed heap. Generally, this means that the programmer does not have to free memory that has been allocated by the use of `new`. There is no way of knowing exactly when garbage collection will be performed however.

For a PC application indeterministic deallocation of resources is usually not a problem (as opposed to a FlexPendant application, which runs on a small device with limited memory). The `IDisposable` interface, however, can be used in a PC application to obtain deterministic deallocation of resources. Using this interface you can make an explicit call to the `Dispose` method of any disposable object.

If your application is running in a Single Threaded Apartment (STA) the `Dispose` call will dispose of managed objects, but native resources (created internally by the PC SDK) will remain. To release these native objects, the method `ReleaseUnmanagedResources` should be called periodically, for example when the user presses a certain button or each time data has been written to the controller. The method call is not costly.

For an application running in a Multi Threaded Apartment (MTA) the `Dispose` call will remove both managed and native objects.

NOTE!

The method `Controller.ReleaseUnmanagedResources` should be called once in a while to avoid memory leaks in PC SDK applications running in STA.

Dispose

It is the creator of a disposable object that is responsible for its lifetime and for calling `Dispose`. A check should be done that the object still exists and any subscriptions to controller events should be removed before the `Dispose` call. This is how you dispose of a `Controller` object:

Continues on next page

Continued

VB:

```
If Not AController Is Nothing Then
  RemoveHandler AController.OperatingModeChanged, AddressOf
       UpdateOpMode
  AController.Dispose()
  AController = Nothing
End If
```

C#:

```
if (aController != null)
{
  AController.OperatingModeChanged -= new UpdateOpMode;
  aController.Dispose();
  aController = null;
}
```

Logon and logoff

Before accessing a robot controller the PC SDK application has to log on to the controller. The `UserInfo` parameter of the `Logon` method has a `DefaultUser` property that can be used. By default all robot systems have such a user configured.

VB:

```
AController.Logon(UserInfo.DefaultUser)
```

C#:

```
aController.Logon(UserInfo.DefaultUser);
```

If it is necessary for your application to handle users with different rights, these users can be configured by using the `UserAuthorizationManagement` namespace or by using the UAS administration tool in RobotStudio. This is how you create a new `UserInfo` object for login purposes.

VB:

```
Dim AUserInfo As UserInfo = New UserInfo("name", "password")
```

C#:

```
UserInfo aUserInfo = new UserInfo("name", "password");
```


Continues on next page

Continued

NOTE!

It is necessary to log off from the controller at application shut down at the latest.

VB: `AController.LogOff()`

C#: `aController.LogOff();`

Mastership

In order to get write access to some of the controller domains the application has to request mastership. The `Rapid` domain, i.e. tasks, programs, modules, routines and variables that exist in the robot system, is one such domain. The `Configuration` domain is another.

See *Mastership on page 41* for detailed information on this topic.

It is important to release mastership after a modification operation. One way of doing this is applying the `using` statement, which results in an automatic disposal of the `Mastership` object at the end of the block. Another possibility is releasing mastership in a `Finally` block, which is executed after the `Try` and `Catch` blocks. See how it can be coded in the examples of *Start program execution on page 208*.

Controller events

The `Controller` object provides several public events, which enable you to listen to operating mode changes, controller state changes, mastership changes etc.

VB:

```
AddHandler AController.OperatingModeChanged, AddressOf
    UpdateOpMode
AddHandler AController.StateChanged, AddressOf UpdateState
AddHandler AController.ConnectionChanged, AddressOf UpdateConn
```

C#:

```
AController.OperatingModeChanged += new UpdateOpMode;
AController.StateChanged += new UpdateState;
AController.ConnectionChanged += new UpdateConn;
```


Continues on next page

Continued

NOTE!

Controller events use their own threads. Carefully study *Controller events and threads on page 67* to avoid threading conflicts.

NOTE!

PC SDK 5.09 and onwards uses the generic event handling introduced by .NET Framework 2.0.

CAUTION!

Do not rely on receiving an initial event when setting up/activating a controller event. There is no guarantee an event will be triggered, so you had better read the initial state from the controller.

Backup and Restore

Using the `Controller` object you can call the `Backup` method. The argument is a string describing the directory path on the controller where the backup should be stored. You can also restore a previously backed up system. This requires mastership of `Rapid` and `Configuration` and can only be done in Auto mode.

Backup sample

As the backup process is performed asynchronously you can add an event handler to receive a `BackupCompleted` event when the backup is completed. The backup directory should be created in the system backup directory, or else an exception will be thrown.

VB:

```
Dim BackupDir As String = "(BACKUP)$"+BackupDirName
AddHandler Me.AController.BackupCompleted, AddressOf
     AController_BackupCompleted)
Me.AController.Backup(BackupDir)
```

C#:

```
string backupDir = "(BACKUP)$"+backupDirName;
this.aController.BackupCompleted += new
     BackupEventHandler(controller_BackupCompleted);
this.aController.Backup(backupDir);
```

Restore sample

The `Restore` method is synchronous, i.e. execution will not continue until the restore operation is completed.

VB:

```
Dim RestoreDir As String = "(BACKUP)$"+dirName
Dim M As Mastership
Try
  MC = Mastership.Request(Me.AController.Configuration)
  MR = Mastership.Request(Me.AController.Rapid)
  Me.AController.Restore(RestoreDir, RestoreIncludes.All,
       RestoreIgnores.All)
Finally
  MC.Release()
  MR.Release()
```

Continues on next page

Continued

```
    End Try
```

C#:

```
    string restoreDir = "(BACKUP)$"+dirName;
    using (Mastership mc =
         Mastership.Request(this.aController.Configuration), mr =
         Mastership.Request(this.aController.Rapid))
    {
      this.aController.Restore(restoreDir, RestoreIncludes.All,
           RestoreIgnores.All);
    }
```

VirtualPanel

You can programmatically change the operating mode of the *virtual IRC5* using the `VirtualPanel` class and its `ChangeMode` method. This blocks the application thread until the user manually accepts the mode change to Auto using the Virtual FlexPendant. An alternative to blocking the application thread eternally is to add a time-out and use a `try-catch` block to catch the `TimeoutException`.

VB:

```
    Dim vp As VirtualPanel = VirtualPanel.Attach(AController)
    Try
      vp.ChangeMode(ControllerOperatingMode.Auto, 5000)
    Catch ex As TimeoutException
      Me.TextBox1.Text = "Timeout occurred at change to auto"
    End Try
    vp.Dispose()
```

C#:

```
    VirtualPanel vp = VirtualPanel.Attach(aController);
    try
    {
      vp.ChangeMode(ControllerOperatingMode.Auto, 5000);
    }
    catch (TimeoutException ex)
    {
      this.textBox1.Text = "Timeout occurred at change to auto";
    }
    vp.Dispose();
```

There are also the asynchronous method calls `BeginChangeOperatingMode` and `EndChangeOperatingMode`. It is important to use the second method in the callback since it returns the waiting thread to the thread-pool.

VB:

```
    Dim vp As VirtualPanel = VirtualPanel.Attach(AController)
    vp.BeginChangeOperatingMode(ControllerOperatingMode.Auto, New
         AsyncCallback(AddressOf ChangeMode), vp)
```

Continues on next page

Continued

C#:

```
VirtualPanel vp = VirtualPanel.Attach(aController);
vp.BeginChangeOperatingMode(ControllerOperatingMode.Auto, new
     AsyncCallback(ChangedMode), vp);
```

The callback method must have the following signature and call the `EndChangeOperatingMode` as well as dispose the `VirtualPanel`.

VB:

```
Private Sub ChangeMode(ByVal iar As IAsyncResult)
  Dim vp As VirtualPanel = DirectCast(iar.AsyncState,
       VirtualPanel)
  vp.EndChangeOperatingMode(iar)
  vp.Dispose()
  .....
```

C#:

```
private void ChangedMode(IAsyncResult iar)
{
  VirtualPanel vp = (VirtualPanel) iar.AsyncState;
  vp.EndChangeOperatingMode(iar);
  vp.Dispose();
  ....
```

Learn more

This User's Guide only covers some of the PC SDK functionality. To get the full potential of the PC SDK you should make use of the *PC SDK Reference* located in the PC SDK installation directory. See *PC SDK Reference on page 201*.

You can also learn a lot by becoming an active member of the *RobotStudio Community*. Its Robot Application Builder *User Forum* should be your number one choice when you find yourself stuck with a coding issue you cannot solve on your own. See *RobotStudio Community on page 17* for further information.

7.5 Rapid domain

7.5.1. Working with RAPID data

Overview

The `RapidDomain` namespace enables access to RAPID data in the robot system. There are numerous PC SDK classes representing the different RAPID data types. There is also a `UserDefined` class used to reference RECORD structures in RAPID.

The `ValueChanged` event enables notification from the controller when persistent RAPID data has changed.

To speed up event notification from the controller there is new functionality in PC SDK 5.10, which allows you to set up subscription priorities. This possibility applies to I/O signals and persistent RAPID data. This mechanism is further described in *Implementing high priority data subscriptions on page 226*.

NOTE!

To read RAPID data you need to log on to the controller. To modify RAPID data you must also request mastership of the `Rapid` domain.

Providing the path to the RAPID data

To read or write to RAPID data you must first create a `RapidData` object. The path to the declaration of the data in the controller is passed as argument. If you don't know the path you need to search for the RAPID data by using the `SearchRapidSymbol` functionality. See *RAPID symbol search on page 157* for further information.

Direct access

Direct access requires less memory and is faster, and is therefore recommended if you do not need to use the task and module objects afterwards.

The example below shows how to create a `RapidData` object that refers to the instance "reg1" in the USER module.

VB:

```
Dim Rd As RapidData = Me.AController.Rapid.GetRapidData(
    "T_ROB1", "USER", "reg1")
```

C#:

```
RapidData rd = aController.Rapid.GetRapidData("T_ROB1", "USER",
    "reg1");
```

Hierarchical access

If you need the task and module objects hierarchical access can be more efficient. `GetRapidData` exists in the Rapid, Task and Module class.

VB:

```
Rd = AController.Rapid.GetTask("T_ROB1").GetModule("USER").
    GetRapidData("reg1")
```

C#:

```
rd = aController.Rapid.GetTask("T_ROB1").GetModule("USER").
    GetRapidData("reg1");
```

Continues on next page

Continued

Accessing data declared in a shared module

If your application is to be used with a multi-move system (one controller and several motion tasks/robots), it may happen that the RAPID instance you need to access is declared in a -*Shared* RAPID module. Such a module can be used by all tasks, T_ROB1, T_ROB2 etc.

This example shows how to create a `RapidData` object that refers to the instance “reg100”, which is declared in a shared module.

C#:

```
Task tRob1 = aController.Rapid.GetTask("T_ROB1");
if (tRob1 != null)
{
  RapidData rData = tRob1.GetRapidData("reg100");
}
```

NOTE!

If the data is declared in a -Shared **-Hidden** module it cannot be accessed by the PC SDK.

Creating an object representing the RAPID data value

The `RapidData` object stores the path to the RAPID data. But this is not enough if you want to access its value (at least not if you want to *modify* it). To do that you need to create another object, which represents the *value* of the RAPID data.

In the `RapidDomain` namespace there are types representing the different RAPID data types. To create the object needed to represent the RAPID data value you use the `RapidData` property `Value` and cast it to the corresponding type, e.g. `Num`, `Bool` or `Tooldata`.

This is how this is done if you want to access the value of a RAPID data of the RAPID data type `bool`:

VB:

```
'declare a variable of data type RapidDomain.Bool
Dim rapidBool As RapidDomain.Bool
Dim rd As RapidData = Me.AController.Rapid.GetRapidData( "T_ROB1",
     "MainModule", "flag" )
'test that data type is correct before cast
If TypeOf rd.Value Is RapidDomain.Bool Then
  rapidBool = DirectCast(rd.Value, RapidDomain.Bool)
  'check if the value of the RAPID data is true
  If (rapidBool.Value) Then
     ' Do something...
  End If
EndIf
```


Continues on next page

Continued

C#:

```
//declare a variable of data type RapidDomain.Bool
RapidDomain.Bool rapidBool;
RapidDomain.RapidData rd =
     aController.Rapid.GetRapidData("T_ROB1", "MainModule",
     "flag");
//test that data type is correct before cast
if (rd.Value is ABB.Robotics.Controllers.RapidDomain.Bool)
{
  rapidBool =
       (ABB.Robotics.Controllers.RapidDomain.Bool)rd.Value;
  //assign the value of the RAPID data to a local variable
  bool boolValue = rapidBool.Value;
}
```

If you just want to *read* this variable you can use this technique instead of creating a RapidDomain.`Bool` object:

VB:

```
Dim b As Boolean = Convert.ToBoolean(rd.Value.ToString)
```

C#:

```
bool b = Convert.ToBoolean(rd.Value.ToString());
```

The .NET type `ToolData` (representing the RAPID data type `tooldata`) can be created like this:

VB:

```
Dim ATool As ToolData
If Rd.Value Is ToolData Then
  ATool = DirectCast(Rd.Value, ToolData)
End If
```

C#:

```
ToolData aTool;
if (rd.Value is ToolData)
{
  aTool = (ToolData) rd.Value;
}
```

IRapidData.ToString method

All RapidDomain structures representing RAPID data types implement the `IRapidData` interface. It has a `ToString` method, which returns the value of the RAPID data in the form of a string. This is a simple example:

```
string boolValue = rapidBool.ToString();
```


Continues on next page

Continued

The string is formatted according to the principle described in *IRapidData.FillFromString method on page 222*.

Below is an example of a more complex data type. The `ToolData` `Tframe` property is of type `Pose`. Its `Trans` value is displayed in a label in the format [x, y, z].

VB:

```
Me.Label1.Text = ATool.Tframe.Trans.ToString()
```

C#:

```
this.label1.Text = aTool.Tframe.Trans.ToString();
```

IRapidData.FillFromString method

The `IRapidData` interface also has a `FillFromString` method, which fills the object with a valid RAPID string representation. The method can always be used when you need to modify RAPID data. Using the method with the `RapidDomain.Bool` variable used earlier in the chapter will look like this:

```
rapidBool.FillFromString("True")
```

Using it for a `RapidDomain.Num` variable is similar:

```
rapidNum.FillFromString("10")
```

String format

The format is constructed recursively. An example is the easiest way of illustrating this.

Example:

The `RapidDomain.Pose` structure represents the RAPID data type `pose`, which describes how a coordinate system is displaced and rotated around another coordinate system.

```
public struct Pose : IRapidData
{
  public Pos trans;
  public Orient rot;
}
```

This is an example in RAPID:

```
VAR pose frame1;
...
frame1.trans := [50, 0, 40];
frame1.rot := [1, 0, 0, 0];
```

The frame1 coordinate transformation is assigned a value that corresponds to a displacement in position where X=50 mm, Y=0 mm and Z=40 mm. There is no rotation.

The `RapidDomain.Pose` structure consists of two struct variables called *trans* and *rot* of the data types `Pos` and `Orient`. `Pos` has three floats and `Orient` consists of four doubles. The `FillFromString` format for a `Pose` object is “[[1.0, 0.0, 0.0, 0.0][10.0, 20.0, 30.0]]”.

The example shows how to write a new value to a RAPID `pose` variable:

VB:

```
If TypeOf rd.Value Is Pose Then
  Dim rapidPose As Pose = DirectCast(rd.Value, Pose)
  rapidPose.FillFromString("[[1.0, 0.0, 0.0, 0.0][10, 20, 30]]")
  rd.Value = rapidPose
```

Continues on next page

Continued

```
    End If
```

C#:

```
    if (rd.Value is Pose)
    {
      Pose rapidPose = (Pose) rd.Value;
      rapidPose.FillFromString("[[1.0, 0.5, 0.0, 0.0][10, 15, 10]]");
      rd.Value = rapidPose;
    }
```

NOTE!

The string format must be carefully observed. If the string argument has the wrong format, a `RapidDataFormatException` is thrown.

Writing to RAPID data

Writing to RAPID data is only possible using the type cast `RapidData` value, to which the new value is assigned. To write the new value to the RAPID data in the controller you must then assign the .Net object to the `Value` property of the `RapidData` object. This example uses the rapidBool object created in *Creating an object representing the RAPID data value on page 220*.

VB:

```
    'Assign new value to .Net variable
    rapidBool.Value = False
    'Request mastership of Rapid before writing to the controller
    Me.master = Mastership.Request(Me.AController.Rapid)
    'Write the new value to the data in the controller
    rd.Value = rapidBool
    'Release mastership as soon as possible
    Me.master.Dispose
```

C#:

```
    //Assign new value to .Net variable
    rapidBool.Value = false;
    //Request mastership of Rapid before writing to the controller
    this.master = Mastership.Request(this.controller.Rapid);
    rd.Value = rapidBool;
    //Release mastership as soon as possible
    this.master.Dispose();
```


Continues on next page

Continued

See *Mastership on page 41* for detailed information about on how the controller handles write access and *Start program execution on page 208* for another code example of implementing mastership in a PC SDK application.

This was an easy example, as the value to change was a simple `bool`. Often, however, RAPID uses complex structures. By using the `FillFromString` method you can assign a new `Value` to any `RapidData` and write it to the controller.

The string must be formatted according to the principle described in the previous section. The following example shows how to write a new value to the `pos` structure (x, y, z) of a RAPID tooldata:

```
VB:
Dim APos As Pos = New Pos
APos.FillFromString("[2,3,3]")
Me.ATool.Tframe.Trans = APos
Me.Rd.Value = Me.ATool

C#:
Pos aPos = new Pos();
aPos.FillFromString("[2,3,3]");
this.aTool.Tframe.Trans = aPos;
this.rd.Value = this.aTool;
```

NOTE!

The new value is not written to the controller until the last statement is executed.

Letting the user know that RAPID data has changed

In order to be notified that RAPID data has changed you need to add a subscription to the `ValueChanged` event of the `RapidData` instance. Note, however, that this only works for **persistent** RAPID data.

Add subscription

This is how you add a subscription to the `ValueChanged` event:

VB:

```
Addhandler Rd.ValueChanged, AddressOf Rd_ValueChanged
```

C#:

```
this.rd.ValueChanged += rd_ValueChanged;
```


Continues on next page

Continued

Handle event

The implementation of the event handler may look like this. Remember that controller events use their own threads, and avoid Winforms threading problems by the use of `Control.Invoke`, which forces the execution from the background thread to the GUI thread.

VB:

```
Private Sub Rd_ValueChanged(ByVal sender As Object, ByVal e As
     DataValueChangedEventArgs)
  Me.Invoke(New EventHandler (AddressOf UpdateGUI), sender, e)
End Sub
```

C#

```
private void rd_ValueChanged(object sender,
     DataValueChangedEventArgs e)
{
  this.Invoke(new EventHandler (UpdateGUI), sender, e);
}
```

See *Controller events and threads on page 67* to learn more about potential threading conflicts in RAB applications.

