RapidPro System – Prototyping ECU

# Implementation Guide

Release 5.0 – December 2005

dSPACE

## How to Contact dSPACE


| | |
|---|---|
| Mail: | dSPACE GmbH<br>Technologiepark 25<br>33100 Paderborn<br>Germany |
| Tel.:<br>Fax:<br>E-mail: | +49 5251 1638-0<br>+49 5251 66529<br>info@dspace.de |
| Web: | http://www.dspace.com |
| Technical Support: | support@dspace.de<br>+49 5251 1638-941<br>http://www.dspace.com/goto?support |


## How to Contact dSPACE Support

dSPACE recommends that you use dSPACE Support Wizard to contact dSPACE support.

It is available

- On your dSPACE CD/DVD at \Diag\Tools\dSPACESupportWizard.exe
- Via Start – Programs – dSPACE Tools (after installation of the dSPACE software)
- At http://www.dspace.com/goto?supportwizard
  You can always find the latest version of dSPACE Support Wizard here.

## Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit http://www.dspace.com/goto?support for software updates and patches.

## Important Notice

# Contents


**Documentation Overview** 7
Documentation Types ... 8
About This Guide ... 10
Related Documents ... 12

**Introduction to RapidPro System Used as Prototyping ECU** 13
Field of Application ... 14
Overview of the Participating Hardware and Software ... 16
Specifics of the RapidPro System ... 20
Modularity of RapidPro Hardware ... 21
Signal Mapping to I/O Pins ... 24
RapidPro Hardware Information Required for Implementation ... 27
Effects of Changing Components of a RapidPro System ... 29

**General Instructions** 31
Description of the Workflow ... 32
Flowchart of the Workflow ... 35
Getting Hardware Details for Implementation ... 37

**Using the Demo Application** 41
Overview of the Demo Application ... 42
Working with the Ready-to-Use Demo Application ... 44
Basics on the Prepared Binaries ... 44
How to Start A Ready-to-Use Demo Application ... 45
Working with the Demo Sources ... 51
Basics on the Demo Source Files ... 51
Overview of Implemented Subroutines and Their Configurations 53
Preconditions for Executing the Demo Application ... 59
How to Build and Execute the Demo Application ... 62
How to Prepare Calibration and Measurement ... 64

**Creating the Main Routine of Your Application** 75
Structuring an Application ... 76
How to Create the Main Routine from Scratch ... 78

**Implementing Messages in Your Application** 81
Basic Information on Message Handling ... 82
How to Implement Messages in Your Application ... 85
How to Flush the Message Buffer ... 88

**Implementing Time-Stamping** 91
Basics of Time-Stamping ... 92
How to Implement Time-Stamping ... 93

**Implementing Exception Handling** 95
Basics on Exception Handling ... 96
How to Implement Exception Handling ... 98

**Creating Tasks** 101
Task Types ... 103
Task Execution Order ... 108
How to Implement One Periodic Task ... 110
How to Implement Multiple Periodic Tasks ... 115
How to Implement Aperiodic Tasks ... 121
How to Implement Inherited Tasks ... 126
Task Overrun Handling ... 131
How to Implement Overrun Handling ... 133

**Implementing the dSPACE Calibration and Bypassing Service** 135
Basics on the dSPACE Calibration and Bypassing Service ... 137
Basics on Implementing Memory Pages ... 138
Requirements for the Variable Descriptions of the RapidPro System 141
How to Implement the dSPACE Calibration and Bypassing Service ... 142
Example of the dSPACE Calibration and Bypassing Service in the Demo Application ... 147

**Implementing I/O Access** 151
Overview of Supported I/O Features ..... 152
Features of the RapidPro Control Unit Based on MPC5554..... 153
A/D Conversion ..... 156
Bit I/O ..... 160
1-Phase PWM Signal Generation..... 162
PWM Signal Measurement..... 165
General Concepts of Implementing I/O Features ..... 167
How to Implement A/D Conversion ..... 171
How to Implement Bit I/O on the I/O PLD..... 176
How to Implement Digital Inputs on the eTPU ..... 179
How to Generate 1-Phase PWM Signals ..... 183
How to Measure Frequencies of PWM Signals ..... 188

**Implementing Bus Protocols** 193
Implementing CAN Communication ..... 194
How to Implement CAN Communication ..... 200
Example of CAN Communication in the Demo Application..... 205

**Building and Executing Real-Time Applications** 207
Basics on Building and Downloading Real-Time Applications ..... 208
Basics on Executing Real-Time Applications ..... 212
How to Build and Download Real-Time Applications Using Your Own Makefile..... 213

**Using ConfigurationDesk and CalDesk Simultaneously** 217
Accessing the RapidPro System..... 218
Handling CalDesk and ConfigurationDesk Projects ..... 221

**Limitations and Troubleshooting** 225
Limitations for CAN Communication ..... 226
Limitation for Using Floating-Point Constants ..... 227
Troubleshooting ..... 228

**Index** 229

# Contents

# Documentation Overview

dSPACE offers different types of documents: see *Documentation Types* on page 8.

For brief information on this document, see *About This Guide* on page 10.

You will also find information on the documents that you are recommended to read when you work with the RapidPro system used as a prototyping ECU: see *Related Documents* on page 12.

