C Integration Tutorial

Adapting CODESYS for a toolchain

Project name: C Integration
JIRA-ID: CDS-19680

Version: 3.0
Template: templ_tecdoc_en_V1.0.docx
File name: CODESYSControlV3_C-Integration_Toolchain_Adaptation.docx

CONTENT

1 Enabling C Integration

The C Integration in CODESYS is split into two parts. CODESYS itsself takes care of all steps required to setup and coordinate the building of a C source code component in your project. The end result will be a dynamically linkable runtime module specific to your runtime environment.

For the actual compilation and creation of that module CODESYS must rely on external tools. There is no C Compiler or linker available in CODESYS.

Enabling the C Integration for a device in a project requires the following steps:

1. Creating a Plug-In which implements the toolchain specific build actions for your target device. Especially there must be an implementation of the interface ICSourceComponentBuilder available.
2. Adding a reduced Runtime system SDK to your setup specific to the devices supported by the C-Integration

Whenever a C module is to be compiled, an instance of the ICSourceComponentBuilder is created. The remaining document describes implementation details for the Plug-In.

2 Implementation of an AP Plug-In for building C modules

A Plug-In able to build a C module for a certain device needs to implement specific interfaces. They can be found in the interface Plug-In +CSourceComponentBuilder.

The most important interface is the ICSourceComponentBuilder. An instance of the class implementing this interface will be created for a build process.

The build progresses in the following order:

1. An Instance of ICSourceComponentBuilder is created and the Configure method is called
2. A temporary directory on the filesystem will be created
3. All sources from the project are exported into that folder
4. The method ICSourceComponentBuilder.CreateDynamicObject is called

2.1 Creation of a CSourceComponentBuilder instance

Every PlugIn providing an implementation of an ICSourceComponentBuilder also needs to provide a factory implementing the interface ICSourceComponentBuilderFactory.

When an instance of the component builder needs to be created, each factory available will be asked if it supports a given device (calling of the method **AcceptsDevice**). If the factory returns true, the **Create** method will be called to receive an initialized instance.

2.2 Description of the methods in ICSourceComponentBuilder

2.2.1 Configure

This method is passed an instance of IBuildConfigurator as parameter. That interface contains all relevant information necessary to setup a build. For example it contains a list of all source and headers files part of this C module. It also contains any additional configuration like include and library paths.

This method should be used to set additional paths and defines in the IBuildConfiguration instance which are specific to the build process for the given device. The remaining steps in the build process will then be able to make use of this information.

2.2.2 CreateDynamicObject

This method executes the build itsself. If the build is successfull, it returns the created dynamic object to CODESYS. Any build or project files (e.g. Makefiles) required for the build process are also created in this method.

The build action should output meaningfull information and errors to the users. For this it can use the event ShowBuildMessage. CODESYS subscribes to this event and will store and output all information passed through this event in the message window.

Note: It is up to the Plug-In to do the filtering of its messages.

2.2.3 Property BuildConfig

The Plug-In does not only receive build information from CODESYS through the interface IBuildConfigurator, but does also provide information to CODESYS through the interface IBuildConfiguration. An example of such an information ist the location of the temporary build folder.

An important part of the interface is the dictionary **DatatypeMapping**. This dictionary is used by CODESYS when scanning a C header file for functions and datatypes to export.

In contrast to IEC the actual sizes of datatypes are compiler dependant in C. So the mapping of basic C datatypes, like integer or float, to a corresponding IEC datatype depends on the compiler used later on to create the dynamic module. Therefore only the device specific Plug-In for building the C module can provide a correct mapping of C to IEC datatypes.

The key of the dictionary is a C datatype as defined in the enum TypeClassC. The value returned by the dictionary is a value from the enum TypeClass defined in the core LanguageModel interface in CODESYS.

2.2.4 OpenProjectInIDE

Since CODESYS does not provide any advanced editing features for C sources it provides the possibility to open a C module in a dedicated external IDE.

If a user calls the corresponding command, CODESYS will create an instance of the ICSourceComponentBuilder and call this method. Before doing that, it will check the property **IDESupported.** If that property returns false the user is informed that this feature is not supported and no further actions are executed.