Read new value from controlller

Update the user interface with the new value. As the value is not part of the event argument, you must use the `RapidData Value` property to retrieve the new value:

VB:

```
Private Sub UpdateGUI(ByVal sender As Object, ByVal e As
     System.EventArgs)
  Dim Tool1 As ToolData = DirectCast(Me.Rd.Value, ToolData)
  Me.Label1.Text = Tool1.Tframe.Trans.ToString()
End Sub
```

C#

```
private void UpdateGUI(object sender, System.EventArgs e)
{
  ToolData tool1= (ToolData)this.rd.Value;
  this.label1.Text = tool1.Tframe.Trans.ToString();
}
```


Continues on next page

Continued

NOTE!

Subscriptions work only for RAPID data declared as PERS.

Implementing high priority data subscriptions

To speed up event notification from the controller it is possible to set up subscription priorities for persistent RAPID data. To do this you use the `Subscribe` method and the enumeration `EventPriority` as argument. The example shows an ordinary signal subscription and a subscription with high priority:

VB:

```
Addhandler Rd.ValueChanged, AddressOf Rd_ValueChanged
Rd.Subscribe(AddressOf Rd_Changed, EventPriority.High)
```

C#:

```
rd.ValueChanged += rd_ValueChanged;
rd.Subscribe(rd_Changed, EventPriority.High);
```

To deactivate subscriptions with high priority you call the `Unsubscribe` method like this:

VB:

```
Rd.Unsubscribe(AddressOf Rd_ValueChanged)
```

C#:

```
rd.Unsubscribe(rd_Changed);
```

NOTE!

High priority subscriptions can be used for I/O signals and RAPID data declared PERS. The controller can handle 64 high priority subscriptions.

RapidData disposal

You are recommended to dispose of `RapidData` objects when they are no longer needed. See *Memory management in PC applications on page 213* for further information.

VB:

```
If Not Rd Is Nothing Then
  Rd.Dispose()
  Rd = Nothing
End If
```

C#:

```
if (rd != null)
{
  rd.Dispose();
  rd = null;
}
```

7.5.2. Handling arrays

Overview

In RAPID you can have up to three dimensional arrays. These are accessible by using a `RapidData` object like for any other RAPID data.

There are mainly two ways of accessing each individual element of an array: by indexers or by an enumerator.

ArrayData object

If the `RapidData` references a RAPID array is `Value` property returns an object of `ArrayData` type. Before making a cast, check the type using the `is` operator or by using the `IsArray` property on the `RapidData` object.

VB:

```
Dim RD As RapidData = AController.Rapid.GetRapidData("T_ROB1",
     "User", "string_array")
If RD.IsArray Then
  Dim AD As ArrayData = DirectCast( RD.Value,ArrayData)
  .....
End If
```

C#:

```
RapidData rd = aController.Rapid.GetRapidData("T_ROB1", "User",
     "string_array");
if (rd.IsArray)
{
  ArrayData ad = (ArrayData)rd.Value;
  .....
}
```

Array dimensions

The dimension of the array is returned by the Rank property. If you need to check the length of the individual arrays you can use the `GetLength` method on the `ArrayData` object passing the dimension index as argument.

VB:

```
Dim ARank As Integer = AD.Rank
Dim Len As Integer = AD.GetLength(ARank)
```

C#:

```
int aRank = ad.Rank;
int len = ad.GetLength(aRank);
```


Continues on next page

Continued

Array item access by using indexers

By the use of indexers you can access each array element, even in three dimensional arrays. A combination of the `GetLength` method and `For` loops makes it possible to access any item:

VB:

```
Dim ASum As Double = 0R
Dim ANum As Num
If AD.Rank = 1 Then
  For I As Integer = 1 To AD.Length
     ANum = DirectCast(ad.[I], Num)
     ASum +=  DirectCast(ANum, Double)
  Next
ElseIf AD.Rank = 2 Then
  For I As Integer = 1 To AD.GetLength(1)
     For J As Integer = 1 To AD.GetLength(2)
        ANum = DirectCast(ad[I, J], Num)
        ASum +=  DirectCast(ANum, Double)
     Next
  Next
Else
  For I As Integer = 1 To AD.GetLength(1)
     For J As Integer = 1 To AD.GetLength(2)
        For K As Integer = 1 To AD.GetLength(3)
           ANum = DirectCast(ad[I, J, K], Num)
           ASum +=  DirectCast(ANum, Double)
        Next
     Next
  Next
End If
```

C#:

```
double sum = 0d;
Num aNum;
if (ad.Rank == 1) {
  for (int i = 1; i <= ad.Length; i++) {
     aNum = (Num)ad.[i];
     aSum +=  (double)ANum;
  }
}
elseif (ad.Rank == 2) {
  for(int i = 1; i< ad.GetLength(1); i++) {
     for (int j = 1; j <= ad.Length; j++) {
        aNum = (Num)ad.[i,j];
        aSum +=  (double)ANum;
     }
  }
```

Continues on next page

Continued

```
}
else {
  for(int i = 1; i< ad.GetLength(1); i++) {
     for(int j = 1; j< ad.GetLength(2); j++) {
        for (int k = 1; k <= ad.GetLength(3); k++) {
           aNum = (Num)ad.[i, j, k];
           aSum +=  (double)ANum;
        }
     }
  }
}
```

Array item access using enumerator

You can also use the enumerator operation (`foreach`) like it is used by collections in .NET. Notice that it can be used for both one dimension and multi-dimensional arrays to access each individual element. The previous example is a lot simpler this way:

VB:

```
Dim ASum As Double = 0R
Dim ANum As Num
For Each ANum As Num In AD
  ASum +=  DirectCast(ANum, Double)
Next
```

C#:

```
double sum = 0d;
Num aNum;
foreach(Num aNum in ad)
{
  aSum +=  (double)ANum;
}
```

7.5.3. ReadItem and WriteItem methods

Overview

An alternative way of accessing RAPID data stored in an array are the `ReadItem` and `WriteItem` methods.

ReadItem method

Using the `ReadItem` method you can directly access a `RapidData` item in an array, e.g. an array with *RobTargets* or *Nums*. The index to the item is explicitly specified in the `ReadItem` call. The first item is in position 1, i.e. the array is 1-based as in RAPID.

VB:

```
Dim ANum As Num
aNum = DirectCast(rd.ReadItem(1, 2), Num)
```

C#:

```
Num aNum = (Num)rd.ReadItem(1, 2);
```

This example retrieves the second *Num* value in the first array of the RAPID data variable referenced by rd.

WriteItem method

In the same manner it is possible to use the `WriteItem` method to write to an individual RAPID data item in an array. This example shows how to write the result of an individual robot operation into an array representing a total robot program with several operations:

VB:

```
Dim ANum As Num = New Num(OPERATION_OK)
rd.WriteItem(ANum, 1, 2)
```

C#:

```
Num aNum = new Num(OPERATION_OK);
rd.WriteItem(aNum, 1, 2);
```

NOTE!

If the index is out of bounds an `IndexOutOfRangeException` will be thrown.

7.5.4. UserDefined data

Overview

RECORD structures are common in RAPID code. To handle these unique data types a `UserDefined` class is available. This class has properties and methods to handle individual components of a RECORD.

In some cases implementing your own structure can improve application design and code maintenance.

Creating UserDefined object

The `UserDefined` constructor takes a `RapidDataType` object as argument. To retrieve a `RapidDataType` object you need to provide a `RapidSymbol` or the path to the declaration of the RAPID data type.

This example creates a `UserDefined` object representing the RAPID RECORD *processdata*:

VB:

```
Dim rdt As RapidDataType
rdt = Me.controller.Rapid.GetRapidDataType("T_ROB1", "MyModule",
      "processdata")
Dim processdata As UserDefined = New UserDefined(rdt)
```

C#

```
RapidDataType rdt;
rdt = this.controller.Rapid.GetRapidDataType("T_ROB1",
      "MyModule", "processdata");
UserDefined processdata = new UserDefined(rdt);
```

Reading UserDefined data

`UserDefined` can be used to read any kind of RECORD variable from the controller. The individual components of the RECORD are accessible using the `Components` property and an index. Each `Component` can be read as a string.

VB:

```
Dim processdata As UserDefined = DirectCast(rd.Value, UserDefined)
Dim No1 As String = processdata.Components(0).ToString()
Dim No2 AS String = processdata.Components(1),ToString()
```

C#:

```
UserDefined processdata = (UserDefined) rd.Value;
string no1 = processdata.Components[0].ToString();
string no2 = processdata.Components[1].ToString();
```

Each individual string can then be used in a `FillFromString` method to convert the component into a specific data type, e.g. `RobTarget` or `ToolData`. See *IRapidData.FillFromString method on page 222* for details.

Continues on next page

Continued

Writing to UserDefined data

If you want to modify `UserDefined` data and write it to the controller you must first read the `UserDefined` object and the apply new values using the `FillFromString` method. Then you need to perform a write operation using the `RapidData.Value` property.

VB:

```
processdata.Components(0).FillFromString("[0,0,0]")
processdata.Components(1).FillFromString("10")
rd.Value = ud
```

C#:

```
processdata.Components[0].FillFromString("[0,0,0]");
processdata.Components[1].FillFromString("10");
rd.Value = ud;
```

See *IRapidData.FillFromString method on page 222* and *Writing to RAPID data on page 223* for further information and code samples.

Recursively reading the structure of any RECORD data type

If you need to know the structure of a RECORD data type (built-in or user-defined) you must first retrieve the *record components* of the record. Then you need to iterate the record components and check if any of them are also records. This procedure must be repeated until all record components are atomic types.This code example shows how to get information about the *robtarget* data type. The *robtarget* URL is “RAPID/robtarget” or just “robtarget”.

```
private void SearchRobtarget()
{
  RapidSymbol[ ] rsCol = tRob1.SearchRapidSymbol(sProp, "RAPID/
       robtarget", "p10");
  RapidDataType theDataType;
  if (rsCol.Length > 0)
  {
     Console.WriteLine("RapidSymbol name = " + rsCol[0].Name);
     theDataType = RapidDataType.GetDataType(rsCol[0]);
     Console.WriteLine("DataType = " + theDataType.Name);
     if (theDataType.IsRecord)
     {
        RapidSymbol[] syms = theDataType.GetComponents();
        SearchSymbolStructure(syms);
     }
  }
}
```


Continues on next page

Continued

```
private void SearchSymbolStructure(RapidSymbol[] rsCol)
{
  RapidDataType  theDataType;
  foreach (RapidSymbol rs in rsCol)
  {
     Console.WriteLine("RapidSymbol name = " + rs.Name);
     theDataType = RapidDataType.GetDataType(rs);
     Console.WriteLine("DataType = " + theDataType.Name);
     if (theDataType.IsRecord)
     {
        RapidSymbol[] syms = theDataType.GetComponents();
        SearchSymbolStructure(syms);
     }
  }
}
```


Continues on next page

Continued

The code example above produces the following printout:

```
RapidSymbol name = p10
DataType = robtarget

RapidSymbol name = trans
DataType = pos
RapidSymbol name = x
DataType = num
RapidSymbol name = y
DataType = num
RapidSymbol name = z
DataType = num

RapidSymbol name = rot
DataType = orient
RapidSymbol name = q1
DataType = num
RapidSymbol name = q2
DataType = num
RapidSymbol name = q3
DataType = num
RapidSymbol name = q4
DataType = num

RapidSymbol name = robconf
DataType = confdata
RapidSymbol name = cf1
DataType = num
RapidSymbol name = cf4
DataType = num
RapidSymbol name = cf6
DataType = num
RapidSymbol name = cfx
DataType = num

RapidSymbol name = extax
DataType = extjoint
RapidSymbol name = eax_a
```

Continues on next page

Continued

DataType = num

RapidSymbol name = eax_b

DataType = num

RapidSymbol name = eax_c

DataType = num

RapidSymbol name = eax_d

DataType = num

RapidSymbol name = eax_e

DataType = num

RapidSymbol name = eax_f

DataType = num

Implement your own struct representing a RECORD

This example shows how you can create your own .NET data type representing a RECORD in the controller instead of using the `UsefDefined` type.

Creating ProcessData type

VB:

```
Dim rdt As RapidDataType = Me.ARapid.GetRapidDataType("T_ROB1",
      "MyModule", "processdata")
Dim pc As ProcessData = New ProcessData(rdt)
pc.FillFromString(rd.Value.ToString())
```

C#

```
RapidDataType rdt = this.aRapid.GetRapidDataType("T_ROB1",
      "MyModule", "processdata");
ProcessData pc = new ProcessData(rdt);
pc.FillFromString(rd.Value.ToString());
```

Implementing ProcessData struct

This example shows how the new data type `ProcessData` may be implemented. As you see, this is done by using a .NET struct and letting `ProcessData` wrap the `UserDefined` object.

The struct implementation should include a `FillFromString` and `ToString` method, i.e. inherit the `IRapidData` interface. Any properties and methods may also be implemented.

VB:

```
Public Structure ProcessData
  Implements IRapidData
  Private data As UserDefined

  Public Sub New(ByVal rdt As RapidDataType)
     data = New UserDefined(rdt)
  End Sub

  Private Property IntData() As UserDefined
  Get
     Return data
```

Continues on next page

7.5.4. UserDefined data

Continued

```
  End Get

  Set(ByVal Value As UserDefined)
     data = Value
  End Set
  End Property
  .....
End Structure

C#:
  public struct ProcessData: IRapidData
  {
     private UserDefined data;

     public ProcessData(RapidDataType rdt)
     {
        data = new UserDefined(rdt);
     }
     private UserDefined IntData
     {
        get { return data; }
        set { data = value; }
     }

     public int StepOne
     {
        get
        {
           int res =
                Convert.ToInt32(IntData.Components[0].ToString())
                ;
           return res;
        }
        set
        {
           IntData.Components[0] = new Num(value);
        }
     }
  }
```

Implementing IRapidData methods

This piece of code shows how the two `IRapidData` methods `ToString` and `FillFromString` can be implemented.

VB:

```
Public Sub FillFromString(ByVal newValue As String) Implements
     ABB.Robotics.Controllers.RapidDomain.IRapidData.FillFromStr
     ing
  IntData.FillFromString(newValue)
```

Continues on next page

Continued

```
End Sub

Public Overrides Function ToString() As String Implements
     ABB.Robotics.Controllers.RapidDomain.IRapidData.ToString
  Return IntData.ToString()
End Function
```

C#:

```
public void FillFromString(string newValue)
{
  IntData.FillFromString(newValue);
}

public override string ToString()
{
  return IntData.ToString();
}
```

NOTE! The `ToString` method has to use the `Overrides` keyword in Visual Basic and the `override` keyword in C#.

Property implementation

Each item in the RECORD structure should have a corresponding property in the extended .NET data type. The get and set methods have to implement the conversion from/to controller data type to .NET data type.

VB:

```
Public Property Step() As Integer
  Get
     Dim res As Integer =
          Convert.ToInt32(IntData.Components(0).ToString())
     Return res
  End Get
  Set(ByVal Value As Integer)
     Dim tmp As Num = New Num
     tmp.FillFromNum(Value)
     IntData.Components(0) = tmp
  End Set
End Property
```

C#:

```
  public int Step
  {
     get
     {
        int res =
             Convert.ToInt32(IntData.Components[0].ToString());
        return res;
```

Continues on next page

7.5.4. UserDefined data

Continued

```
        }
        set
        {
            Num tmp = new Num();
            tmp.FillFromNum(value);
            IntData.Components[0] = tmp;
        }
    }
```

7.5.5. RAPID symbol search

Overview

Most RAPID elements (variables, modules, tasks, records etc.) are members of a symbol table, in which their names are stored as part of a program tree structure.

It is possible to search this table and get a collection of `RapidSymbol` objects, each one including the RAPID object name, location and type.

Search method

The search must be configured carefully, due to the large amount of RAPID symbols in a system. To define a query you need to consider from where in the program tree the search should be performed, which symbols are of interest and what information you need for the symbols of interest.To enable search from different levels the `SearchRapidSymbol` method is a member of several different SDK classes, e.g. `Task`, `Module` and `Routine`. This example shows a search performed with `Task` as the starting point:

VB:

```
Dim RSCol As RapidSymbol()
RSCol = ATask.SearchRapidSymbol(SProp, "num", string.Empty)
```

C#:

```
RapidSymbol[] rsCol;
rsCol = aTask.SearchRapidSymbol(sProp, "num", string.Empty);
```

The `SearchRapidSymbol` method has three arguments. The first argument, of data type `RapidSymbolSearchProperties`, is detailed in the next section. The second and third arguments are detailed in the following sections.

Search properties

The `RapidSymbolSearchProperties` type is rather complex and requires some knowledge about RAPID concepts.

It is used to specify search method, type of RAPID symbol to search for, whether the search should be recursive, whether the symbols are local and/or global and whether or not the search result should include only symbols currently used by a program. If a property is not valid for a particular symbol, it will be discarded and will not exclude the symbol from the search result.

The table describes the different properties of `RapidSymbolSearchProperties`.

Property	Description
SearchMethod	Specifies the direction of the search, which can be `Block` (down) or `Scope` (up). Example: If the starting point of the search is a routine, a block-search will return the symbols declared within the routine, whereas a scope-search will return the symbols accessible from the routine.

Continues on next page

Continued

Property	Description
Types	Specifies which RAPID type(s) you want to search for. The `SymbolTypes` enumeration includes `Constant`, `Variable`, `Persistent`, `Function`, `Procedure`, `Trap`, `Module`, `Task`, `Routine`, `RapidData`. etc. (`Routine` includes `Function`, `Procedure` and `Trap`. `RapidData` includes `Constant`, `Variable` and `Persistent`.)
Recursive	For both block and scope search it is possible to choose if the search should stop at the next scope or block level or recursively continue until the root (or leaf) of the symbol table tree is reached.
GlobalSymbols	Specifies whether global symbols should be included.
LocalSymbols	Specifies whether local symbols should be included.
InUse	Specifies whether only symbols in use by the loaded RAPID program should be searched.

Default instance

`RapidSymbolSearchProperties` has several static methods that return a default instance.

VB:

```
Dim SProp As RapidSymbolSearchProperties =
     RapidSymbolSearchProperties.CreateDefault()
```

C#:

```
RapidSymbolSearchProperties sProp =
     RapidSymbolSearchProperties.CreateDefault();
```

The default instance has the following values:.

Property	Description
SearchMethod	SymbolSearchMethod.Block
Types	SymbolTypes.NoSymbol
Recursive	True
GlobalSymbols	True
LocalSymbols	True
InUse	True

Using this instance you can specify the search properties of the search you want to perform.