# Documentation Types

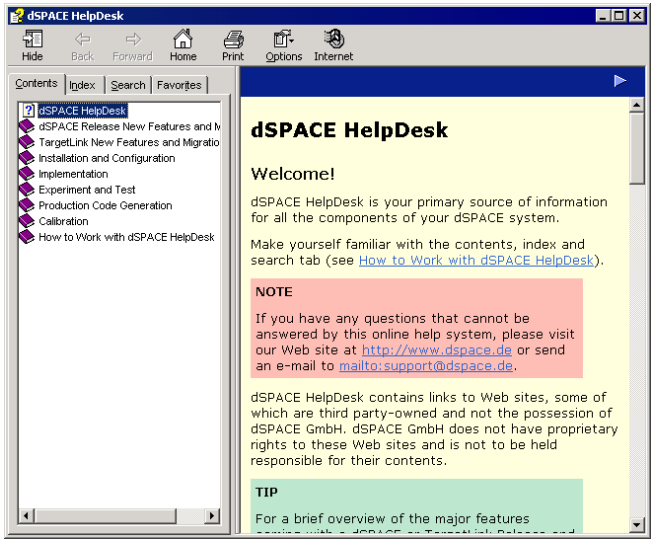
After you install your dSPACE system, you can access the entire documentation as online help or printable Adobe® PDF files. You will also receive a printed version of some important documents.

## dSPACE HelpDesk

dSPACE HelpDesk is your primary source of information on both the hardware and the software of your dSPACE system.

### To open dSPACE HelpDesk

➤ Select **dSPACE HelpDesk** from the **dSPACE Tools** program group of the Windows Start menu.




From each dSPACE HelpDesk page, you can easily search and navigate to the desired information. You also have direct access to printable Adobe PDF files: see *How to Work with dSPACE HelpDesk* in dSPACE HelpDesk.

Only the documents of the products installed on your system are available. The entire product documentation is available if you open dSPACE HelpDesk on the dSPACE CD/DVD.

**dSPACE HelpDesk structure**

The structure of the documents in dSPACE HelpDesk reflects the different phases of your work:

- Installation and Configuration
- Implementation
- Experiment and Test
- Production Code Generation
- Calibration

The topics that are shown depend on your dSPACE system.

**Context-sensitive help**

When you work with any dSPACE software, you can get context-sensitive help via the F1 key and/or Help button.

### PDF Files

All documents are also available as printable Adobe PDF files in `%DSPACE_ROOT%\Doc\Print`: see *How to Work with dSPACE HelpDesk* in dSPACE HelpDesk.

### Printed Documents

You will receive a printed version of the documents that are essential for working away from your PC.

# About This Guide

This implementation guide:

- Introduces you to the RapidPro system used as a stand-alone prototyping ECU.
- Gives instructions and examples for implementing your control models with the C functions provided by the Real-Time Library (RTLib1603) and the Real-Time Kernel (RTK1603) developed for the MPC5554 microcontroller module.
- Describes a demo application, and shows how to use it as a template and to adapt it to your needs.

**Required knowledge**

To work with the implementation software for the RapidPro system efficiently, you must have the following knowledge:

- Experience in programming C-code
- Knowledge of real-time operating systems
- Experience with I/O drivers and CAN communication
- Understanding of embedded coding
- Basics of the RapidPro hardware

## Legend

The following symbols are used in this document.

Warnings provide indispensable information to avoid severe damage to your system and/or your work.

Notes provide important information that should be kept in mind.

Tips show alternative and/or easier work methods.

Examples illustrate work methods and basic concepts, or provide ready-to-use templates.

# Related Documents

Below is a list of documents that you are recommended to read when using the RapidPro system as a stand-alone prototyping ECU.

**Implementation**

- *RapidPro System - Prototyping ECU Implementation Guide* gives instructions and examples to implement your control models on your RapidPro hardware with the help of a demo application.
- *RapidPro System - Prototyping ECU MPC5554 RTLib Reference* provides detailed descriptions of the C functions needed to implement your control models for the MPC5554 via C programs (handcoding).

**Getting Started**

- *RapidPro System Getting Started* describes how to start handling the RapidPro hardware via ConfigurationDesk. The steps necessary for powering and connecting the hardware are described for laboratory conditions and configuration purposes only.

**ConfigurationDesk**

- *ConfigurationDesk Configuration Guide* introduces you to the configuration features and shows how to use them.
- *ConfigurationDesk Configuration Reference* provides detailed information on the menus, context menus, and dialogs contained in ConfigurationDesk.

**CalDesk**

- *CalDesk Tutorial* guides you through your first steps with CalDesk.
- *CalDesk Calibration Guide* explains CalDesk's basic concepts, and provides detailed instructions on carrying out measurement and calibration tasks with CalDesk.
- The *dSPACE Calibration System New Features and Migration* document for CalDesk 1.2.2 provides an overview of the RapidPro support by CalDesk. It gives a description of the new RapidPro device, and provides you with information on how to work with RapidPro devices in CalDesk.

  In future versions of CalDesk, you will find this information in the *CalDesk Calibration Guide*.

# Introduction to RapidPro System Used as Prototyping ECU

**Objective**

The RapidPro system is a modular, flexible hardware platform that you can use for different fields of application. One of these is rapid control prototyping (RCP), in which the RapidPro system is used as a stand-alone prototyping ECU.