It is important that the IDE is opened non-blocking in a new process. CODESYS can then be informed through the event IDEClosed that the user finished editing the module outside of CODESYS. The C sources will then be checked and updated inside the project, if necessary.

2.3 Environment specific settings

If the Plug-In for building the C module requires settings or configurations of any kind it is free to provide the required dialogs and checks itself through the usual means in CODESYS. The interfaces for the C Integration do not provide any additional means for this.

2.4 Error handling

The handling of errors is generally up to the Plug-In. Build errors (e.g. compile errors) can be output through the mechanism for build messages. More urgent or fatal errors can also be displayed through the MessageService of CODESYS as popups.

Aborting the build when detecting fatal errors in SetupBuild or CreateDynamicObject can be done by throwing Exceptions. Every exception will be catched by CODESYS and lead to the abortion of the build process.

3 Runtime system SDK

Building a dynamic object loadable by the CODESYS runtime requires a set of tools and headers referred to as an SDK.The common storage for this SDKs is the folder named C-Integration the installation root folder of CODESYS.

3.1 Folder layout

Every PlugIn uses its own subfolder and must use the Guid of the ComponentBuilder instance as folder name. If different device versions are to be supported, it is recommended to group the versions by folders.

Eventually a certain layout of an SDK folder is expected. The folder **Components** should contain the headers of runtime components either required for building a dynamic object or which are additionally selectable by the user for usage in their code. The folder **Platforms** contains architecture and OS specific headers.

3.2 User selectable runtime system components

The runtime system components selectable by user for usage in their code through the configuration editor in CODESYS are acquired through the IBuildConfiguration interface.The property **AvailableRuntimeComponents** returns a list of all components visible to the user.

The list returned by the property can either be hardcoded or parsed from configuration files. The implementation details are left open. The reference implementation parses an XML file to get the list of the components.

4 Note for creating precompiled library modules and distributing them as part of IEC libraries

For this use case it is not necessary to create an AP Plug-In supporting a certain compiler toolchain. The compiled component can be created through the usual C development workflows.

There are two important constraints to consider though when compared to a regular runtime system component:

1. The created component may only export functions marked as externally implemented library functions (the default setting when exporting an IEC interface to M4/C in CODESYS)
2. The names of the C functions must bear the string “_cext” in their name. Otherwise the component manager will refuse to make the functions from the modules loaded by the C-Integration available to other components and applications. This is a security feature.

5 Runtime System Requirements

The runtime system does not require any additional customization for specific devices for the C-Integration to work.

The only requirements are a runtime system version of at least V3.5 SP7 and an entry in 3S.dat to license the feature for the device.

6 Security considerations

Before enabling the C-Integration on PLC runtimes there are certain security considerations to be made. Read the following notices carefully and be sure to understand the consequences before actually enabling this feature for PLCs.

Since a C Integration Module is mostly loaded like any other runtime component, it makes its API visible and accessible through the component manager. It will not be possible though to override any other system functions exported by other runtime components. It will also not be possible to override any other externally implemented IEC library functions.

This is achieved by marking C Modules as such for the component manager. This allows the component manager to implement strict policies regarding the handling of API functions exported by Modules from the C Integration.

Nevertheless this feature will enable users to easily write and execute C code on the PLC which has full access to the OS and system library APIs. This means for example that accessing files can be done without the wrapping layer of SysFile by directly using OS functions. Runtime settings like the IecFilePath will not affect this file access and users will be able to work outside of any sandboxes defined.

Change History

Version	Description	Author	Date
0.1	Creation	BeR	19.11.2014
0.2	Adaption to final version of interfaces	BeR	12.03.2015
0.3	Added factory which replaces the target setting.	BeR	27.04.2015
0.4	Added “Security considerations”	BeR	12.05.2015
1.0	Release	BW	02.06.2015
1.1	Updated Security note and added new section for usage of components in libs	BeR	11.06.2015
1.2	Added RTS Requirements	BeR	22.07.2015
1.3	Added legal note	TZ	23.08.2016
2.0	Release after formal review	MaH	23.08.20166
2.1	Legal note removed	GeH	24.10.2016
3.0	Release after formal review	MN	28.11.2016