Example:

VB:

```
SProp.SearchMethod = SymbolSearchMethod.Scope
SProp.Types = SymbolTypes.Constant Or SymbolTypes.Persistent
SProp.Recursive = False
```


Continues on next page

Continued

C#:

```
sProp.SearchMethod = SymbolSearchMethod.Scope;
sProp.Types = SymbolTypes.Constant | SymbolTypes.Persistent
sProp.Recursive = false;
```


Continues on next page

Continued

NOTE!

The default instance has the property `Types` set to `NoSymbol`. It must be specified in order for a meaningful search to be performed!

NOTE!

The `Types` property allows you to combine several types in a search. See the example above.

NOTE!

See *PC SDK Reference* for the static methods `CreateDefaultForData` and `CreateDefaultForRoutine`.

Data type argument

The second argument of the `SearchRapidSymbol` method is the RAPID data type written as a string. The data type should be written with small letters, e.g. “num”, “string” or “robtarget”. It can also be specified as `string.Empty`.

NOTE!

To search for a `UserDefined` data type the *complete* path to the module that holds the RECORD definition must be passed, like this:

```
result = tRob1.SearchRapidSymbol(sProp,"RAPID/T_ROB1/MyModule/
MyDataType", string.Empty);
```

However, if `MyModule` is configured as *-Shared* the system sees its data types as installed, and the task or module should **not** be included in the path

```
result = tRob1.SearchRapidSymbol(sProp,"MyDataType", string.Empty);
```

Symbol name argument

The third argument is the name of the RAPID symbol. It can be specified as `string.Empty` if the name of the symbol to retrieve is not known, or if the purpose is to search ALL “num” data in the system for example.

Instead of the name of the RAPID symbol a regular expression can be used. The search mechanism will then match the pattern of the regular expression with the symbols in the symbol table. The regular expression string is not case sensitive

A regular expression is a powerful mechanism. It may consist of ordinary characters and meta characters. A meta character is an operator used to represent one or several ordinary characters, and the purpose is to extend the search.

Within a regular expression, all alphanumeric characters match themselves, i.e. the pattern “abc” will only match a symbol named “abc”. To match all symbol names containing the character sequence “abc”, it is necessary to add some meta characters. The regular expression for this is “.*abc.*”.

The available meta character set is shown below:

Expression	Meaning
.	Any single character

Continues on next page

Continued

Expression	Meaning
^	Any symbol starting with
[s]	Any single character in the non-empty set s, where s is a sequence of characters. Ranges may be specified as c-c.
[^s]	Any single character not in the set s.
r*	Zero or more occurrences of the regular expression r.
r+	One or more occurrences of the regular expression r.
r?	Zero or one occurrence of the regular expression r.
(r)	The regular expression r. Used for separate that regular expression from another.
r | r'	The regular expressions r or r'.
.*	Any character sequence (zero, one or several characters).

Example 1

"^c.*"

Returns all symbols starting with c or C.

Example 2

"^reg[1-3]"

Returns reg1, Reg1, REG1, reg2, Reg2, REG2, reg3, Reg3 and REG3.

Example 3

"^c.*|^reg[1,2]"

Returns all symbols starting with c or C as well as reg1, Reg1, REG1, reg2, Reg2 and REG2.

SearchRapidSymbol example

This example searches for VAR, PERS or CONST num data in a task and its modules. The search is limited to globally declared symbols. By default the search method is `Block`, so it does not have to be set.

VB:

```
Dim SProp As RapidSymbolSearchProperties =
     RapidSymbolSearchProperties.CreateDefault()
SProp.Types = SymbolTypes.RapidData
SProp.LocalSymbols = False
Dim RSCol As RapidSymbol()
RSCol = ATask.SearchRapidSymbol(SProp, "num", string.Empty)
```

C#:

```
RapidSymbolSearchProperties sProp =
     RapidSymbolSearchProperties.CreateDefault();
sProp.Types = SymbolTypes.RapidData;
sProp.LocalSymbols = false;
RapidSymbol[] rsCol;
rsCol = aTask.SearchRapidSymbol(sProp, "num", string.Empty);
```

Continues on next page

Continued

Search for UserDefined RAPID data - example

In this example a user defined RECORD data type (“mydata”) is declared in a module (“myModule”). Assuming that the end-user can declare and use data of this data type in any program module, the search method must be Block (default). A search for all “mydata” instances may look like this:

VB:

```
Dim SProp As RapidSymbolSearchProperties =
     RapidSymbolSearchProperties.CreateDefault()
SProp.Types = SymbolTypes.RapidData
Dim RSCol As RapidSymbol()
RSCol = ATask.SearchRapidSymbol(SProp, "RAPID/T_ROB1/myModule/
     mydata", string.Empty)
```

C#:

```
RapidSymbolSearchProperties sProp =
     RapidSymbolSearchProperties.CreateDefault();
sProp.Types = SymbolTypes.RapidData;
RapidSymbol[] rsCol;
rsCol = aTask.SearchRapidSymbol(sProp, "RAPID/T_ROB1/myModule/
     mydata", string.Empty);
```

NOTE!

If *myModule* is configured as -*Shared* and all *myData* instances are declared in *myModule* the search method must be set to `Scope` and the `SearchRapidSymbol` call should look like this:

```
rsCol = aTask.SearchRapidSymbol(sProp, "mydata", string.Empty);
```

7.5.6. Working with RAPID modules and programs

Overview

Using the `Task` object it is possible to load and save individual modules and programs. You can also unload programs, as well as reset the program pointer and start program execution.

NOTE!

All these operations require mastership of the RAPID domain. See *Mastership on page 215* for details.

Load modules and programs

To load a module or program file you need the path to the file on the controller. When the file is loaded into memory the `RapidLoadMode` enumeration argument, `Add` or `Replace,specifies whether or not it should` replace old modules or programs.

If the file extension is not a valid module (mod or sys) or program (pgf) extension an `ArgumentException` is thrown.

VB:

```
Try
  ATask.LoadProgramFromFile(APrgFileName, RapidLoadMode.Replace)
  ATask.LoadModuleFromFile(AModFileName, RapidLoadMode.Add)
Catch ex As ArgumentException
  Return
End Try
```

C#:

```
try
{
  aTask.LoadProgramFromFile(aPrgFileName, RapidLoadMode.Replace);
  aTask.LoadModuleFromFile(aModFileName, RapidLoadMode.Add);
}
catch (ArgumentException ex)
{
  return;
}
```


Continues on next page

Continued

NOTE!

All program files must reside in the file system of the controller and not locally on the PC. In order to load a program from the PC you must first download it to the controller by using the `FileSystem.PutFile` method. See *Saving files on page 265* for details.

NOTE!

If the User Authorization System of the controller is used by the PC SDK application, it is required that the logged on user has the UAS grant UAS_RAPID_LOADPROGRAM to load and unload RAPID programs. See the *PC SDK Reference* for further information about which grants are necessary for a specific PC SDK method.

Save programs and modules

You can save programs using the `Task.SaveProgramToFile` method and a single module by using the `Module.SaveToFile` method.

To unload a program after it has been saved to file you call `DeleteProgram()`.

VB:

```
Dim TaskCol As Task() = AController.Rapid.GetTasks()
Dim AnObject As Object
For Each AnObject in TaskCol
   ATask = DirectCast(AnObject, Task)
   ATask.ProgramName = ATask.Name
   ATask.SaveProgramToFile(SaveDir)
   ATask.DeleteProgram()
Next
```

C#:

```
Task[] taskCol = aController.Rapid.GetTasks();
foreach (Task aTask in taskCol)
{
   aTask.ProgramName = aTask.Name;
   aTask.SaveProgramToFile(saveDir);
   aTask.DeleteProgram();
}
```

In this example a module is saved to file:

VB:

```
AModule.SaveToFile(AFilePath)
```

C#

```
aModule.SaveToFile(aFilePath);
```

ResetProgramPointer method

Using `ResetProgramPointer` you can set the program pointer to the main entry point of the task.

VB:

```
ATask.ResetProgramPointer()
```

Continues on next page

C#:

```
aTask.ResetProgramPointer();
```

Start program

Starting program execution in the robot controller can only be done in automatic operating mode. There are several overloaded `Start` methods to use, the simplest way to start RAPID execution of a controller task is:

VB:

```
ATask.Start()
```

C#:

```
aTask.Start();
```

NOTE!

If your application uses the User Authorization System of the controller (see *User Authorization System on page 70*) you should also check that the current user has the grant UAS_RAPID_EXECUTE before calling the `Start` method.

Execution change event

It is possible to subscribe to events that occur when a RAPID program starts and stops. It is done like this:

VB:

```
AddHandler AController.Rapid.ExecutionStatusChanged, AddressOf
     UpdateUI
```

C#

```
aController.Rapid.ExecutionStatusChanged += UpdateUI;
```

See *Avoiding threading conflicts on page 259*and *Letting the user know that RAPID data has changed on page 224* for information about how to write the event handler that is needed to update the GUI due to a controller event.

7.5.7. Enable operator response to RAPID UI-instructions from a PC

Remote operator dialog

RAB 5.12 supports operator dialogs to be launched on a PC instead of the FlexPendant when RAPID UI- and TP-instructions are executed. In this chapter this new feature is referred to as *Remote operator dialog*. It enables an operator to give the feedback required by the RAPID program from a PC instead of using the FlexPendant.

NOTE!

Remote operator dialog can only be used with RobotWare 5.12 and later.

Supported RAPID instructions

The following RAPID instructions are supported:

- UIAlphaEntry
- UIListView
- UIMessageBox
- UIMsgBox
- UINumEntry
- UINumTune
- TPErase
- TPReadFK
- TPReadNum
- TPWrite

UIInstructionType

The PC SDK `UIInstructionType` enumeration defines the different RAPID instructions listed above. For a description of each instruction type, see *PC SDK Reference*. Below is an example of such a description.

Example `UIInstructionType.UIAlphaEntry` :

Member	Description
UIAlphaEntry	The UIAlphaEntry (User Interaction Alpha Entry) is used to let an operator communicate with the robot system via RAPID, by enabling him to enter a string from the FlexPendant or from a PC SDK application. After the operator has entered the text, it is transferred back to the RAPID program by calling UIAlphaEntryEventArgs.SendAnswer.

Continues on next page

Continued

TIP!

For complete information about the usage in RAPID refer to *RAPID Technical reference manual* (easily accessed from within RobotStudio for example).

Increased flexibility

Making use of the *Remote operator dialog* feature, the end-user of the robot system can choose whether to use the FlexPendant or the PC SDK application to answer a RAPID UI- or TP-instruction.

The FlexPendant will always show the operator dialog the usual way. If the operator responds from the PC the message on the FlexPendant will disappear.

NOTE!

Likewise, the dialog of the PC SDK application should disappear if the operator chooses to respond from the FlexPendant. This is handled by the PC SDK programmer.

Basic approach

The basic procedure for implementing *Remote operator dialog* in a PC SDK application is shown below. The same approach is used internally by the FlexPendant when it launches its operator view.

Step	Action
1	Set up a subscription to `UIInstructionEvent`.
2	In the event handler check the `UIInstructionEventType` from the event arguments. If `Post` or `Send` create an operator dialog by using the information provided by the event arguments.
3	To transfer the response of the end-user to the RAPID program call the `SendAnswer` method of the specialized `UIInstructionEventArgs` object.
4	Remove any existing operator dialog if you get a `UIInstructionEvent` of `UIInstructionEventType.Abort`.

Continues on next page

Continued

NOTE!

Remember that controller events are always received on a background thread and that you need to enforce execution to the GUI thread by the use of `Invoke` before launching the operator dialog. See *Controller events and threads on page 67* for details.

UIInstructionEvent

To be notified when a UI-instruction event has occurred in the controller, you need to set up a subscription to `UIInstructionEvent`. To do that you use the `UIInstruction` property of the `Rapid` class, like this:

```
Controller c = new Controller();
c.Rapid.UIInstruction.UIInstructionEvent += OnUIInstructionEvent;
```

TIP!

For a code example including an event handler see UIInstructionEvent in the *PC SDK Reference*.

UIInstruction event arguments

To create the dialog in accordance with the arguments of the RAPID instruction and to transfer the response of the operator back to the executing RAPID program, you use the information of the event arguments.

UIInstructionEventArgs

The `UIInstructionEventArgs` object holds information about which RAPID task and which UI- or TP-instruction triggered the event. The picture below shows all `UIInstructionEventArgs` members.

Continues on next page

Continued

7.5.7_1

`UIInstructionEventArgs` is a base class of several specialized classes, one for each UI- and TP- instruction. The specialized class holds the additional information needed to create the operator dialog, so type casting the `UIInstructionEventArgs` object to the correct specialized type is necessary. To do that you first check the `InstructionType` property, which you can see in the picture above.

Continues on next page

7.5.7. Enable operator response to RAPID UI-instructions from a PC

Continued

UIListViewEventArgs

As an example of a specialized type, the members of the `UIListViewEventArgs` class are shown below. The `Buttons` and `ListItems` properties are of course crucial for creating the operator dialog.

7.5.7_2

UIInstructionEventType

An important property in the picture above is `UIInstructionEventType`. It is inherited from the base class and comes with all UI- and TP- instruction events.

The table shows the members of the `UIInstructionEventType` enumeration.

Member	Description
Undefined	Undefined. Should not occur.
Post	Post event type, e.g. TPWrite, TPErase. When the event is of this type RAPID expects no response from the operator.

Continues on next page

Continued

Member	Description
Send	Send event type, e.g. TPReadNum, UIListView. When the event is of this type the running RAPID program expects feedback from the operator before execution continuous.
Abort	When the controller gets a response from a client (the FlexPendant or a PC SDK application) it sends an event of `Abort` type. This tells all subscribing clients that the UI-Instruction has been aborted, closed or confirmed by the operator. When you get an event of this type you should remove any open operator dialog.

NOTE!

If the robot system has several RAPID tasks it is of course necessary to keep track of which operator dialog belongs to which task etc.

A RAPID task can only handle ONE pending `Send`, and it is not guaranteed that an `Abort` event will always follow a `Send` event. Therefore, if you receive a new `Send` event from the same task without a preceding `Abort`event, you should remove the existing dialog and display the new one.

SendAnswer method

To transfer the response of the end-user back to the RAPID program you call the `SendAnswer` method. See the picture of the `UIListViewEventArgs` class above. `SendAnswer` is called with different arguments depending on the RAPID instruction.

For example, if it is a `UIAlphaEntry` instruction you just send the string that the operator has entered as argument. But if it is a `UIListView` instruction the `SendAnswer` method will look like this:

```
public void SendAnswer(int listItemIdx, UIButtonResult btnRes);
```

NOTE!

There is no mastership handling involved in using *Remote operator dialog*.

7.6. IO system domain

Overview

A robot system uses input and output signals to control processes. Signals can be of digital, analog or group signal type. Such IO signals are accessible using the SDK.

Signal changes in the robot system are often significant, and there are many scenarios where end-users of the system need notification of signal changes.

To speed up event notification from the controller there is new functionality in PC SDK 5.10, which allows you to set up subscription priorities. This possibility applies to I/O signals and persistent RAPID data. This mechanism is further described in *Implementing high priority event subscription on page 258*.

Accessing signals

Accessing signals is done through the `Controller` object and its property `IOSystem`, which represents the IO signal space in the robot controller.

To access a signal you need the system name of the signal. The object that is returned from the `IOSystem.GetSignal` method is of type `Signal`.

VB:

```
Dim Signal1 As Signal = AController.IOSystem.GetSignal("signal
     name")
```

C#:

```
Signal signal1 = aController.IOSystem.GetSignal("signal name");
```

The returned `Signal` object has to be typecast to digital, analog or group signal. This example shows a how a signal of type `DigitalSignal` is created:

VB:

```
Dim DISig As DigitalSignal = DirectCast(Signal1, DigitalSignal)
```

C#:

```
DigitalSignal diSig = (DigitalSignal) signal1;
```

This example shows a how an `AnalogSignal` is created:

VB:

```
Dim AISig As AnalogSignal = DirectCast(Signal1, AnalogSignal)
```

C#:

```
AnalogSignal aiSig = (AnalogSignal) signal1;
```

This example shows a how a `GroupSignal` is created:

VB:

```
Dim GISig As GroupSignal = DirectCast(Signal1, GroupSignal)
```

C#:

```
GroupSignal giSig = (GroupSignal) signal1;
```


Continues on next page

NOTE!

Remember to call the `Dispose` method of the signal when it should no longer be used.

Getting signals using SignalFilter

Instead of just getting one signal at a time you can get a signal collection using a signal filter. Some of the `SignalFilter` flags are mutually exclusive, e.g. `SignalFilter.Analog` and `SignalFilter.Digital`. Others are inclusive, e.g. `SignalFilter.Digital` and `SignalFilter.Input`. You can combine the filter flags using the “|” character in C# and the `Or` operator in VB:

VB:

```
Dim ASigFilter As SignalFilter = SignalFilter.Digital Or
     SignalFilter.Input
Dim Signals As SignalCollection  =
     AController.IOSystem.GetSignals(ASigFilter)
```

C#:

```
SignalFilter aSigFilter = SignalFilter.Digital |
     SignalFilter.Input;
SignalCollection signals =
     aController.IOSystem.GetSignals(aSigFilter);
```

This piece of code iterates the signal collection and adds all signals to a `ListView` control. The list has three columns displaying signal name, type and value:

VB:

```
For Each ASignal As Signal In Signals
  Item = New ListViewItem(ASignal.Name)
  Item.SubItems.Add(ASignal.Type.ToString())
  Item.SubItems.Add(ASignal.Value.ToString())
  Me.ListView1.Items.Add(Item)
Next
```

C#:

```
foreach(Signal signal in signals)
{
  item = new ListViewItem(signal.Name);
  item.SubItems.Add(signal.Type.ToString());
  item.SubItems.Add(signal.Value.ToString());
  this.listView1.Items.Add(item);
}
```

If the signal objects are no longer needed they should be disposed of:

VB:

```
For Each ASignal As Signal In Signals
  ASignal.Dispose()
Next
```

Continues on next page

Continued

C#:

```
foreach(Signal signal in signals)
{
  signal.Dispose();
}
```

Reading IO signal values

These examples show how to read a digital and an analog signal.