**Where to go from here**

Information in this section


| |
|---|
| ***Field of Application*** **on page 14** |
| ***Overview of the Participating Hardware and Software*** **on page 16** |
| ***Specifics of the RapidPro System*** **on page 20** |

# Field of Application

**Objective**

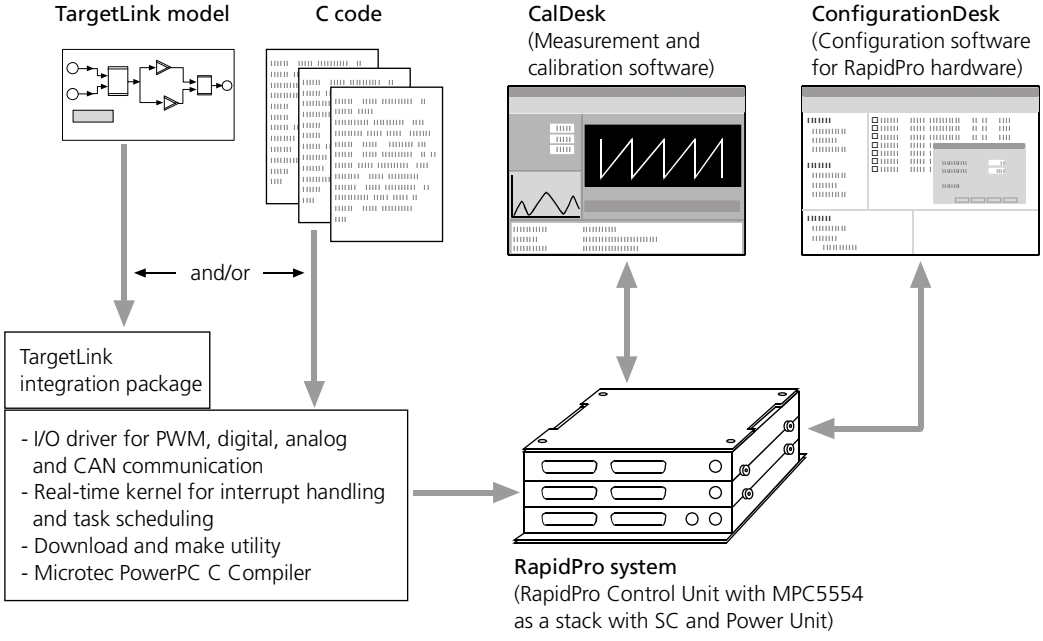
The RapidPro system used as a stand-alone prototyping ECU represents a close-to-production hardware and software environment.

**Core component**

In this scenario the RapidPro Control Unit based on the MPC5554 microcontroller is the core component of the system. The MPC5554 (Copperhead) from Freescale is a microcontroller which is designed to be used in production ECUs.

**Overview illustration**

The illustration below shows the main parts which are involved:



**Software interfaces**

Open software interfaces enable you to integrate:

- C code from different sources
- A measurement and calibration tool

If you already have existing C code, you can integrate it and afterwards execute it on the RapidPro Control Unit. There are I/O drivers for PWM generation, PWM measurement, digital I/O, A/D conversion and CAN communication. The dSPACE Real-time Kernel provides functions for interrupt handling and task scheduling.

With the software provided, the RapidPro system can be used to develop control applications in fields such as transmission, chassis, body, and drives control.

**RapidPro hardware**

For flexible sensor adaptation, there are slots for several signal conditioning modules on the RapidPro Control Unit. If you need power stages for actuators or additional signal conditioning, you can add further RapidPro unit's to build a stack.

You can manage the RapidPro hardware with ConfigurationDesk, the configuration software from dSPACE.

**TargetLink integration package**

dSPACE offers an additional TargetLink integration package for model-based code generation. TargetLink from dSPACE is a software tool, that generates C code straight from Simulink block diagrams or Stateflow state diagrams.