Digital signal

This piece of code reads the digital signal DO1 and checks a checkbox if the signal value is 1 (ON):

VB:

```
Dim Sig As Signal = AController.IOSystem.GetSignal("DO1")
Dim DigitalSig As DigitalSignal = DirectCast(Sig, DigitalSignal)
Dim val As Integer = DigitalSig.Get
If val = 1 Then
  Me.CheckBox1.Checked = True
EndIf
```

C#:

```
Signal sig = aController.IOSystem.GetSignal("DO1");
DigitalSignal digitalSig = (DigitalSignal)sig;
int val = digitalSig.Get();
if (val == 1)
{
  this.checkBox1.Checked = true;
}
```

Analog signal

This piece of code reads the value of the analog signal AO1 and displays it in a textbox:

VB:

```
Dim Sig As Signal = AController.IOSystem.GetSignal("AO1")
Dim AnalogSig As AnalogSignal = DirectCast(Sig, AnalogSignal)
Dim AnalogSigVal As Single = AnalogSig.Value
Me.TextBox1.Text = AnalogSigVal.ToString()
```

C#:

```
Signal sig = aController.IOSystem.GetSignal("AO1");
AnalogSignal analogSig = (AnalogSignal)sig;
float analogSigVal = analogSig.Value;
this.textBox1.Text = analogSigVal.ToString();
```


Continues on next page

Writing IO signal values

This section shows how the value of a digital or an analog IO signal can be modified by a RAB application.

NOTE!

In manual mode a signal value can be modified only if the *Access Level* of the signal is *ALL*. If not, the controller has to be in auto mode.

Digital signal

This piece of code changes the value of a digital signal in the controller when the user checks/unchecks a checkbox:

VB:

```
Private Sub CheckBox1_Click(ByVal sender As Object, ByVal e As
     System.EventArgs) Handles CheckBox1.Click
  If Me.CheckBox1.Checked Then
     DigitalSig.Set()
  Else
     DigitalSig.Reset()
  End If
End Sub
```

C#:

```
private void checkBox1_Click(object sender, EventArgs e)
{
  if (this.checkBox1.Checked)
  {
     digitalSig.Set();
  }
  else
  {
     digitalSig.Reset();
  }
}
```

NOTE! You can also set the value using the `Value` property.

Analog signal

This piece of code writes the value entered in a text box to the analog signal AO1. The value is converted from string to a float before it is written to the controller:

VB:

```
Dim AnalogSigVal As Single = Convert.ToSingle(Me.TextBox1.Text)
AnalogSig.Value = AnalogSigVal
```

Continues on next page

Continued

C#:

```
float analogSigVal = Convert.ToSingle(this.textBox1.Text);
analogSig.Value = analogSigVal;
```

Listening to signal changes

Once a `Signal` object is available it is possible to add a subscription to its Changed event, which is triggered at a signal change such as changed value, changed simulated status or changed signal quality.

Visual Basic

```
Friend WithEvents Sig As AnalogSignal
...
AddHandler Sig.Changed, AddressOf AISig_Changed
...
Private Sub Sig_Changed(sender As Object, e As
     SignalChangedEventArgs) Handles Sig.Changed
End Sub
```

C#

```
this.sig.Changed += sig_Changed;
...
private void sig_Changed(object sender, SignalChangedEventArgs e)
{ }
```

Start and stop subscriptions

It is recommended that you activate and deactivate subscriptions to the `Changed` event if these are not necessary throughout the lifetime of the application:

VB:

```
AddHandler Sig.Changed, AddressOf Sig_Changed
RemoveHandler Sig.Changed, AddressOf Sig_Changed
```

C#:

```
this.sig.Changed += sig_Changed;
this.sig.Changed -= sig_Changed;
```

Implementing high priority event subscription

To speed up event notification from the controller it is possible to set up subscription priorities for I/O signals. To do this you use the `Subscribe` method and the enumeration `EventPriority` as argument. The example shows an ordinary signal subscription and a subscription with high priority:

VB:

```
AddHandler Sig.Changed, AddressOf Sig_Changed
Sig.Subscribe(AddressOf Sig_Changed, EventPriority.High)
```


Continues on next page

C#:

```
signal.Changed += sig_Changed;
signal.Subscribe(sig_Changed, EventPriority.High);
```

To deactivate subscriptions with high priority you call the `Unsubscribe` method like this:

VB:

```
sig.Unsubscribe(AddressOf sig_Changed)
```

C#:

```
signal.Unsubscribe(sig_Changed);
```

Limitations for high priority events

High priority subscriptions can be used for I/O signals and RAPID data declared PERS. The controller can handle 64 high priority subscriptions.

Avoiding threading conflicts

It is important to keep in mind that all controller events use their own threads, which are different from the application GUI thread. This can cause problems if you want to display signal changes in the application GUI. See *Controller events and threads on page 67* for further information.

If an update of the user interface is not necessary, you do not need to take any special action, but can execute the event handler on the event thread. If, however, you need to show to the user that the signal has changed you should use the `Invoke` method. It forces execution to the window control thread and thus provides a solution to potential threading conflicts.

VB:

```
Me.Invoke(New EventHandler(AddressOf UpdateUI), New Object()
     {sender, e})
```

C#:

```
this.Invoke(new EventHandler<SignalChangedEventArgs>(UpdateUI),
     new Object[] { sender, e });
```

Reading the new value

The `SignalChangedEventArgs` object has a `NewSignalState` property, which has information about signal value, signal quality and whether the signal is simulated or not:

VB:

```
Private Sub UpdateUI(ByVal Sender As Object, ByVal e As
     SignalChangedEventArgs)
  Dim State As SignalState = e.NewSignalState
  Dim val As Single
  Val = State.Value
  Me.TextBox1.Text = Val.ToString()
  ....
End Sub
```


Continues on next page

Continued

C#:

```
private void UpdateUI(object sender, SignalChangeEventArgs e)
{
  SignalState state = e.NewSignalState;
  ....
  float val = state.Value
  this.textBox1.Text = val.ToString()
}
```

NOTE!

There is no guarantee you will receive an initial event when setting up the subscription. To get initial information about the value of a signal you should read it using the `Value` property.

NOTE!

Make sure the subscription is removed before you dispose of the signal. See *Memory management in PC applications on page 213* for further information.

7.7. Event log domain

Overview

Event log messages may contain information about controller status, RAPID execution, the running processes of the controller etc.

Using the SDK it is possible to either read messages in the queue or to use an event handler that will receive a copy of each new log message. An event log message contains queue type, event type, event time, event title and message.

Accessing the controller event log

You access the event log domain through the `Controller` property `EventLog`.

VB:

```
Private Log As EventLog = AController.EventLog
```

C#:

```
private EventLog log = aController.EventLog;
```

Accessing event log categories

All event log messages are organized into categories. To search for an individual message you have to know what category it belongs to. The enumeration type, `CategoryType`, defines all available categories. You can get a category either by using the method `GetCategory` or by using the `Categories` property, which is an array of all available categories.

VB:

```
Dim Cat As EventLogCategory
Cat = Log.GetCategory(CategoryType.Program)
```

or

```
Cat = Log.Categories(4)
```

C#:

```
EventLogCategory cat;
cat = log.GetCategory(CategoryType.Program);
```

or

```
cat = log.GetCategory[4];
```

NOTE!

The `EventLogCategory` should be disposed of when it is no longer used.

Accessing event log messages

To access a message you use the `Messages` property of the `Category` object. A collection of messages is returned. The collection implements the `ICollection` and `IEnumerable` interfaces , which means you can use the common operations for collections. Access is done either using an index or by iterating using `foreach`.

VB:

```
Dim Msg As EventLogMessage = Cat.Messages(1)
```

Continues on next page

Continued

or

```
Dim Msg As EventLogMessage
For Each Msg In Cat.Messages
  Me.TextBox1.Text = Msg.Title
  .....
Next Item
```

C#:

```
EventLogMessage msg = cat.Messages[1];
```

or

```
foreach(EventLogMessage msg in cat.Messages)
{
  this.textBox1.Text = msg.Title;
  .....
}
```

MessageWritten event

It is possible to add an event handler that is notified when a new messages is written to the controller event log. This is done by subscribing to the EventLog event `MessageWritten`.

The event argument is of type `MessageWrittenEventArgs` and has a `Message` property, which holds the latest event log message.

VB:

```
Private Sub Log_MessageWritten(sender As Object, e As
     MessageWrittenEventArgs) Handles Log.MessageWritten
  Dim Msg As EventLogMessage = e.Message
End Sub
```

C#:

```
private void log_MessageWritten(object sender,
     MessageWrittenEventArgs e)
{
  EventLogMessage msg = e.Message;
}
```

NOTE!

If the application user interface needs to be updated as a result of the event, you must delegate this job to the GUI thread using the `Invoke` method. See *Invoke method on page 68* for further information and code samples.

TIP!

Find out more about the `EventLogDomain` in the PC SDK Reference help.

7.8. Motion domain

Overview

The `MotionDomain` namespace lets you access the mechanical units of the robot system.

Motion system

You access the motion system by using the `Controller` property `MotionSystem`.

VB:

```
Private AMotionSystem As MotionSystem
AMotionSystem = AController.MotionSystem
```

C#

```
private MotionSystem aMotionSystem;
aMotionSystem = aController.MotionSystem;
```

By using the `MotionSystem` object you can, for example, use its `SpeedRatio` property to find out about the current speed of the robot.

Accessing Mechanical units

The mechanical units can be of different types, e.g. a robot with a TCP, a multiple axes manipulator or a single axis unit. All these are available through the MotionSystem property `MechanicalUnits`. If only the currently active mechanical unit is of interest you had better use the `ActiveMechanicalUnit` property.

VB:

```
Dim AMechCol As MechanicalUnitCollection =
     AController.MotionSystem.MechanicalUnits
Dim AMechUnit As MechanicalUnit =
     AController.MotionSystem.ActiveMechanicalUnit;
```

C#:

```
MechanicalUnitCollection aMechCol =
     aController.MotionSystem.MechanicalUnits;
MechanicalUnit aMechUnit =
     aController.MotionSystem.ActiveMechanicalUnit;
```

Mechanical unit properties and methods

There are numerous properties available for the mechanical unit, e.g. `Name`,`Model`, `NumberOfAxes`,`SerialNumber`,`CoordinateSystem`, `MotionMode`, `IsCalibrated`, `Tool` and `WorkObject` etc. It is also possible to get the current position of a mechanical unit as a `RobTarget` or `JointTarget`.

VB:

```
Dim ARobTarget As RobTarget =
     AController.MotionSystem.GetActiveMechanicalUnit.GetPositio
     n(CoordinateSystemType.World)
Dim AJointTarget As JointTarget =
     AController.MotionSystem.ActiveMechanicalUnit.GetPosition()
```

Continues on next page

Continued

C#:

```
RobTarget aRobTarget =
     aController.MotionSystem.GetActiveMechanicalUnit.GetPositio
     n(CoordinateSystemType.World);
JointTarget aJointTarget =
     aController.MotionSystem.ActiveMechanicalUnit.GetPosition()
     ;
```

TIP!

Find out more about the `MotionDomain` in the PC SDK Reference help.

7.9. File system domain

Overview

Using the SDK it is possible to create, save, load, rename and delete files in the controller file system. It is also possible to create and delete directories.

Accessing files and directories

You access the file system domain through the `Controller` property `FileSystem`.

VB:

```
Private AFileSystem As FileSystem = AController.FileSystem
```

C#:

```
FileSystem aFileSystem = aController.FileSystem;
```

Controller and PC directory

You can get and set the directory on the controller and on the local PC system using the `RemoteDirectory` and `LocalDirectory` properties.

VB:

```
Dim RemoteDir As String = AController.FileSystem.RemoteDirectory
Dim LocalDir As String = AController.FileSystem.LocalDirectory
```

C#:

```
string remoteDir = aController.FileSystem.RemoteDirectory;
string localDir = aController.FileSystem.LocalDirectory;
```

Environment variables

When specifying file system paths you can use environment variables to denote the HOME, system, backup and temp directories of the currently used system. When an application uses “(BACKUP)$” it is internally interpreted as the path to the backup directory of the current system. The other environment variables are: HOME, TEMP and SYSTEM.

Loading files

You can load a file from the controller to the PC using the `GetFile` method. The method generates an exception if the operation did not work. The arguments are complete paths including filenames.

VB:

```
AController.FileSystem.FileSystem.GetFile(RemoteFilePath,
     LocalFilePath)
```

C#:

```
aController.FileSystem.GetFile(remoteFilePath, localFilePath);
```

Saving files

You can save a file on the controller file system by using the `PutFile` method. The method generates an exception if the operation did not work. The arguments are complete paths including filenames.

VB:

```
AController.FileSystem.PutFile(LocalFilePath, RemoteFilePath)
```

Continues on next page

Continued

C#:

```
aController.FileSystem.PutFile(localFilePath, remoteFilePath);
```

CopyFile and CopyDirectory

The `PutFile` / `GetFile` methods generate a copy of a file and transfer it to or from the controller file system. Using the `CopyFile` and `CopyDirectory` you can create a copy directly on the controller:

VB:

```
AController.FileSystem.CopyFile(FromFilePath, ToFilePath)
AController.FileSystem.CopyDirectory(FromDirPath, ToDirPath)
```

C#:

```
aController.FileSystem.CopyFile(fromFilePath, toFilePath);
aController.FileSystem.CopyDirectory(fromDirPath, toDirPath);
```

Getting multiple files and directories

The `FileSystem` class has a method called `GetFilesAndDirectories`. It can be used to retrieve an array of `ControllerFileSystemInfo` objects with information about individual files and directories. The `ControllerFileSystemInfo` object can then be cast to either a `ControllerFileInfo` object or a `ControllerDirectoryInfo` object.

This example uses search pattern to limit the search.

VB:

```
Dim AnArray As ControllerFileSystemInfo()
Dim info As ControllerFileSystemInfo
AnArray = AController.FileSystem.GetFilesAndDirectories("search
     pattern")
Dim I As Integer
For I = 0 To array.Length -1
  info = AnArray(I)
  ......
Next
```

C#:

```
ControllerFileSystemInfo[] anArray;
ControllerFileSystemInfo info;
anArray = aController.FileSystem.GetFilesAndDirectories("search
     pattern");
for (int i=0;i<anArray.Length;i++) {
  info = anArray[i];
  ......
}
```


Continues on next page

Using search patterns

As seen in the example above, you can use search patterns to locate files and directories using the `GetFilesAndDirectories` method. The matching process follows the Wildcard pattern matching in Visual Studio. This is a brief summary:

Character in pattern	Matches in string
?	Any single character
*	Zero or more characters
#	Any single digit (0–9)
[*charlist*]	Any single character in *charlist*
[!*charlist*]	Any single character not in *charlist*

TIP!

Find out more about the `FileSystemDomain` in the PC SDK Reference help.

7.10. Messaging domain

Overview

The `Messaging` domain of the PC SDK can be used to send and receive data between a PC SDK application and a RAPID task.

The corresponding RAPID functionality, *RAPID Message Queue*, includes RAPID data types and RAPID instructions and functions for sending and receiving data. It enables communication between RAPID tasks or between a RAPID task and a RAB application.

This chapter provides information about how to implement messaging in a PC SDK application. To make it work it is necessary to do part of the implementation in RAPID. In order to show how this can be done a simple but complete code example in C# and RAPID is provided at the end of the chapter.

NOTE!

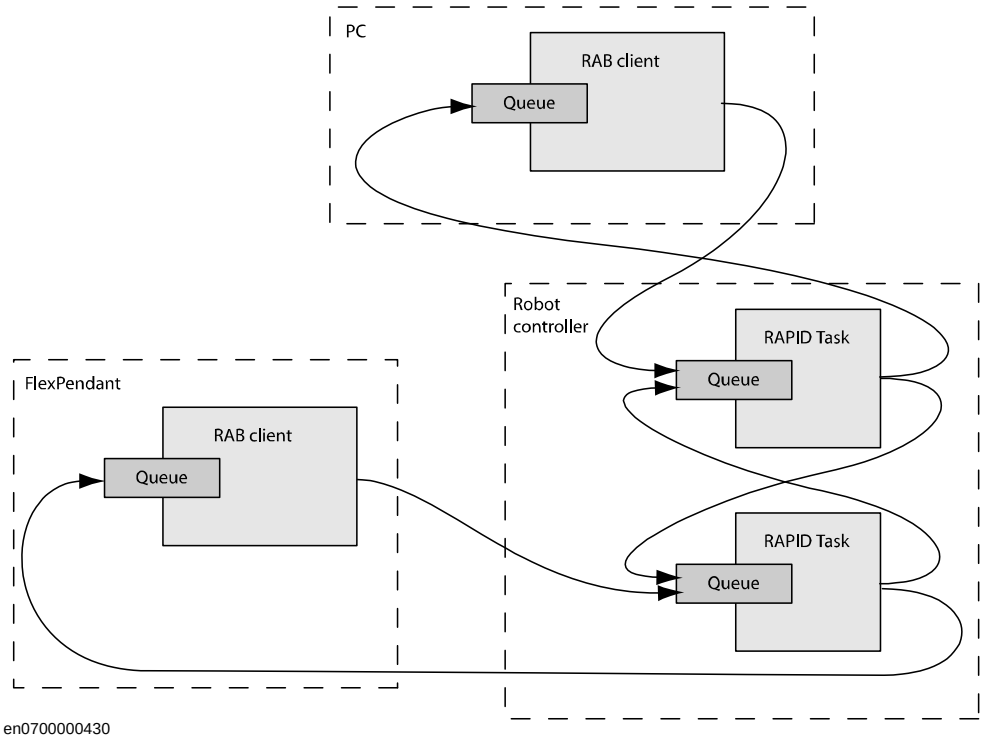
See *Application manual - Robot communication and I/O Control* for detailed information about how to implement messaging in RAPID.

RobotWare option

The functionality in RAPID that is needed to utilize messaging - *RAPID Message Queue* - is included in the RobotWare options *PC Interface*, *FlexPendant Interface* and *Multitasking*. As *PC Interface* is required on a robot controller to be used with a PC SDK client, this means no extra option is needed to start using *RAPID Message Queue* with a PC SDK application.

Messaging illustration

The illustration shows possible senders and receivers in the robot system. The arrows represent ways to communicate by posting a message to a queue.



en0700000430

Continues on next page

Continued

NOTE!

In principle, messages might as well be sent between a PC SDK client and a RAB application running on the FlexPendant.

NOTE!

The messaging functionality of the FlexPendant SDK has not yet been made public, but in RAB 5.11 a mechanism has been implemented, which allows advanced users access to it. You should contact support if you need information about how to use it.

Benefits

Together with *RAPID Message Queue* the functionality of the `Messaging` domain represent a new, flexible way for a RAB application to interact with a RAPID task.

Messaging is usually done when a RAPID task is executing, but it is also possible to send a message to a RAPID task when it has been stopped. The RAPID interrupt will then occur once the RAPID task has been started.

An simple example of usage would be to set a flag from a RAB application in order to control the program flow in the RAPID program.

NOTE!

Sending messages can be done in both manual and auto mode. As opposed to using `RapidData` to modify a RAPID variable no mastership is required.

The Messaging namespace

The Microsoft Windows operating system provides mechanisms for facilitating communications and data sharing between applications. Collectively, activities enabled by these mechanisms are called *Interprocess communications* (IPC).