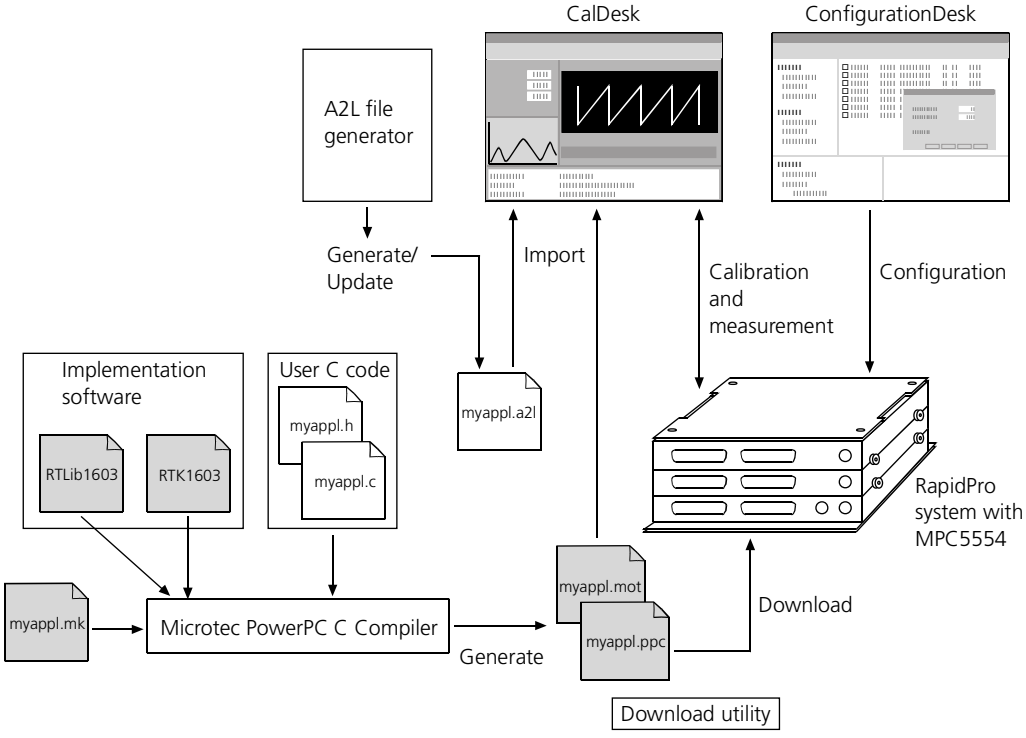
With the help of the integration package, you can implement TargetLink models on the Control unit's microcontroller.

The integration package simplifies the integration of I/O, performs the build and download process at a click, and provides a TargetLink example model. A2L (ASAM-MCD 2MC) files are generated via TargetLink.

# Overview of the Participating Hardware and Software

**Overview illustration**

The following illustration shows the hardware and software components involved.



**Implementation software**

The implementation software for the RapidPro Control Unit based on the MPC5554 microcontroller module (DS1603) provides functions and macros to implement your application via handcoded user C code.

| Software | Description |
|---|---|
| RTLib1603 | The Real-Time Library (RTLib) provides functions and macros to implement:<br>• Initialization of your RapidPro system<br>• Power supply management of the RapidPro system<br>• I/O handling (1-phase PWM signal generation, PWM signal measurement, A/D conversion, bit I/O)<br>• CAN support<br>• Message handling<br>• Time-stamping<br>• Exception handling<br><br>The RTLib1603 also includes the dSPACE Calibration and Bypassing Service (DsECU service). This service provides functions to control communication between an ECU and CalDesk, the measurement and calibration software from dSPACE. |
| RTK1603 | The dSPACE Real-Time Kernel (RTKernel) is used for task management and interrupt handling of your real-time application. The RTK1603 provides functions for:<br>• Initialization of the kernel<br>• Task handling<br>• Task overrun handling<br>• Task time management<br>• Interrupt handling |

- To learn to implement an application on your RapidPro system, you should follow a recommended workflow. Refer to *General Instructions* on page 31.
- To simplify your work, dSPACE provides a demo application. You can use this as a template and adapt it to your requirements. For an introduction to the demo application and how to use it, refer to *Using the Demo Application* on page 41.

**Download utility**

Building and downloading real-time applications is a step-by-step process. To simplify this process, dSPACE provides an utilitiy with which you can compile, link and, download your applications to your hardware.

- The build and download process is managed by the `down1603.exe` utility.
- `down1603.exe` can use a custom makefile (`myappl.mk`), which contains instructions for building, compiling and generating a specific application.

For basics, instructions for downloading, and information on the files which are generated during the download process, refer to *Building and Executing Real-Time Applications* on page 207.

**Microtec PowerPC C Compiler**

For code generation, you need the Microtec PowerPC C Compiler Version 3.2.

You can order the required compiler together with your RapidPro system. The Microtec PowerPC C Compiler is then installed automatically with the dSPACE software.

**RapidPro Hardware**

Because of the modularity of the RapidPro hardware, you have to familiarize yourself with some specifics of the hardware. For details, refer to *Specifics of the RapidPro System* on page 20.

**Connection to the host PC**

If you want to configure your RapidPro system with ConfigurationDesk, download the compiled application or access it via CalDesk, the Control Unit of your RapidPro system must be connected to the host PC.

The communication is established via a specific communication module (USB prototyping interface for measurement, calibration and hardware configuration), which is installed in the Control Unit.

For instructions, refer to *Putting RapidPro Hardware into Operation in the RapidPro System – Hardware Installation Guide*.

**ConfigurationDesk**

ConfigurationDesk allows intuitive and efficient configuration of the RapidPro hardware. ConfigurationDesk provides access to all hardware details and enables you to monitor the hardware states during operation.

**CalDesk**

CalDesk from dSPACE is a measurement and calibration software environment for function prototyping, ECU calibration, measurement, and data analysis all in one tool.

The communication between CalDesk and the RapidPro hardware is controlled by the functions of the dSPACE Calibration and Bypassing Service.

**A2L file generator**

ASAM-MCD 2MC (A2L) files contain information on parameters and measurement variables of the application that is implemented on an ECU (in our case: the real-time application running on the RapidPro system).

A2L files must be provided and generated by yourself. Each time your application is modified, you must check your A2L file and probably update it before you can use CalDesk.

If you need an A2L file generator (editor), contact dSPACE for further information.

**Required software versions**

The following software versions are required for using the RapidPro system as a stand-alone prototyping ECU:

| Tool | Version |
|---|---|
| ConfigurationDesk | 1.1 |
| CalDesk [1)] | 1.2.2 |
| Microtec Power PC C Compiler | 3.2 |

1) Must be installed with the CalDesk Prototyping Module.

# Specifics of the RapidPro System

**Objective**

The architecture of a RapidPro system differs from other dSPACE hardware. Its high modularity affects implementation work and hardware handling. So it is necessary to consider some specials.

**Where to go from here**

Information in this section


| |
|---|
| ***Modularity of RapidPro Hardware*** **on page 21** |
| ***Signal Mapping to I/O Pins*** **on page 24** |
| ***RapidPro Hardware Information Required for Implementation*** **on page 27** |
| ***Effects of Changing Components of a RapidPro System*** **on page 29** |

# Modularity of RapidPro Hardware

**Objective**

A RapidPro system is built up individually by choosing different RapidPro units and modules.

**RapidPro units**

A RapidPro system is a combination of the following RapidPro units:

- RapidPro Control Unit:
  - Modular microcontroller unit
  - Is equipped with a MC-MPC5554 1/1 microcontroller module (if the RapidPro system is used as a prototyping ECU)
  - Is equipped with a COM-USB-PI 1/1 communication module (if the RapidPro system is used as a prototyping ECU)
  - Provides 6 slots for signal conditioning (SC) modules
- RapidPro SC Unit (optional):
  - Modular signal conditioning unit
  - Provides 8 slot for signal conditioning (SC) modules
- RapidPro Power Unit (optional):
  - Modular power stage unit
  - Provides 6 slots for power stage (PS) modules.

If your RapidPro system only contains the RapidPro Control Unit with applicable SC modules, you can implement applications which require up to 30 analog input channels, for example, for A/D conversion, or up to 48 digital input or output channels, for example, for PWM signal generation and measurement.

If your application requires more channels for acquiring sensor signals than are provided by the SC modules of the Control Unit, you must add one or more SC Units to your RapidPro system. If your application requires signals to drive actuators and other loads, you must add one or more Power Units to your RapidPro system.

Because of the modular concept of the RapidPro hardware, you can use only implementation features that are supported by the signal conditioning modules and/or power stage modules installed on the units of your RapidPro system.

For example, if you want to implement A/D conversion, the RapidPro system must contain modules providing analog input channels. If no suitable modules are available, you cannot use the A/D conversion feature of the Control Unit's microcontroller.

**Combining units to a stack**

As described above, the RapidPro Control Unit can be combined with RapidPro SC and/or Power Units to a stack to build a common system. In this case an integrated unit connection bus (UCB) connects the SC and Power Units electrically to the Control Unit without external wiring. This configuration is named Stack with UCB.

**Modules for Control Unit**

The Control Unit provides 6 slots for signal conditioning modules (SC modules). The slots can be equipped with up to 6 SC modules of the same type, or any combination of types. Available modules are, for example:

| SC Module | Description |
|---|---|
| SC-AI 4/1 | Analog input module with 4 channels |
| SC-AI 10/1 | Analog input module with 10 channels (requires 2 slots) |
| SC-DI 8/1 | Digital input module with 8 channels |
| SC-DO 8/1 | Digital output module with 8 channels |
| … | … |

**Modules for SC Unit**

An SC Unit provides 8 slots for signal conditioning modules. It can be equipped individually with the same modules as are available for the Control Unit.

**Modules for Power Unit**

A Power Unit provides 6 slots for power stage modules (PS modules), for example:

| PS Module | Description |
|---|---|
| PS-FBD 2/1 | Full-bridge driver module with 2 channels |
| PS-LSD 6/1 | Low-side driver module with 6 channels |
| PS-HSD 6/1 | High-side driver module with 6 channels |
| … | … |

**Hardware details**

For a complete list of available SC and PS modules, refer to *Hardware Overview* in the *RapidPro Installation and Configuration Reference*. This overview guides you also to hardware details of the RapidPro units and modules.