These are the classes and enumerations available in the `Messaging` namespace:

Continues on next page

7.10. Messaging domain

Continued

7.9_1

Continues on next page

The `Ipc` class is used to handle message queues with methods like `GetQueue`, `CreateQueue`, `DeleteQueue` etc. When you have an `IpcQueue` object you can use its `Send` method to send an `IpcMessage` to a RAPID task or its `Receive` method to receive a message from a RAPID task.

When sending a message you use an existing queue in the controller as the `IpcQueue` object. The naming principle of queues in the controller is using the name of the corresponding task prefixed with “RMQ_”, e.g “RMQ_T_ROB1”. To be able to receive a message from RAPID you must first create your own message queue and use that object with the `Receive` method.

NOTE!

When the execution context in a RAPID task is lost, e.g. when the program pointer is moved to main, the corresponding queue is emptied.

Basic approach

To utilize messaging in a PC SDK application you need to do the implementation both in RAPID and in the PC application.

This is the general approach for sending data from a PC application and receiving it in a RAPID task:

1. In the PC application connect to the queue of the RAPID task.
2. Create the message.
3. Send the message.
4. In the RAPID program set up a trap routine that reads the message. Connect an interrupt so that the trap routine is called each time a new message appears.

For a complete code example using this scenario see *Code example on page 274*.

Continues on next page

Continued

What can be sent in a message?

In RAPID there is a `rmqmessage data type`. In the PC SDK the corresponding type is `IpcMessage`. An `IpcMessage` object stores the actual data in the message, but also information about message size, who the sender is etc.

The data in a message is a pretty-printed string with data type name (and array dimensions) followed by the actual data value. The data type can be any RAPID data type. Arrays and user defined records are allowed.

Message data - examples:

“robtarget;[[930,0,1455],[1,0,0,0],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]]”

“string;“Hello world!””

“num;23”

“bool;FALSE”

“bool{2, 2};[[TRUE,TRUE],[FALSE,FALSE]]”

“msgrec;[100,200]” (user defined data type)

The method `IpcMessage.SetData` is used to fill the `IpcMessage` with the appropriate data. Likewise, the `GetData` method retrieves the data from an `IpcMessage` object.

NOTE!

The `IpcMessage.Data` is set and retrieved as a byte array, `SetData(byte[] data)` and `byte[] GetData()`. This means you must convert the message data string to a byte array before calling the `SetData` method. It may look like this in C#:

```
Byte[] data = new UTF8Encoding().GetBytes("string;\"Hello world\"");
```

See *Code example on page 274* for a few other examples.

NOTE!

The RAPID program can specify what RAPID data type it expects to receive by connecting it to a TRAP routine. A message containing data of a data type that no interrupt is connected to will be discarded with only an event log warning.

RAPID Message Queue system parameters

This is a brief description of each system parameter of *RAPID Message Queue*. For further information, see the respective parameter in *Technical reference manual - System parameters*.

These parameters belong to the *Task* type in the *Controller* topic:.

Parameter	Description
RmqType	The following values are possible: • None - Disables the RAPID Message Queue functionality in this RAPID task. This is the default value. • Internal - Enables the RAPID Message Queue for local usage on the controller. • Remote - Enables the RAPID Message Queue for local usage and for PC and FlexPendant applications.

Continues on next page

Continued

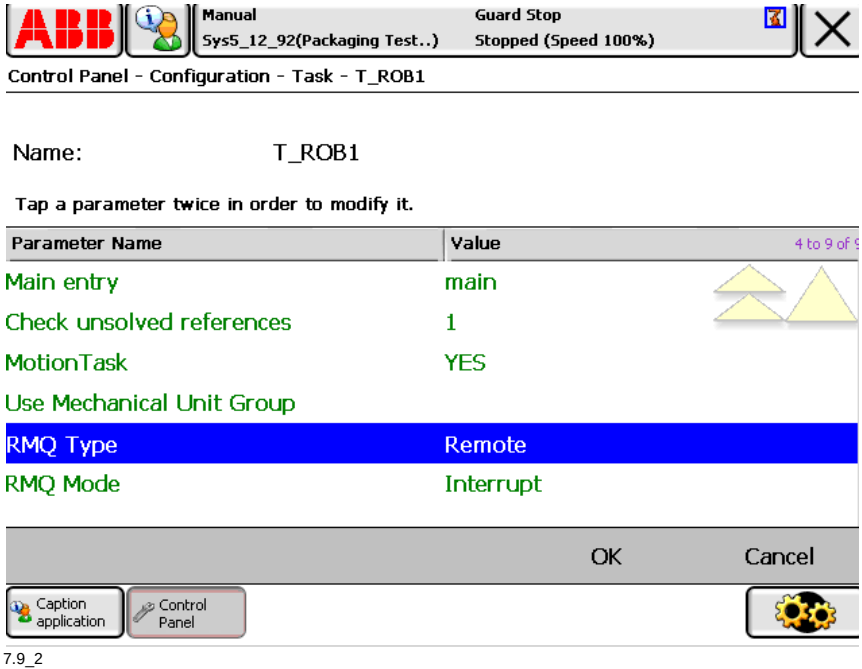
Parameter	Description
RmqMode	• Interrupt mode - A message can be received either by connecting a trap routine to a specified message type or by using the send-wait functionality. Any messages that are not the answer to an active send-wait instruction or have the type connected to a trap routine will be discarded. This is the default mode. • Synchronous mode - All messages will be queued and can only be received through the new read-wait instruction `RMQReadWait`. No messages will be discarded unless the queue is full. The send-wait instruction is not available in this mode. New mode from 5.12.
RmqMaxMsgSize	The maximum data size, in bytes, for a message. The default value is 350. The value cannot be changed in RobotStudio or on the FlexPendant.
RmqMaxNoOfMsg	Maximum number of messages in queue. The default value is 5. The value cannot be changed in RobotStudio or on the FlexPendant.

NOTE!

To read the values of these system parameter from the PC SDK you use the `IpcQueue` properties `RemoteAccessible, MessageSizeLimit` and `Capacity`.

Remote RmqType

The system parameter *RmqType* must be set to **Remote** to enable messaging between RAPID and RAB:



7.9_2

Continues on next page

Continued

Code example

This simple messaging example can be tested with a virtual or a real controller. The system parameter *RmqType* must be set to **Remote** as shown in *RAPID Message Queue system parameters on page 272*.

The following code sample creates a message and sends it to a RAPID task, which reads it and sets a RAPID variable accordingly. Then an “Acknowledged” message is sent back to the PC SDK queue. Finally, the PC SDK application launches the received message in a Message Box.

PC SDK - C#

A message is created and sent to the RAPID queue “RMQ_T_ROB1”. An answer message is then received from RAPID and launched in a Message Box.

C#:

```
//declarations
private Controller c;
private IpcQueue tRob1Queue;
private IpcQueue myQueue;
private IpcMessage sendMessage;
private IpcMessage recMessage;
...

//initiation code, eg in constructor
c = new Controller(); //default ctrl used here (App.config)

//get T_ROB1 queue to be able to send msgs to RAPID task
tRob1Queue = c.Ipc.GetQueue("RMQ_T_ROB1");

//create my own RAB queue to be able to receive msgs
if (!c.Ipc.Exists("RAB_Q"))
{
  myQueue = c.Ipc.CreateQueue("RAB_Q", 5, Ipc.IPC_MAXMSGSIZE);
  myQueue = c.Ipc.GetQueue("RAB_Q");
```


Continues on next page

Continued

```
}

//Create IpcMessage objects for sending and receiving
sendMessage = new IpcMessage();
recMessage = new IpcMessage();
...

//in an event handler, eg. button_Click
SendMessage(true);
CheckReturnMsg();
...
public void SendMessage(bool boolMsg)
{
  Byte[] data = null;
  //Create message data
  if (boolMsg)
  {
     data = new UTF8Encoding().GetBytes("bool;TRUE");
  }
  else
  {
     data = new UTF8Encoding().GetBytes("bool;FALSE");
  }

  //Place data and sender information in message
  sendMessage.SetData(data);
  sendMessage.Sender = myQueue.QueueId;

  //Send message to the RAPID queue
  tRob1Queue.Send(sendMessage);
}
```


Continues on next page

Continued

```
private void CheckReturnMsg()
{
  IpcReturnType ret = IpcReturnType.Timeout;
  string answer = string.Empty;
  int timeout = 5000;

  //Check for msg in the RAB queue
  ret = myQueue.Receive(timeout, recMessage);

  if (ret == IpcReturnType.OK)
  {
     //convert msg data to string
     answer = new UTF8Encoding().GetString(recMessage.Data);
     MessageBox.Show(answer);

     //MessageBox should show: string;"Acknowledged"
  }
  else
  {
     MessageBox.Show("Timeout!");
  }
}
```

RAPID

A trap is created for a message of data type `bool`. In the trap the value of the message data is assigned to the *flag* variable. Then an *"Acknowledged"* message is sent back to the PC SDK client. In *main* the WHILE loop is executed until a message with a TRUE value is received.

```
MODULE RAB_COMMUNICATION
   VAR bool flag := FALSE;
   VAR intnum connectnum;

   PROC main()
     CONNECT connectnum WITH RABMsgs;
     IRMQMessage flag, connectnum;
     WHILE flag = FALSE DO
        !do something, eg. normal processing...
        WaitTime 3;
     ENDWHILE
     !PC SDK message received - do something...
     TPWrite "Message from PC SDK, will now...";
     IDelete connectnum;
     EXIT;
```


Continues on next page

Continued

```
    ENDPROC

    TRAP RABMsgs
      VAR rmqmessage msg;
      VAR rmqheader header;
      VAR rmqslot rabclient;
      VAR num userdef;
      VAR string ack := "Acknowledged";

      RMQGetMessage msg;
      RMQGetMsgHeader msg \Header:=
           header\SenderId:=rabclient\UserDef:=userdef;

      !check data type and assign value to flag variable
      IF header.datatype = "bool" THEN
        RMQGetMsgData msg, flag;
        !return receipt to sender
        RMQSendMessage rabclient, ack;
      ELSE
        TPWrite "Unknown data received in RABMsgs...";
      ENDIF

    ENDTRAP
ENDMODULE
```

NOTE!

Error handling should be implemented in C#, as well as in RAPID.

NOTE!

From RW 5.12 there is a new RAPID instruction, `RMQEmptyQueue` that can be used to empty the queue in a task.

7.10. Messaging domain

8 Debugging and troubleshooting

8.1 FlexPendant - Debugging and troubleshooting

8.1.1. Debug output

Overview

It is possible to get debug output from the FlexPendant by using the network. This is useful for several reasons. It will reveal any exceptions thrown during execution, for example, providing you with error messages and call stacks. Moreover, it can help you check memory consumption and memory leaks in your application.

The FP message is packed into a network message and sent out on the network as a broadcast message on port 9998. Such a message can be picked up in different ways:

- Connect a hub on the local FlexPendant network and connect a PC to it. Use *nc.exe* to pick up the messages (nc - lup 9998).
- The messages sent are also stored in a ring-buffer of size 100kB. To read the buffer to file and store it on the controller, you can use the FlexPendant Command Server. Use *fpcmd "-d"*.
- It is also possible to start a task on the controller that listens to port 9998 and displays all messages in the console buffer. Use command *fp_enable_console_output.*

TIP!

For a list of exceptions that the IRC5 Controller may throw, see *Exception error codes on page 292*.

Enable debug output

To enable debug output write `fpcmd_enable_console_output` in the controller console window.

These are the console commands used to enable and disable printouts:

Console command	Result
fpcmd_enable_console_output 1	Starts producing printouts from RobotWare to the robot controller console.
fpcmd_enable_console_output 2	Starts producing printouts from SDK application.
fpcmd_enable_console_output 3	Combines the two above: RW + SDK printouts.
fpcmd_disable_console_output	Stops printout to the robot controller console.

The command below can be used to retrieve detailed status of the robot controller, which may be useful, although it may not be specifically related to your application.

Console command	Result
fpcmd "-d"	Produces a log file with extensive information on system status to the robot controller file system(hd0a/temp). Use an ftp client or the *File Manager* in RobotStudio to transfer the file to your PC.

Continues on next page

Continued

FlexPendant Command Server

By using the command line on the controller you can send commands to the FlexPendant. The FlexPendant has a command server that interprets the commands and performs the requested operation. The syntax for command line commands is *fpcmd "<command>"*. The only command you need to remember is `fpcmd "-h"`, which is the *Help* command. It produces a printout of available FlexPendant commands in the controller console:

-> fpcmd “-h”value = 8 = 0x8-> [fp]: FlexPendantCmd: Help fpcmd “-h”: Help fpcmd “-a”: Adapter show routine fpcmd “-m”: Measure time in adapters fpcmd “-i”: Display FlexPendant Information fpcmd “-f”: Bring GTPU Services to front fpcmd “-x”: Hide startup progress bar fpcmd “-s”: Start application fpcmd “-d”: Copy Debug file to controller fpcmd “-as”: Print screen fpcmd “-rd”: RobAPI debug fpcmd “-restart”:

Continues on next page

Continued

Restart Device fpcmd “-memShow”: Available memory fpcmd “-module”: Module information of a process fpcmd “-filePut”: Upload a file to the FlexPendant fpcmd “-filleted”: Download a file to the FlexPendant fpcmd “-diarist”: List a directory

NOTE!

All commands support -? (e.g. *fpcmd “-memShow -?”*), which gives further information about the specific command.

TIP!

It is possible to monitor memory consumption in the robot controller console window:

1. Write `fpcmd_enable_console_output 3`.
2. Write `fpcmd "-memShow"`.

See *Discover memory leaks on page 186* for further information.

Trace and Debug

The `ABB.Robotics.Diagnostics` namespace provides trace and debug services. Its `Trace` and `Debug` classes are specifically designed for the FlexPendant environment.

The properties and methods in the `Trace` class are used to instrument release builds, which allows you to monitor the health of your application running in a real-life setting. Tracing can help you to isolate problems and fix them without disturbing a running system.

If you use methods in the `Debug` class to print debugging information and check your logic with assertions, you can make your code more robust without impacting the performance and code size of your shipping product. In Visual Studio, creating a debug build enables `Debug`.

`Trace` and `Debug` give printout information during execution. Messages are displayed on the FlexPendant screen or in the robot controller console. The functionality is similar to that provided by the .Net`Trace` and `Debug` classes.

The `Assert` method checks for a condition and displays an assert message on the FlexPendant, including detailed information and a stack trace, if the condition is false. The message is also displayed in the controller console window if you enter the command `fpcmd_enable_console_output` first.

NOTE!

Add the `ABB.Robotics.Diagnostics` namespace to the `using` section at the top of your source code file.

8.1.2. Debugging the virtual FlexPendant

Overview

When debugging your application it is often convenient to use the virtual environment, but it is almost as easy to attach the Visual Studio debugger to the real FlexPendant device. For information about how that is done see *Debugging the FlexPendant device on page 286*.

This section describes how to start the VS debugger, attach a running Virtual FlexPendant to it, set up break points and step through the source code of a FlexPendant application.

Continues on next page

Debugging procedure

There are several ways of attaching a Visual Studio debugger to a running Virtual FlexPendant application. In this section one method is described in detail.

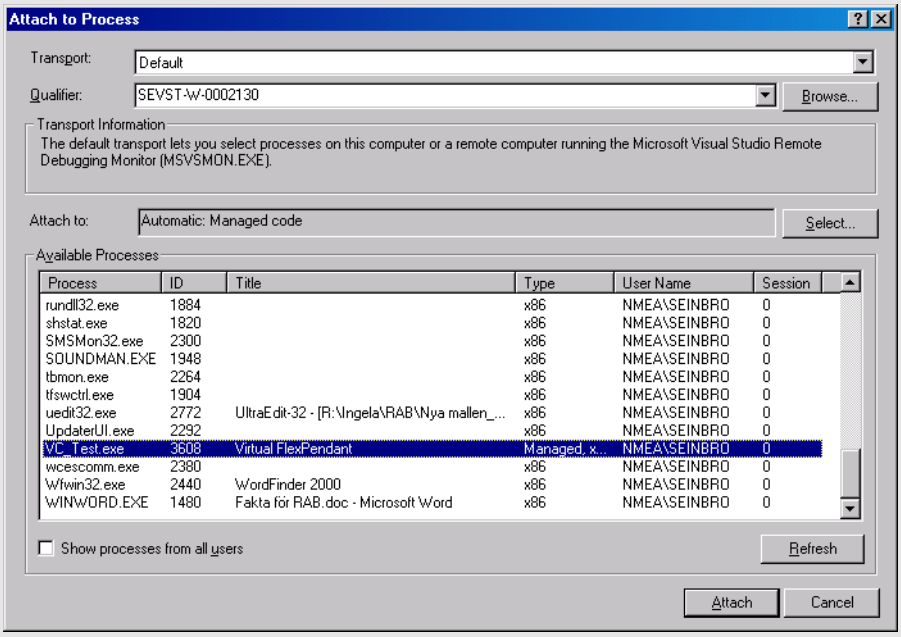
There is no way to start your FlexPendant application from inside the Visual Studio environment. You must start by deploying the application to the Virtual FlexPendant in RobotStudio. How to do this is described in *Hands on - Hello world on page 86*. Then you start the Virtual FlexPendant and attach the Visual Studio debugger to it.

NOTE!

In order to use break points the project build configuration must be *Debug*. You set it in the the **Build** tab of the Project **Properties**. The output directory, where you find the assembly (*.dll), the proxy assembly (*gtpu.dll) and the program database (*.pdb) is the bin/Debug directory, a sub-directory of your VS project.