# Signal Mapping to I/O Pins

**Objective**

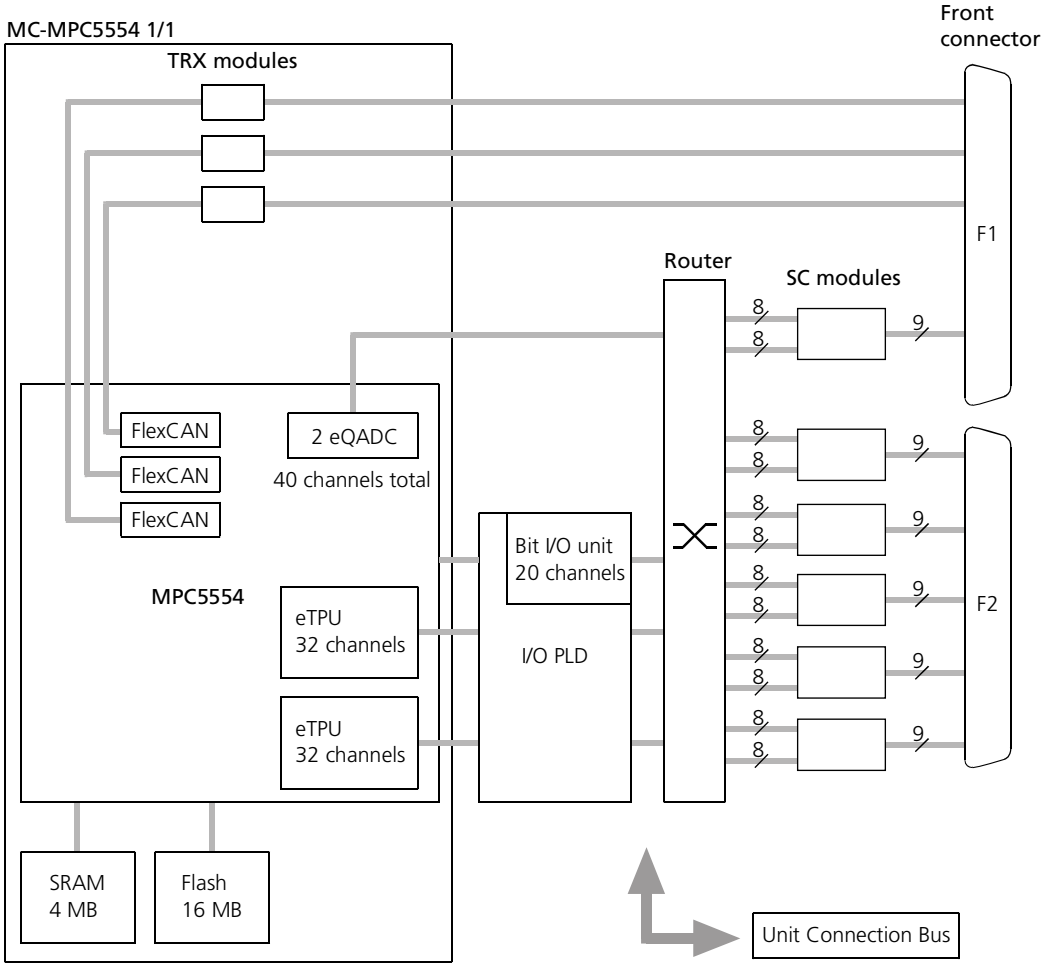
RapidPro hardware does not have dedicated I/O mapping. The signal mapping to the I/O pins depends on various factors, for example, the modules which are installed in your RapidPro system. You can get the I/O mapping and pinout information of your hardware via ConfigurationDesk.

**I/O mapping**

Most hardware platforms used for rapid prototyping provide static I/O mapping. That means that each pin of an I/O connector is reserved for one specific purpose. A microcontroller signal is directly connected to the corresponding I/O pin.

With the RapidPro system, all analog and digital signals to be processed by the microcontroller are first connected to a corresponding signal conditioning or power stage module. Module types and the slots on which modules are installed can differ between the RapidPro systems because of their hardware modularity. The route of a signal from the I/O connector via the SC or PS module to the MPC5554 is controlled by routing code.

The illustration below shows the mapping between the microcontroller signals and the I/O pins of the Control Unit's front connectors in simplified form.



**Signal routing**

Theoretically, the router (shown above) makes it possible to mount any type of SC module on any SC slot on the carrier board (or any type of PS module on any PS slot in the Power Unit). In practice, the router is controlled by a default routing code, which allows a restricted selection of slot positions. For further information, refer to *Changing the Installation of SC/PS/COM Modules* in the *RapidPro System – Hardware Installation and Configuration Guide*.

The routing code is provided by dSPACE and it cannot be changed by the user. dSPACE can change the routing code on request.

The code varies according to the unit type, the unit assembly, and the microcontroller used in the Control Unit. The routing code can be identified by its RoutingID. This is a unique identification number and can be displayed via ConfigurationDesk.

# RapidPro Hardware Information Required for Implementation

**Objective**

Because of the dependencies of available software features and installed RapidPro hardware, you need specific information about the hardware when you implement your application.

**TopologyID**

This number identifies the hardware topology of your RapidPro system. The topology of your RapidPro system depends on the installed types of units and modules, and the slots the modules are installed on. Systems with identical topology and routing code can be identified by the same TopologyID.

The TopologyID of your RapidPro system must be identical with the TopologyID specified in the initialization section of your application. Therefore the DS1603RTlib uses the `ds1603_topology_id_check` function to check consistency.

If the TopologyID in your source file is not identical with the TopologyID of your RapidPro system, your application starts briefly and then immediately stops running during checking the TopologyID. A corresponding error message appears. Thus, this check guarantees that an application implemented for specific hardware equipment cannot damage the hardware and/or connected devices, if it is downloaded to different hardware, or to the same hardware after it was modified.

**I/O mapping information**

The I/O mapping depends on the hardware topology and the routing code stored on the hardware. ConfigurationDesk lets you export the required mapping information to a pinout information file. This file contains the available channels of your RapidPro system, and shows the mapping of a microcontroller signal (channel) to the corresponding I/O channel of the SC- or PS module and it's I/O pin.

You need this information to specify parameters used in hardware-related RTLib functions for I/O drivers. For example, before you implement a RTLib function using analog input channel 20 on the MPC5554, you have to check if the channel is mapped (routed) to an I/O pin of the system via a signal conditioning module.

The I/O mapping information also contains default channel names. You can change them via ConfigurationDesk, to define your own channel names which comply with your project definitions. For example, the default label *analog input channel 30* can be renamed *TemperatureSensor* to identify it as the temperature sensor signal. For instructions, refer to *How to Communicate via Channel Names* in the *RapidPro System – Getting Started* document.

**Installing of CAN transceivers**

The MC-MPC5554 1/1 microcontroller module provides three slots for CAN transceiver modules. These modules give you access to the three FlexCAN controllers on the MPC5554.

To specify parameters used in hardware-related RTLib functions for CAN communication, you must know the available CAN transceivers, their types, and the slots they are installed in. The slot number corresponds to the CAN controller number of the MPC5554.

**Getting the required information**

You can access the required details via ConfigurationDesk. For further information, refer to *Getting Hardware Details for Implementation* on page 37.

# Effects of Changing Components of a RapidPro System

**Objective**

When you modify your RapidPro system, there are different effects depending on the changes you made. Hardware modifications change the TopologyID of the RapidPro system and the I/O mapping.

**If hardware changes have been made, you must compare the already specified channels with the new generated pinout information file before executing the application to avoid hardware damage.**

**Modifications that change the TopologyID**

The TopologyID of your system changes when you:

- Add or remove a module
- Replace a module with one of another type
- Change the slot of a module
- Add or remove a unit to/from the system

Adding or removing a unit is possible, but must be performed by dSPACE.

**Modifications that do not change the TopologyID**

Some modifications have no effect on the TopologyID. These are:

- Configuring a module
- Changing the channel names in ConfigurationDesk
- Replacing modules with ones of the same type but different hardware configurations
- Replacing a defective module with a module of the same type

**Modifications that change the I/O mapping information**

If you perform one of the following actions, the I/O mapping changes and the pinout information has to be updated:

- Change the name of a channel
- Change of RapidPro hardware (= change of the TopologyID):
  - Add or remove a module
  - Replace a module with one of another type
  - Change the slot of a module
  - Add or remove a unit to/from the system

Adding or removing a unit is possible, but must be performed by dSPACE.

# General Instructions

**Objective**

Implementing a prototyping application on a RapidPro system is almost the same as on other hardware platforms. You must consider some specifics.

**Where to go from here**

Information in this section

| |
|---|
| ***Description of the Workflow*** **on page 32**<br>To implement a prototyping application on your RapidPro system, you should follow the recommended workflow. |
| ***Flowchart of the Workflow*** **on page 35**<br>The flowchart gives you a graphical overview of the workflow. |
| ***Getting Hardware Details for Implementation*** **on page 37**<br>Before you implement your application, you must get specific details about the RapidPro hardware on which your application will be executed. |

# Description of the Workflow

**Objective**

The workflow shows the recommended work steps you should follow to learn how to implement a prototyping application on a RapidPro system. Each work step represents a function module or a collection of functions that are required to implement the feature of the work step. The steps required for building and experimenting with the application are also included. It is important to be aware of the workflow, because most of the work steps depend on each other.

**Demo application as template**

Instead of starting from scratch, you can use the source files of the demo application as a template for your own application. The template guarantees that functions are called in the correct order. If you adapt the template to your requirements, you must know the purpose of the functions, and the dependencies between them.

**Using ConfigurationDesk**

ConfigurationDesk is needed for some work steps, such as getting the TopologyID of your system, monitoring your application, and generating the pinout information in tabular form (CSV or XLS format) required for parameterizing channels used for I/O access. You must create or open a ConfigurationDesk project before you can use these features.

For further information about using ConfigurationDesk, refer to the *ConfigurationDesk Configuration Guide*.

**Typical workflow**

**The following steps comprise the workflow for implementing and handling applications**

**1** Creating the main routine

Your application code is to be structured into the main routine and several subroutines. In the main routine, which is the entry point of your application, you first initialize the RapidPro system and access to the real-time library of the RapidPro Control Unit (RTLib1603), and then set up communication between RapidPro system and the host PC in a background loop.

For further information, refer to *Creating the Main Routine of Your Application* on page 75.

2 Implementing messages

You can specify messages in the source code to make your application talk to you.

For further information, refer to *Implementing Messages in Your Application* on page 81.

3 Building and executing applications

The application can be built and executed now for the first time. You can monitor your application using ConfigurationDesk. For example, you can see the implemented messages in ConfigurationDesk's Log Viewer.

You must rebuild your application if you modified the source code.

For further information, refer to *Building and Executing Real-Time Applications* on page 207.

4 Implementing time measurement

You can use time-stamping functions in your source code, for example, to calculate the execution time.

For further information, refer to *Implementing Time-Stamping* on page 91.

5 Handling exceptions

You can specify the functions to be executed if an exception occurs in your application, for example, numerical overflows.

For further information, refer to *Implementing Exception Handling* on page 95.

6 Defining tasks

Tasks are functions to be executed at defined time intervals or external events. If you implement multiple tasks, you can influence the task scheduling by specifying task priorities. Tasks are bound to interrupts to control task execution. You can also specify the reaction to task overruns. The handling of tasks is based on the functions of the dSPACE Real-Time Kernel.

For further information, refer to *Creating Tasks* on page 101.