Attach to Process

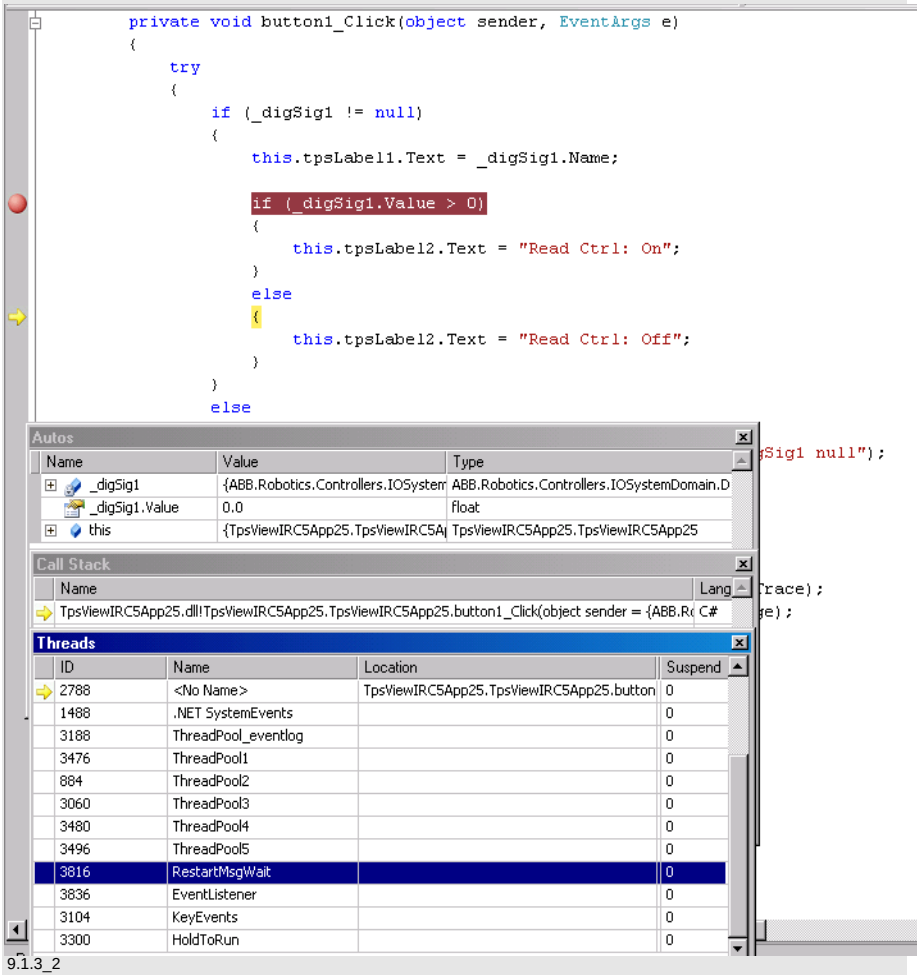
When you have a running application on the Virtual FlexPendant follow these steps:

Step	Action
1	In Visual Studio on the **Debug** menu, select **Attach to Process**. It brings up this dialog:  9.1.3_1
2	Select the *Virtual FlexPendant.exe* process and press the **Attach** button.
3	Set a break point in your source code.
4	On the Virtual FlexPendant, press a button of your application or something else that will make program execution hit the breakpoint.

Continues on next page

8.1.2. Debugging the virtual FlexPendant

Continued

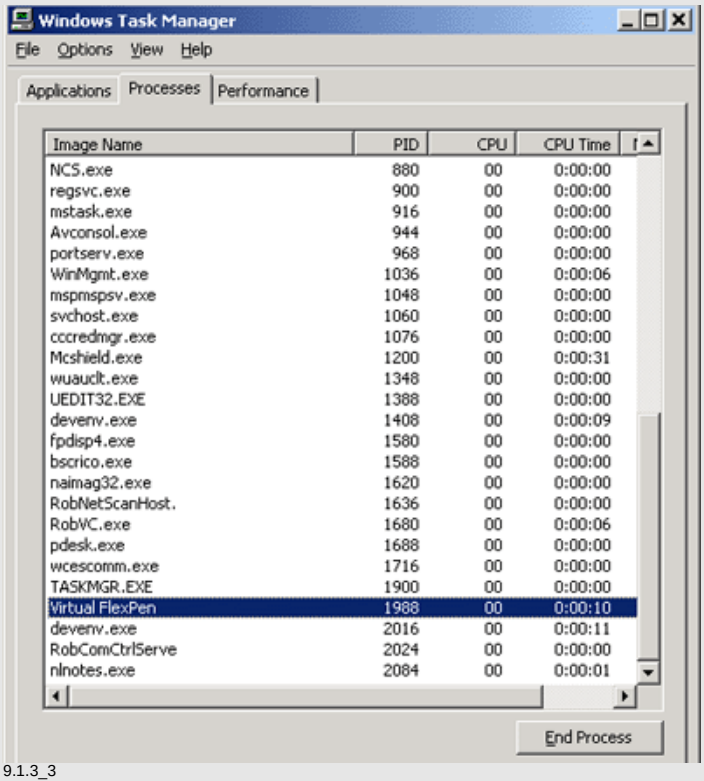
Step	Action
5	On the **Debug** menu, point at Windows and select the debugging information to be displayed while debugging. See example below: 
6	On the **Debug** menu, select the appropriate **Step** command when stepping through your code.
7	On the **Debug** menu, click **Detach All** or **Stop Debugging** when you want to stop debugging.

Continues on next page

Continued

Windows Task Manager

You can also attach a running application through the Windows Task Manager.

Step	
1	Start *Windows Task Manager* and select the **Processes** tab.  9.1.3_3
2	Select the *Virtual FlexPendant.exe* process and right-click to get the context menu. In that menu select **Debug**. You will get a warning message, but select **Yes**.
3	A *Just -In -Time Debugging* dialog will appear. Select your application project as the debugger to use.

Launching debugger programatically

Yet another way of attaching a debugger is to launch it programatically, by writing code in your application.

Step	Action
	Insert the following line where you want the debugger to start: `System.Diagnostics.Debugger.Launch()`
	Start the application in the Virtual FlexPendant and perform the action which is to launch the debugger. A *Just- In-Time Debugging* dialog will appear.
	Select your VS project as the debugger. Click **OK** and start the debug session.

8.1.3. Debugging the FlexPendant device

Overview

This section provides information on how to do debug an application executing on the real FlexPendant device.

In general, using the Visual Studio debugger to debug the device works very well and is strongly recommended, but depending on the OS, Visual Studio version and FlexPendant version you are using the requirements for setting up and attaching the Visual Studio debugger will differ somewhat.

Further information and updates concerning this topic will from this release be published on the RAB User Forum (and not in the User's Guide). See *RAB User Forum on page 288*.

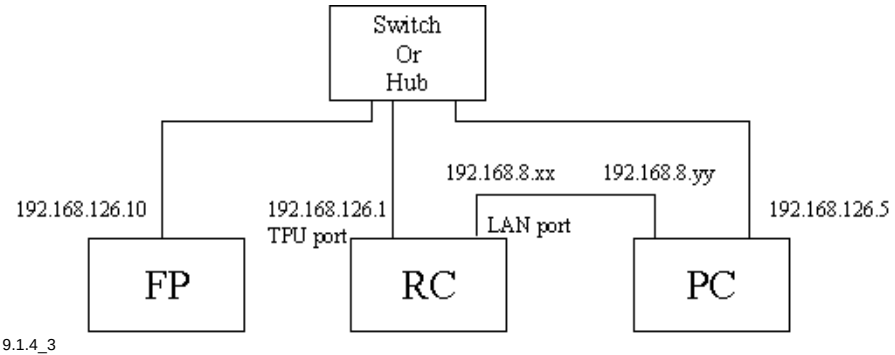
Prerequisites

The following requirements have to be met:

- VS 2005 with SP1 is required to debug the FlexPendant without any adaptations.
- Service Pack 1 or 2 for .NET Compact Framework 2.0 (.NET CF 2.0 SP2) is required for setting up and using the Visual Studio debugger on the FlexPendant device. It can be downloaded from http://www.microsoft.com/downloads.
- To debug with VS 2008 (as well as VS 2005 without SP1) you must follow a procedure that will be presented on the RAB User Forum. The procedure will be different for each RobotWare release.
- If your PC is running under Windows Vista “Windows Mobile Device Center” needs to be installed in order to connect to the device.

Setting up the network

This illustration shows how to connect the FlexPendant, the Robot Controller and your PC in order to debug your application using the Visual Studio debugger.



9.1.4_3

The FlexPendant has a static IP address 192.168.126.10. Your PC IP address must be one on the 126 subnet, i.e. 192.168.126.x (not 1 or 255).

Note that this setup is completely independent of the LAN and Service ports. You are plugging the TPU cable from the RC into the switch, a cable from the switch to the RC, and a cable from your PC to the switch.

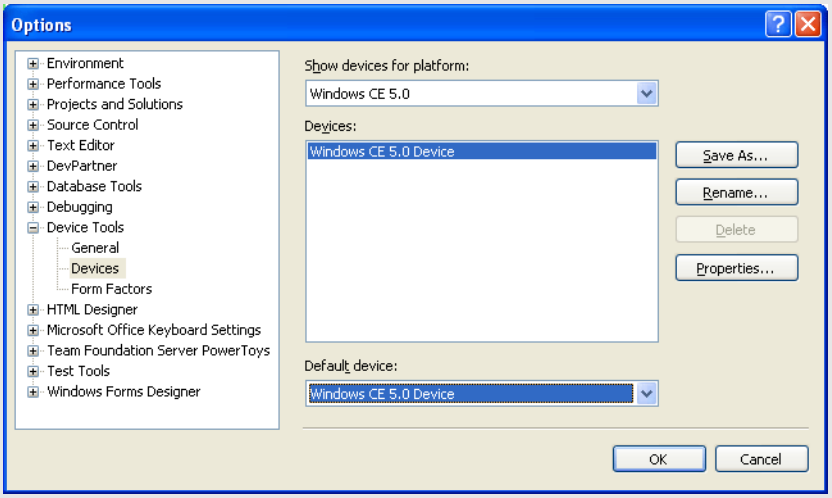
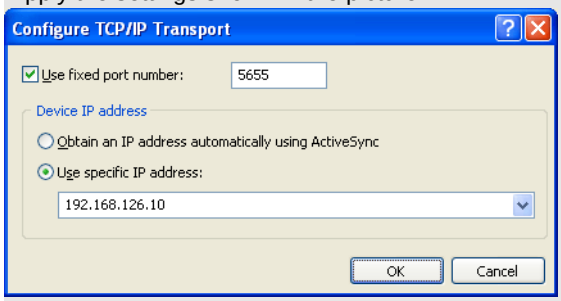
A connection over the LAN port is optional, but useful, as you may need to use RSO or FTP at the same time. To use both connections, your PC requires two NICs.

Continues on next page

Continued

Debugging procedure

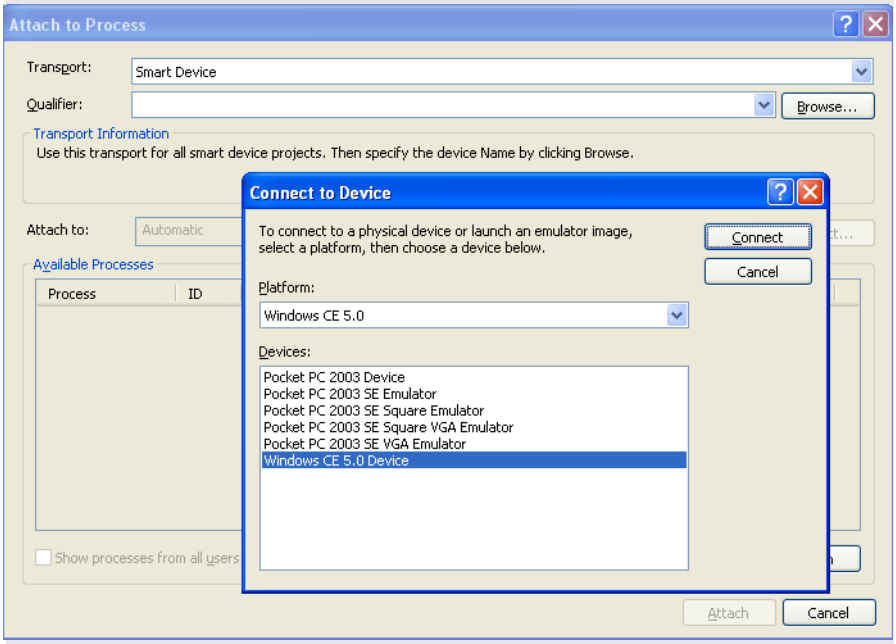
Follow these steps to set up and attach the Visual Studio 2005 SP1debugger to the FlexPendant device:

Step	Action
1	On the **Tools** menu in Visual Studio 2005 click **Options**.
2	In the **Options** dialog expand the **Device Tools** node and select **Devices**. Select *Windows CE 5.0 Device* as shown below.  9.1.4_6 **Note!** If your FlexPendant version is *SxTpu1* the option *Windows CE 5.0 Device* is not available. Instead you should select *Pocket PC 2003 Device*.
3	When the right device has been selected click **Properties** and then **Configure**.
4	Apply the settings shown in the picture.  9.1.4_7

Continues on next page

8.1.3. Debugging the FlexPendant device

Continued

Step	Action
5	On the **Debug/Tools** menu in Visual Studio click **Attach To Process**. In the dialog select *Smart Device* for **Transport** and click the **Browse** button to specify platform and device. Then click **Connect**.  9.1.4_8
6	In the **Available Processes** list select *taf.exe*. Click **Attach**.
7	Set a break point in your source code.
8	On the FlexPendant, press a button of your application or something else that will make program execution hit the breakpoint.

NOTE!

If your PC is NOT using XP with VS 2005 SP1 you need to find information on how to attach the VS debugger for your specific environment on the RAB User Forum.

RAB User Forum

The *User Forum* of ABB's *RobotStudio Community* has a section dedicated to Robot Application Builder. Here beginners as well as experts discuss code and solutions online. This is also where you find RAB releases for free download and any information that the development or support team want to share with you. See *RobotStudio Community on page 17*.

8.1.4. Troubleshooting FlexPendant applications

Overview

If you encounter problems when running your application there are a few things you should do before contacting your service organization. The following steps represent a rough guideline:

	Action
1	Is it possible to start your application? If not see *FlexPendant application does not start on page 289*.
2	Have you checked the RAB Release Notes? Many questions will find an answer in the Release Notes of the specific release. These are available on the RW DVD and at the Software Download Site.
3	Have you tried to pinpoint the problem by debugging the FlexPendant? If not see *Debugging the FlexPendant device on page 286* for information about how to do it.
4	Have you tried to get debug printouts? See *Debug output on page 279* for further information.
5	Is the problem FlexPendant hangings? Make sure you use `Invoke` when modifying the user interface due to a robot controller event. See *GUI and controller event threads in conflict on page 68* and *Invoke method on page 68*. When a hanging occurs attach the Visual Studio debugger to the FlexPendant. On the **Debug** menu, point at **Windows** and select **Threads**. Examine the threads to discover any deadlocks.

TIP!

If you still have not found a solution to your problem, take a look at the *User Forum* of *RobotStudio Community*, which includes a forum dedicated to discussion and chat on Robot Application Builder topics. See *RobotStudio Community on page 17*

FlexPendant application does not start

If you are unable to start your application the table below suggests possible scenarios.

Problem	Possible solution
The proxy assembly (*.gtpu.dll) is not built.	Correct any parameter error in the TpsView attribute. See *FlexPendant TpsView attribute on page 54*.

Continues on next page

8.1.4. Troubleshooting FlexPendant applications

Continued

Problem	Possible solution
The ABB Compliance Tool complains it cannot find the C# compiler.	It happens that the installation of Visual Studio does not set the path to the C# compiler properly. The C# compiler is necessary for running the ABB Compliance Tool. Confirm that the C# compiler is available by starting a command window and run “CSC.EXE”. If the result is similar to this the path is properly set: C:\>csc.exe Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4 for Microsoft (R) .NET Framework version 1.1.4322Copyright (C) Microsoft Corporation 2001-2002. All rights reserved. fatal error CS2008: No inputs specified If not, find CSC.EXE and copy its path to the **Properties** dialog of your Visual Studio project. Also add the path to the system PATH environment variables.For Windows 2000 Pro and Windows XP find the dialog for this at:Control Panel -> System -> Advanced ->Environment Variables -> System variables-> Add new. Add the path to the directory where the C# compiler is kept. Notice that a semicolon separates the path items.Verify in the command window afterwards that the CSC.EXE runs.
The application does not appear in the ABB menu.	Make sure the robot system has the *FlexPendant Interface* option.
Null reference exception or *“Can’t find object”* when tapping the application icon in the ABB menu.	Make sure all arguments in the TpsView attribute are appropriate. See *FlexPendant TpsView attribute on page 54*.
The bitmap constructor fails when the real FlexPendant tries to load your images, in the virtual environment this does not happen.	Change your images if they use more than 256 colors. The operating system of the first generation FlexPendant device (SxTPU1) only supports 256 colors.
The Virtual FlexPendant process remains alive.	If you forget to dispose an object that has a COM reference, such as the Controller object or a Signal, the Virtual FlexPendant process might stay alive. Go through the application and make sure that you dispose of all objects that have a Dispose method.

Important support information

If you cannot solve the problem on your own, make sure that before taking contact with a support organization this information is available:

- Written description of the problem.
- Application source code.
- System error logs.
- A backup of the system.
- Description of work-around if such exists.

Continues on next page

Continued

TIP!

Even better than the complete application is a small repro program, which exposes your problem.

8.2 PC - Debugging and troubleshooting

8.2.1. Debugging

Introduction

Using the Visual Studio debugger for a PC SDK application presents no difference compared to standard .NET development. Debugging can be done using the virtual IRC5 in RobotStudio or a real controller.

Exception error codes

Some exceptions that may appear during development have error codes associated with them. The error codes may help you correct the problem.

Code	Description
0x80040401	The requested poll level could not be met, poll level low is used.
0x80040402	The requested poll level could not be met, poll level medium is used.
0xC0040401	No connection with controller.
0xC0040402	Error connecting to controller.
0xC0040403	No response from controller.
0xC0040404	Message queue full. (Should only happen if asynchronous calls are made.)
0xC0040405	Waiting for a resource.
0xC0040406	The message sent is too large to handle.
0xC0040408	A string does not contain characters exclusively from a supported encoding, e.g. ISO-8859-1 (ISO-Latin1).
0xC0040409	The resource can not be released since it is in use.
0xC0040410	The client is already logged on as a controller user.
0xC0040411	The controller was not present in NetScan.
0xC0040412	The NetScanID is no longer in use. Controller removed from list.
0xC0040413	The client id is not associated with a controller user. Returned only by methods that need to check this before sending request to controller. Otherwise, see 0xC004840F.
0xC0040414	The RW version is later than the installed RobAPI. A newer RobAPI needs to be installed. Returned by RobHelperFactory.
0xC0040415	The major and minor part of the RW version is known, but the revision is later and not fully compatible. A newer RobAPI needs to be installed. Code returned by RobHelperFactory.
0xC0040416	The RW version is no longer supported. Code returned by RobHelperFactory.
0xC0040417	The helper type is not supported by the RW. Helper might be obsolete or for later RW versions, or the helper may not be supported by a BootLevel controller. Code returned by RobHelperFactory.
0xC0040418	System id and network id mismatch, they do not identify the same controller.
0xC0040601	Call was made by other client than the one that made the Connect() call.

Continues on next page

Continued

Code	Description
0xC0040602	File not found on the local file system. Can be that file, directory or device does not exist.
0xC0040603	File not found on the remote file system. Can be that file, directory or device does not exist.
0xC0040604	Error when accessing/creating file on the local file system.
0xC0040605	Error when accessing/creating file on the remote file system.
0xC0040606	The path or filename is too long or otherwise bad for the VxWorks file system.
0xC0040607	The file transfer was interrupted. When transferring to remote system, the cause may be that the remote device is full.
0xC0040608	The local device is full.
0xC0040609	Client already has a connection and can not make a new connection until the present one is disconnected.
0xC0040701	One or more files in the release directory is corrupt and cannot be used when launching a VC.
0xC0040702	One or more files in the system directory is corrupt and cannot be used when launching a VC.
0xC0040703	A VC for this system has already been started; only one VC per system is allowed.
0xC0040704	Could not warm start VC since it must be cold started first.
0xC0040705	The requested operation failed since VC ownership is not held or could not be obtained.
0xC0048401	Out of memory.
0xC0048402	Not yet implemented.
0xC0048403	The service is not supported in this version of the controller.
0xC0048404	Operation not allowed on active system.
0xC0048405	The data requested does not exist.
0xC0048406	The directory does not contain all required data to complete the operation.
0xC0048407	Operation rejected by the controller safety access restriction mechanism.
0xC0048408	The resource is not held by caller.
0xC0048409	An argument specified by the client is not valid for this type of operation.
0xC004840A	Mismatch in controller id between backup and current system.
0xC004840B	Mismatch in key id, i.e. options, languages etc. between backup and current system.
0xC004840C	Mismatch in robot type between backup and current system.
0xC004840D	Client not allowed to log on as local user.
0xC004840F	The client is not logged on as a controller user.
0xC0048410	The requested resource is already held by caller
0xC0048411	The max number of the requested resources has been reached.
0xC0048412	No request active for the given user.
0xC0048413	Operation/request timed out on controller.
0xC0048414	No local user is logged on.
0xC0048415	The operation was not allowed for the given user.

Continues on next page

8.2.1. Debugging

Continued

Code	Description
0xC0048416	The URL used to initialize the helper does not resolve to a valid object.
0xC0048417	The amount of data is too large to fulfill the request.
0xC0048418	Controller is busy. Try again later.
0xC0048419	The request was denied.
0xC004841A	Requested resource is held by someone else.
0xC004841B	Requested feature is disabled.
0xC004841C	The operation is not allowed in current operation mode. For example, a remote user may not be allowed to perform the operation in manual mode.
0xC004841D	The user does not have required mastership for the operation.
0xC004841E	Operation not allowed while backup in progress.
0xC004841F	Operation not allowed when tasks are in synchronized state.
0xC0048420	Operation not allowed when task is not active in task selection panel.
0xC0048421	Mismatch in controller id between backup and current system.
0xC0048422	Mismatch in controller id between backup and current.
0xC0048423	Invalid client id.
0xC0049000	RAPID symbol was not found.
0xC0049001	The given source position is illegal for the operation.
0xC0049002	The given file was not recognized as a program file, e.g. the XML semantics may be incorrect.
0xC0049003	Ambiguous module name.
0xC0049004	The RAPID program name is not set.
0xC0049005	Module is read protected.
0xC0049006	Module is write protected.
0xC0049007	Operation is illegal in current execution state.
0xC0049008	Operation is illegal in current task state.
0xC0049009	The robot is not on path and is unable to restart. Regain to or clear path.
0xC004900A	Operation is illegal at current execution level.
0xC004900B	Operation can not be performed without destroying the current execution context.
0xC004900C	The RAPID heap memory is full.
0xC004900D	Operation not allowed due to syntax error(s).
0xC004900E	Operation not allowed due to semantic error(s).
0xC004900F	Given routine is not a legal entry point. Possible reasons are: routine is a function, or routine has parameters.
0xC0049010	Illegal to move PCP to given place.
0xC0049011	Max number of rob targets exceeded.
0xC0049012	Object is not mod possible. Possible reasons are: object is a variable, object is a parameter, object is an array.
0xC0049013	Operation not allowed with displacement active.
0xC0049014	The robot is not on path and is unable to restart. Regain to path. Clear is not allowed.

Continues on next page

Continued

Code	Description
0xC0049015	Previously planned path remains. Choose to either consume the path, which means the initial movement might be in an unexpected direction, or to clear the path and move directly to next target.
0xC004A000	General file handling error.
0xC004A001	The device is full.
0xC004A002	Wrong disk. Change disk and try again.
0xC004A003	The device is not ready.
0xC004A004	Invalid path.
0xC004A005	Not a valid device.
0xC004A006	Unable to create directory.
0xC004A007	The directory does not exist.
0xC004A008	The directory already exists.
0xC004A009	The directory contains data.
0xC004A00B	Unable to create file.
0xC004A00C	File not found or could not be opened for reading.
0xC004A200	Disable of unit not allowed at trustlevel 0.

8.2.2. Troubleshooting

Overview

If you encounter problems with your PC SDK application follow these steps before contacting ABB support.

	Action
1	See if your problem is in the checklist of the next section.
2	Many questions will find an answer in the Release Notes of the specific RAB release. The document is available on the RW DVD and on the Software Download Site.
3	Pinpoint the problem by debugging your code so that a precise problem description can be provided.
4	See the User Forum of ABB's RobotStudio Community, which includes a forum dedicated to discussion and chat on Robot Application Builder topics.

TIP!

At www.abb.com/roboticssoftware there is a link to the *RobotStudio Community*.

Checklist

- Cannot connect to controllers? Make sure the system on the controller has the RobotWare option *PC Interface*. This applies to both virtual and real controllers.
- Is the problem GUI hangings? Make sure you use `Invoke` when modifying the user interface due to a robot controller event. See *GUI and controller event threads in conflict on page 68* and *Invoke method on page 68* for further information.
- Is the problem related to netscan? If NetworkScanner.Scan does not find the robot controller during netscan you should try to increase the time allowed for scanning. Increase the *networks canner delay* time in an app.config file as explained in *Application configuration file on page 43* or add the time directly in the code like this:
```
NetworkScanner aScanner = new NetworkScanner(); aScanner.Scan();
System.Threading.Thread.Sleep(4000);aScanner.Scan();
```
- Do you get "*Invalid Client ID*" when trying to do a read operation toward the robot controller? If so, the reason is probably that you have forgotten to log on to the controller. To be able to *write* to RAPID data or to the configuration database, for example, you also need to require mastership. For further information see *Logon and logoff on page 214* and *Mastership on page 41*.
- If you are working with a previous version of RAB (earlier than 5.10) you might run into problems related to licence verification? If you get the run-time error *"A valid license cannot be granted for the type ABB.Robotics.Controllers.Licenses.PCSdk. Contact the manufacturer of the component for more information"* when accessing the `NetworkScanner` and the `Controller` classes you need to add a licx file to the project. See *Licenses.licx on page 40* for further information.

Continues on next page

Important support information

If you cannot solve the problem on your own, make sure this information is available when taking contact with ABB's support organization:

- Written description of the problem.
- Application source code.
- System error logs.
- A backup of the system.
- Description of work-around if such exists.

TIP!

Even better than the complete application is a small repro program, which exposes your problem.

8.2.2. Troubleshooting

9 Localizing a FlexPendant application

9.1. Adding support for several languages

Introduction

This chapter provides the information needed to localize a customized application. The FlexPendant has built-in support for localization, and the mechanisms used by the standard FlexPendant applications can also be used by RAB applications.

This enables customized applications to be presented in the active language of the FlexPendant, i.e. the language selected in the standard *view Control Panel - Language*.

For this to work the texts displayed in the customized application must be translated and the application localized as described in this chapter.

Get started

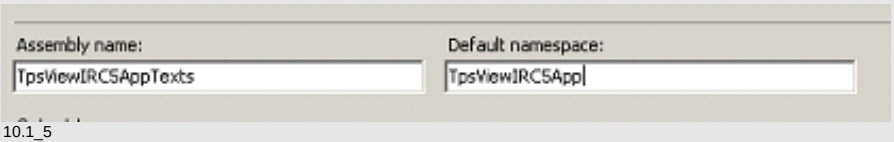
Develop the application using English as the default language. The recommendation is to design, implement and test the application before adding support for other languages. To localize a FlexPendant application carefully complete each procedure of this chapter.

NOTE!

A method is needed to handle localization when new application functionality is added.

1 Create project for text resources

This procedure sets up a separate project for the user interface texts.

Step	Action
1.	Create a new project in the solution. Choose a *Smart Device - Windows CE 5.0 - Empty Project* and name the project *<YourAppName>Texts*. **NOTE!** Both projects should belong to the same solution. The application will now compile to three assemblies: *<YourAppName>.dll* , *<YourAppName>.gtpu.dll* and *<YourAppName>Texts.dll*.
2.	In the *Texts* project add a reference to *System (.Net).*
3.	Set the **Output type** to *Class Library* (this is done in the **Project Properties**).
4.	The namespace used for the *Texts* project must be the same as used for the main project. As the namespace is not visible in the resource file you must change it in the **Project Properties** like this:  10.1_5
5.	Add a *Resources* file to the project by right clicking the project in **Solution Explorer** and selecting **Add New Item**. Set the name of the file to "*strings.resx*". This file will contain the texts of the default language (normally English).

Continues on next page

Continued

Step	Action
6.	Open the resource file and add *name* and *value* for the texts of the default language. Use capital letters in the *name* column.

Data for data

	name	value	comment	type	mimetype
▶	TXT_ABB_MENU_TITLE	Your Application	(null)	(null	(null)
	TXT_MESSAGE_BTN	Message	(null)	(null	(null)
	TXT_BOX_MSG	Press one of the button	(null)	(null	(null)
	TXT_BOX_CAPTION	Message Caption	(null)	(null	(null)
	TXT_RESPONSE_YES	You pressed Yes	(null)	(null	(null)
	TXT_RESPONSE_NO	You pressed No	(null)	(null	(null)
	TXT_RESPONSE_CANCE	You pressed Cancel	(null)	(null	(null)
*					

6.5.1_1

7.	In the *Texts* project create a folder with the culture short form as name, e.g. *de* for German and *sv* for Swedish.

10.1_6

NOTE!

Russian (ru) has been added in RW/RAB 5.11.

CAUTION!

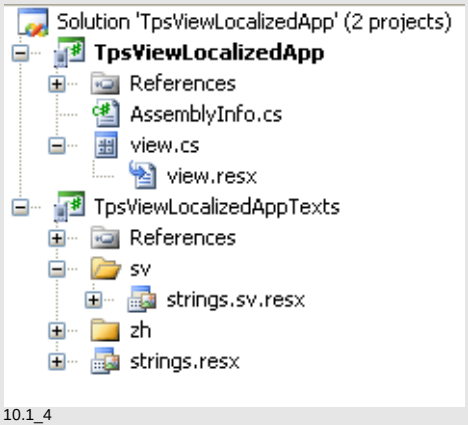
The standard short forms listed above must be used.

NOTE!

To be able to use Chinese or another language with non-western characters, you must use the FlexPendant font, `TpsFont`, for any UI controls. It internally checks what language is currently active on the FlexPendant and uses the correct font for that language.

Continues on next page

Continued

Step	Action
8.	Copy the “strings.resx” file to the folder(s) created in the previous step.
9.	The name of the file used for the foreign resources should be *strings.<culture>.resx*, e.g. “*strings.sv.resx*" as in the picture below. Right click the file and rename it.  10.1_4
10.	Open the resource file and translate the texts in the *value* column. The *name* is the identity of the text and should remain the same. **NOTE!** Obviously, texts might get longer or shorter when translated. You may therefore need to increase the sizes of some GUI controls when you have tested the application.

2 Prepare main project for localization

Follow these steps to add localization to your main project:

Step	Action
1.	Add a reference to *ABB.Robotics.Taf.Base* in the main project.
2.	In the `TpsView` attribute of the view class insert the name of the Texts dll like this: `[assembly: TpsView("ABB_MENU_TITLE_TXT","tpu-Operator32.gif","tpu-Operator16.gif", "TpsViewIRC5App.dll", "TpsViewIRC5App.TpsViewIRC5App", StartPanelLocation.Left, TpsViewType.Static,"TpsViewLocalizedAppTexts.dll", TpsViewStartupTypes.Manual)]`

Continues on next page

Continued

Step	Action

3.	Declare a `TpsResourceManager` object at the top of the class as a private member variable and initialize it in the constructor.
	Add a call to an `InitializeTexts` method.
	
	`//declaration`
	`private ABB.Robotics.Tps.Resources.TpsResourceManager _tpsRM;`
	
	`//constructor method`
	`_tpsRM = new ABB.Robotics.Tps.Resources.TpsResourceManager("TpsViewLocalizedApp.strings", ABB.Robotics.Taf.Base.TafAssembly.Load("TpsViewLocalizedAppTexts.dll"));`
	
	`InitializeComponent();`
	`InitializeTexts();`
	
	NOTE!
	The first constructor argument should be the name of your application with *.strings* as an added suffix. The second argument is the name of the assembly containing the resources.

4.	Implement `InitializeTexts()`. Use the `TpsResourceManager` object and call `GetString()` using the identity (*name*) of the text you want as argument. Depending on the active language of the FlexPendant this call will retrieve the corresponding language resource.
	Example:
	`InitializeTexts()`
	`{`
	`  this.Label1.Text = _tpsRM.GetString("TXT_INSTR_LABEL");`
	`}`
	
	Leave the contents of the `InitializeComponent` method as it is, e.g. `Label1.Text = "Signals"` etc.

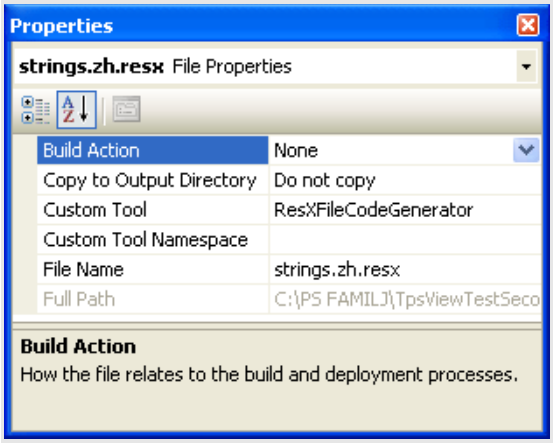
5.	The `TpsResourceManager` object has not yet been created when the application icon and title are to appear in the ABB menu, and therefore another technique must be used to have them correctly displayed. Add a resource *name* for the application title and a *value* in the resource file of each language. In the `TpsView` attribute the first argument is the application title. Replace it with the resource *name*. It may look like this: `[assembly: TpsView("ABB_MENU_TITLE_TXT", ...,... TpsViewIRC5AppTexts.dll")]` The corresponding resource *value* will now be used. In case no resource is found the *name* will be used as is.

Continues on next page

Continued

3 Build satellite assembly

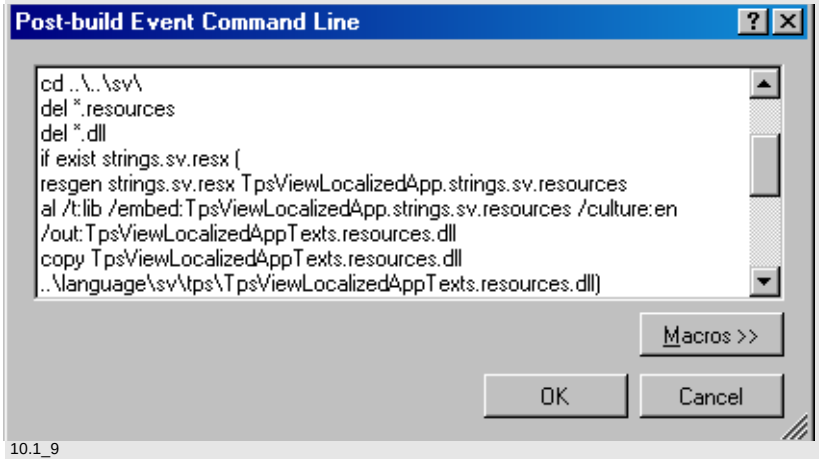
Follow these steps to create a satellite assembly of the localized resx file:

Step	Action
1.	The localized resource file should not be built with the ordinary build process, i.e. the property **Build Action** should be set to "*None*". Right click on the strings.**<culture>**.resx file and select properties:  6.5.1_4
2.	You should now use the Visual Studio 2005 tool *resgen.exe* to compile the resx file to a binary resource. After this the Visual Studio 2005 assembly linker tool *al.exe* should make a satellite assembly of the binary resource. Read the following steps very carefully to make sure the satellite assembly is built correctly. **NOTE!** Localization with Visual Studio 2008 has not yet been tested.

Continues on next page

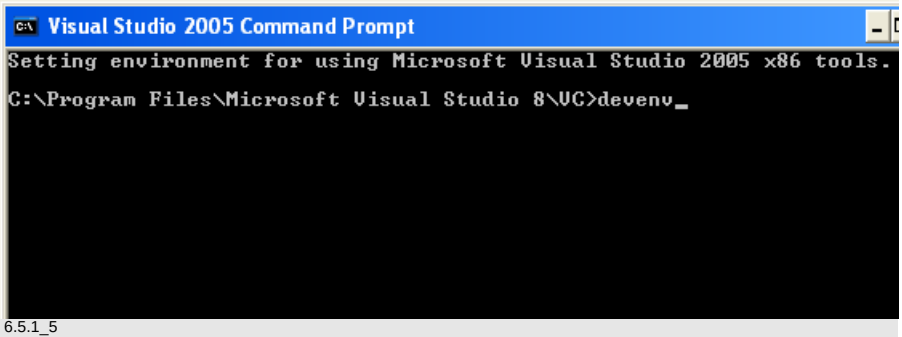
Continued

Step	Action

3.	Create a post-build event in the *Texts* project in order to automate the building process.
	
	Example: `TpsViewLocalizedApp` with resources in Swedish:
	`mkdir ..\..\language`
	`mkdir ..\..\language\sv`
	`mkdir ..\..\language\sv\tps`
	`cd ..\..\sv\`
	`del *.resources`
	`del *.dll`
	`if exist strings.sv.resx (`
	`resgen strings.sv.resx TpsViewLocalizedApp.strings.sv .resources`
	`al /t:lib /embed:TpsViewLocalizedApp.strings.sv.resources /culture:en / out:TpsViewLocalizedAppTexts.resources.dll`
	`copy TpsViewLocalizedAppTexts.resources.dll ..\language\sv\tps\TpsViewLocalizedAppTexts.resources.dll)`
	
	 
	
	10.1_9
	The `resgen` command is written like this:
	`resgen strings.<culture>.resx <Namespace>.strings.<culture>.resources`
	where `<culture>` should be replaced with the correct language short form and `<Namespace>` should be replaced with the application namespace.
	
	The `al` command takes the resulting dll located in the same directory as the resx file and makes a satellite assembly of it:
	`al /t:lib /embed:<Namespace>.strings.<culture>.resources / culture:en /out:<AssemblyName>.resources.dll`
	
	NOTE!
	The name of the satellite assembly will be the same for all localized languages.
	The third argument of the `al` command, *culture:en, should be “en”* . The reason is that the FlexPendant operating system has English as the underlying language.

Continues on next page

Continued

Step	Action
4.	It is necessary to ensure that the post-build step is executed with the correct versions of *resgen.exe* and *al.exe*. The easiest way to do this is to use a VS 2005 Command prompt to build the project (or solution) or to start Visual Studio from that prompt. First click Windows Start menu to launch the Command Prompt (at Programs > Microsoft Visual Studio 2005 > Visual Studio Tools).Then use the command `devenv` in order to start VS 2005. Now open your solution and build it. The post-build command is now guaranteed to execute with the correct settings.  6.5.1_5 **NOTE!** As default, Visual Studio will run post-build commands using the PC's user settings (paths etc.). If you had VS 2003 installed earlier the post-build command is therefore very likely to use the wrong versions of *resgen.exe* and *al.exe*. The procedure described above guarantees that the VS 2005 versions are used. (These can be found at: C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\bin/ resgen.exe C:\WINNT\Microsoft.NET\Framework\v2.0.50727/al.exe)

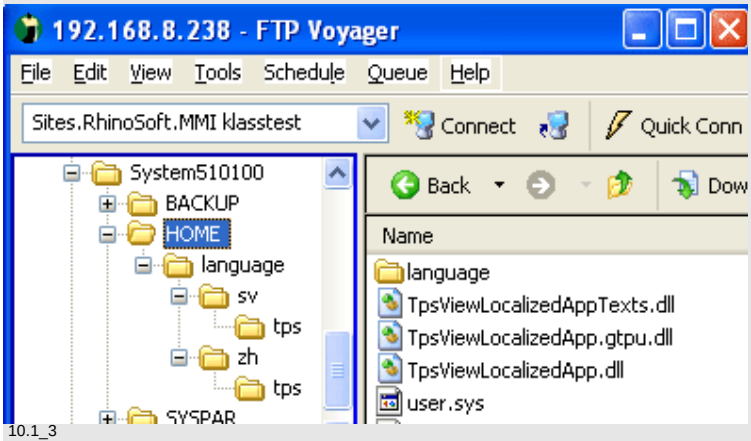
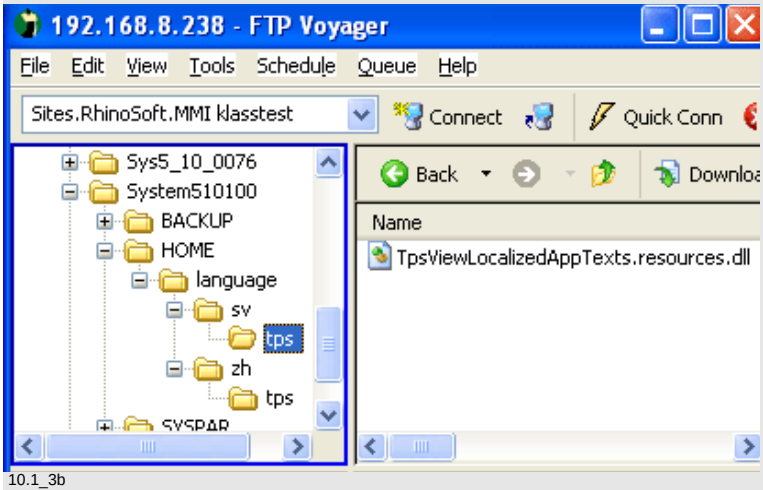
4 Test a localized application

In order to test a localized application, the resources must be correctly organized in the file system of the controller. This should be done in the same way either the test is done on a virtual or a real FlexPendant. Follow these steps to test the application on a real FlexPendant:

Step	Action
1.	Transfer, using an FTP client or the *File Manager* of RobotStudio, your application assemblies to the HOME directory of the active system. (*TpsViewLocalizedApp-Texts.dll, TpsViewLocalizedApp.gtpu.dll, TpsViewLocalizedApp.dll* in the picture below).

Continues on next page

Continued

Step	Action
2.	Copy the *language* folder that was created by the post-build event to the HOME directory.  10.1_3 **NOTE!** It is also possible to have both the assemblies and the language folder located under the SYSTEM folder one level up. The advantage of using HOME is that the files are included in a backup.
3.	Verify that the language folder has folders for each language and that each *tps* folder have a *.resources* dll. If this is not the case, you need to go back to the post-build event of your *Texts* project and check what has gone wrong.  10.1_3b **NOTE!** The name of the satellite assembly is the same for all languages, *TpsViewLocalizedAppTexts.resources.dll* in the example above.
4.	Switch to another language (Control Panel - Languages) and restart the FlexPendant.
5.	Verify that the name of your application in the ABB menu is correct.

Continues on next page

Continued

Step	Action
6.	Open the application and verify that the GUI texts are correctly displayed. Especially verify that text boxes, labels, listview columns etc. are wide enough to display the translated texts.

10 Packaging RAB applications

10.1 Deployment of a PC SDK application

10.1.1. Overview

Introduction

When your application is ready it has to be deployed to the customer's PC. This chapter gives information about the facilities for deployment included in the RAB installation.

NOTE!

Neither RAB nor a RAB licence need to be installed on the PC that will host your application. Furthermore, from RAB 5.10 you do NOT need to add the licence key to your project as described in *Licenses.licx on page 40*, as deployed PC applications no longer perform license verification when executing.

Facilities for deployment

In the *redistributable* folder at C:\Program Files\ABB Industrial IT\Robotics IT\Robot Application Builder\ there are some files to be used for deployment of a PC SDK application:

- ABBControllerAPI.msm
- ABB Industrial Robot Communication Runtime.msi

These two packages include all dependencies a PC SDK application has apart from .NET 2.0.

Continues on next page

10.1.1. Overview

Continued

ABBControllerAPI.msm

A PC SDK application cannot execute without the PC SDK assemblies it references. For your convenience, the ABBControllerAPI merge module contains the PC SDK assemblies. Add it to your install program to have them installed in the Global Assembly Cache (GAC).

The GAC is automatically installed with the .NET runtime. It enables a PC to share assemblies across numerous applications. If the customer's PC has Robot Studio Online of the same release as the PS SDK used to create the application the PC SDK dlls your application needs should be in the GAC already.

NOTE!

If you want to create an msi file (or a setup.exe) of the msm file, you can include the ABBControllerAPI.msm file in a Visual Studio **SetUp Project**.

NOTE!

Before, ABBControllerAPI.msm worked only with *InstallShield*. This problem has now been resolved.

ABB Industrial Robot Communication Runtime.msi

For a PC SDK application to be able to connect to a controller either RobotStudio or Robot Communications Runtime is required. If RobotStudio is not installed on the PC that will host your application, Robot Communications Runtime needs to be included in your installation. ABB Industrial Robot Communication Runtime.msi can be used for redistribution as a separate installation.

10.2 Deployment of a FlexPendant SDK application

10.2.1. Overview

Introduction

For the end customer to be able to use your application, it has to be deployed to the customer's robot controller. This is done when the customer robot system is created, by using the *System Builder* in RobotStudio. The custom application can then either be added as an additional option by using a license key, or added to the *Home* directory of the controller file system by using one of the dialogs of the *System Builder* wizard.

For an application with multi-language support there are a few more things to deal with.

NOTE!

Using an FTP client to upload the application from a PC to a robot controller can be done for testing purposes. It can also be done if the custom application needs to be added to an existing system, which is already running in production. See *Deployment using FTP on page 318* for information about how this is done.

Making a product

These are the steps to make a product of a custom FlexPendant application:

1. Approval of the FP SDK product requirement specification.
2. Design and development of a FP SDK GUI prototype.
3. Approval of the FP SDK GUI prototype.
4. Design and development of a FP SDK functional prototype.
5. Approval of the FP SDK functional prototype.
6. Design and development of a FP SDK product.
7. Approval of the FP SDK product.
8. Design and development of a deployable FP SDK product.

Deployment of a FlexPendant SDK product

Before deploying a custom application you need to consider these issues:

- Should the product be licensed, i.e. be sold as an option?
- Is there a need to localize the product, i.e. create support for native languages?

Depending on how the above questions are answered there are four alternatives (detailed separately in the following sections of this manual):

1. License and localization
2. License but no localization
3. No license but localization
4. No license and no localization

NOTE!

If the product is to be licensed it should be deployed as an additional option. If not, RobotStudio should be used to deploy the application to the *Home* directory of the system.

10.2.2. Deployment of an application without a license

Overview

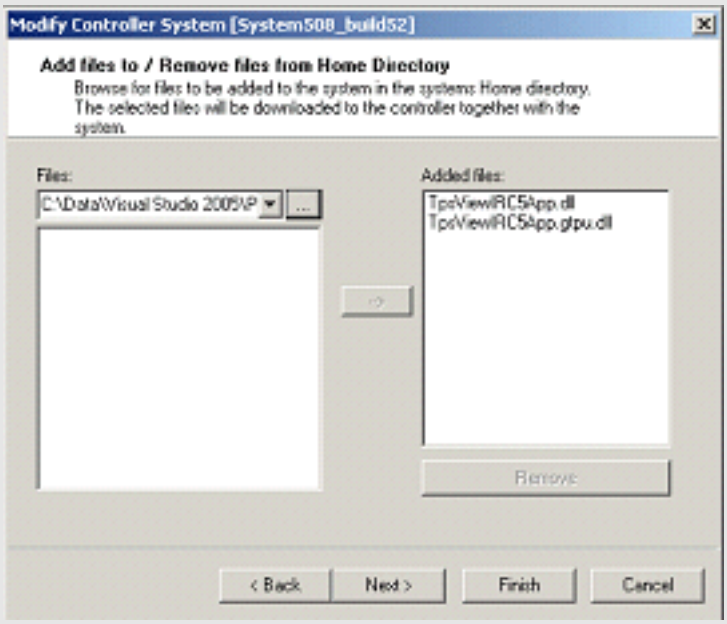
If you do not make an additional option of your application, the end user does not need a license to use it.

When the customer system is created, by using the *System Builder* of RobotStudio, your application should be added to the *Home* directory of the system.

This section gives information about how this is done. The easiest alternative, which offers no support for additional languages, is explained first.

No license and no localization

This is how you deploy an application without license nor multi-language support.

Step	Action
1.	Use *System Builder* in RobotStudio.
2.	Add the application assemblies(.dll) and other resources (‘.jpg, *.gif,*.bmp) to the *Home* directory:  10.1_2
3.	Download the system to the robot controller.

Continues on next page

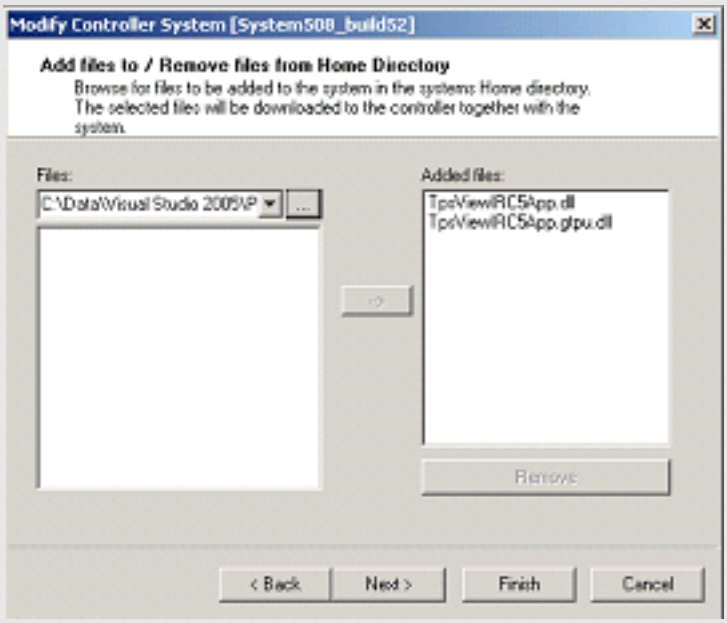
Continued

NOTE!

Having the application deployed to the *Home* directory means it will be included in a system backup.

No license but localization

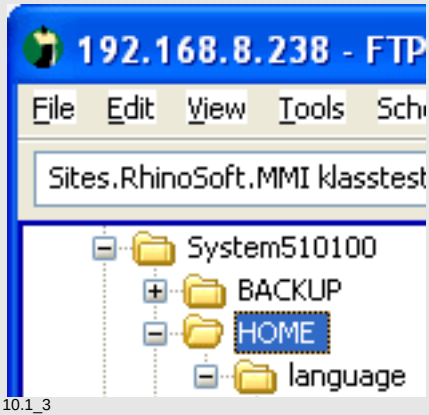
This is how you deploy an application with no license but with multi-language support.

Step	Action
1.	Implement multi-language support. See *Localizing a FlexPendant application on page 299* for information on how to do it.
2.	Use *System Builder* in RobotStudio.
3.	Add the application assemblies(.dlls) and other resources (‘.jpg, *.gif,*.bmp) to the *Home* directory:  10.1_2 **Note!** *TpsViewIRC5AppTexts.dll* is missing in the figure.
4.	Generate the robot system.

Continues on next page

10.2.2. Deployment of an application without a license

Continued

Step	Action
5.	In the generated RobotWare system, add a *language* directory with all supported languages in the *Home* directory. Then for each supported language, add a culture specific and a *tps* directory.  10.1_3 **NOTE!** Standard names (de, zh etc.) must be used for the different languages. See *1 Create project for text resources on page 300* for the complete list.
6.	Transfer each resource binary to the tps sub-directory of the respective culture directory, e.g.: *TpsViewIRC5App.strings.de.resources.dll* to *language/de/tps* directory etc.
7.	Download the system to the robot controller. See *Localizing a FlexPendant application on page 299* for information on how to add support for native languages to your custom application.

10.2.3. Deployment of a licensed application

Overview

When the customer system is created by *System Builder* of RobotStudio, a FlexPendant application can be added to the system as an additional option.

This section describes how to make the additional option, which is necessary for deployment of a licensed custom application. It also describes how the customer installs the option.

CAUTION!

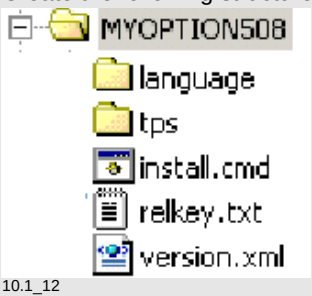
An additional option must be distributed in accordance with RobotWare version and revision. Pay attention if you are using functionality, which has been included in a revision!

NOTE!

An additional option must be deployed with the structure of RobotStudio.

Procedure for making an additional option

This is how you make an additional option of a FlexPendant SDK application.

Step	Action
1.	Implement multi-language support if considered necessary. See *Localizing a FlexPendant application on page 299* for information on how to do it.
2.	Order a license and a CD Key Maker from your local ABB office, who will in turn contact ABB SEROP product support in Sweden.
3.	Create the following structure:  10.1_12 The *language* folder is needed if there is to be support for other languages. **NOTE!** Always use capital letters for the option name.

Continues on next page

Continued

Step	Action

4.	Create the *version.xml* file. It may look like this:
	
	`<Version>`
	`  <Major>5</Major>`
	`  <Minor>08</Minor>`
	`  <Revision>00</Revision>`
	`  <Build>0</Build>`
	`  <Title>MYOPTION508</Title>`
	`  <Description>My Optional FlexPendant Application</Description>`
	`  <Date>2006-04-30</Date>`
	`  <Type>AdditionalOption</Type>`
	`</Version>`
	10.1_13

5.	Create the *install.cmd* file. It may look like this:
	
	`echo -text "Installing My Optional FlexPendant Application"`
	`register -type option -description MYOPTION508 -path $BOOTPATH`
	10.1_14

6.	Create *relkey.txt* file. It may look like this: `title  MYOPTION508` 10.1_15
7.	With the license, which is entered in the CD key Maker, generate a key file including the serial number of the Robot controller, e.g. MYOPTION508.kxt (This is the file the user will enter in *System Builder* when the robot system is created.)
8.	Package the product on CD the way your organization recommends.

NOTE!

If the option is only a FlexPendant SDK application the described procedure is enough. But if the option should also include RAPID and configuration, you need to read about how to make an additional option in another manual, which is for ABB internal use. See *Related information* at the end of this section.

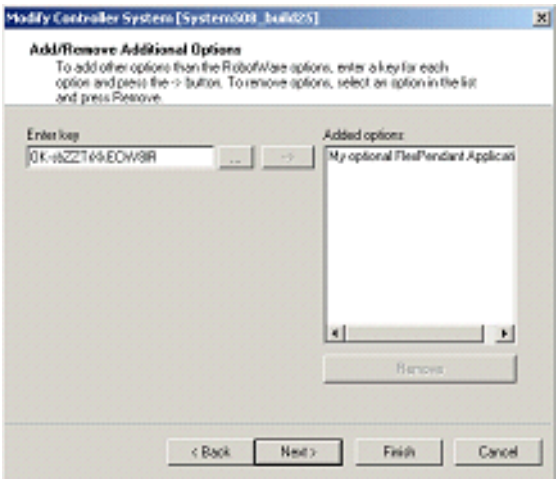
Installing the option at the customer

This is the how the customer installs a licensed application:

1. Install the additional option from a CD.
2. Create the robot system by using System Builder.
3. In the *Add Parameters/Add Additional Options* dialog browse to the key file, e.g. MYOPTION508.kxt which came with the installation of the additional option.

Continues on next page

Continued



10.1_16

Related information

Application manual - Additional Options, 3HAC023668-001

The manual is intended for ABB use only. Contact After Sales in Västerås, Sweden.

10.2.4. Deployment using FTP

Overview

The general rule is that deployment using an FTP client should only be done during the development phase.

If deployment to a customer is done this way each controller has to be individually updated with assembly files as well as graphical and language resources. The organization of files in the robot controller file system is the responsibility of the application developer or the system integrator.

Procedure

Follow these steps to deploy your application using for example the *File Manager* of RobotStudio or an FTP client such as *Ftp Voyager:*

Step	Action
1	On the controller navigate to the system you want to update with the FlexPendant application.
2	Transfer the assembly, the proxy assembly and graphical resources used by the application to the system directory *Home*.
3	For multi-language support, in the *Language* directory create sub-directories for each language using the short name of the culture.
4	Create a *"tps"* sub-directory in each of these directories.
5	Copy each language resource to the *tps* folder of the corresponding culture.
6	Restart the FlexPendant. The custom application should now be accessible from the ABB menu. See *Restart the FlexPendant on page 49* for information about how to restart the FlexPendant but not the controller.

3HAC028083-001, Revision C, en

ABB AB
Robotics Products
S-721 68 VÄSTERÅS
SWEDEN
Telephone: +46 (0) 21 344000
Telefax: +46 (0) 21 132592