**7** Integrating the dSPACE Calibration and Bypassing Service

The dSPACE Calibration and Bypassing Service provides communication between your RapidPro system as prototyping ECU and CalDesk as measurement and calibration tool. First you create a basic framework for the ECU service in the background of your main application, then you must enlarge the framework in your subroutines, for example, when you implement I/O access.

For further information, refer to *Implementing the dSPACE Calibration and Bypassing Service* on page 135.

**8** Creating an ASAP2 file

Before you can measure variables or calibrate parameters of your application using CalDesk, you must create an A2L file according to the ASAM-MCD 2MC standard.

For further information, refer to *How to Prepare Calibration and Measurement* on page 64.

**9** Measuring and calibrating

You can create a CalDesk project to display certain values calculated in your application. You can compare the results with the expected behavior of your algorithms. Every time your application is modified, you must check your A2L file and probably update it before you can use CalDesk.

For further information, refer to *Implementing the dSPACE Calibration and Bypassing Service* on page 135.

**10** Integrating I/O functionality

To implement I/O access in your application, RTLib1603 provides several I/O drivers for the RapidPro system, for example, for generating and measuring PWM signals. The I/O features that you can implement depend on the installed hardware components of your RapidPro system.

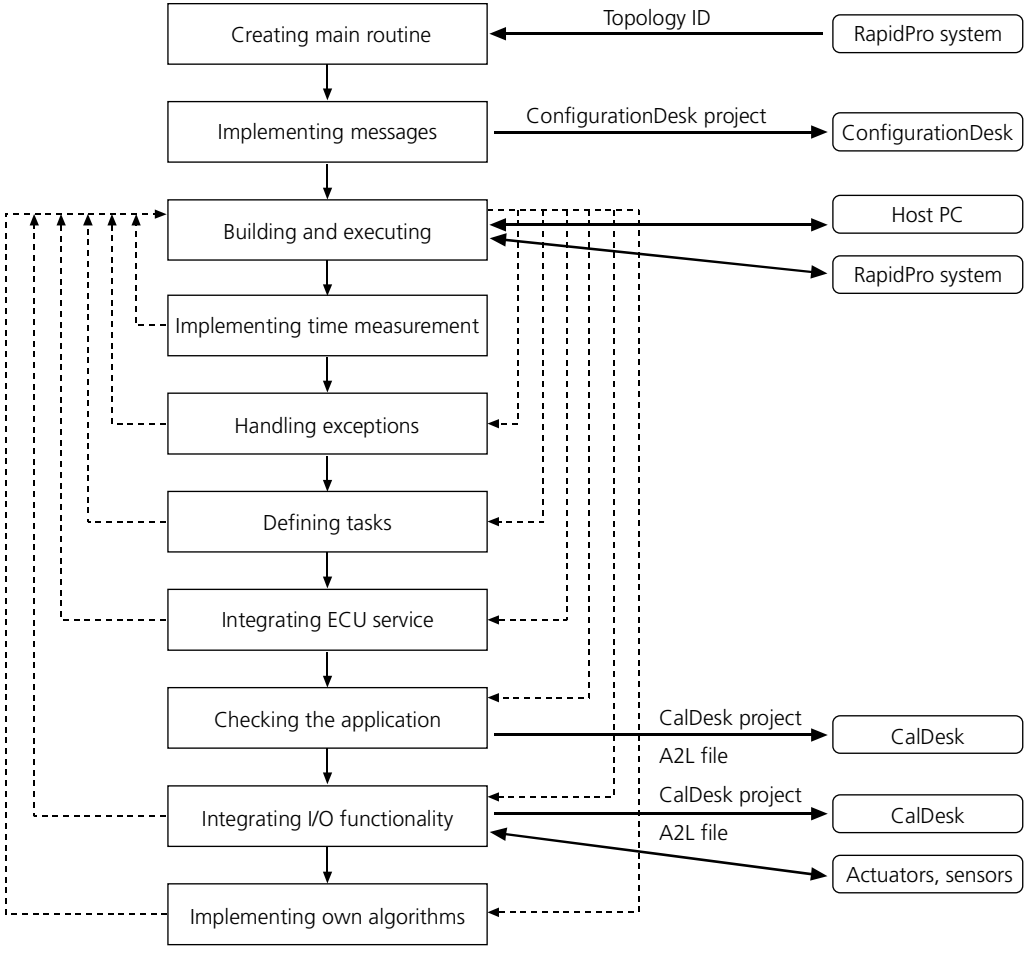
For further information, refer to *Implementing I/O Access* on page 151 and *Implementing Bus Protocols* on page 193.

**11** Implementing custom code

If you followed these work steps, you can now develop your own application code.

# Flowchart of the Workflow

The flowchart shows the described workflow. Some steps must be repeated when you have changed your source code, for example, building the application.



If you have learned how to create a basic application for a RapidPro system, you will turn off the workflow and implement the application code in a more parallel way according to the many dependencies between the work steps.

# Getting Hardware Details for Implementation

**Objective**

When implementing an application, you have to specify parameters of hardware-related RTLib functions. These parameters must correspond to your specific RapidPro hardware.

For easy access to the hardware details, the following overview tells you where to find detailed instructions in other documents.

**Required details**

You have to get the following details of your RapidPro hardware:

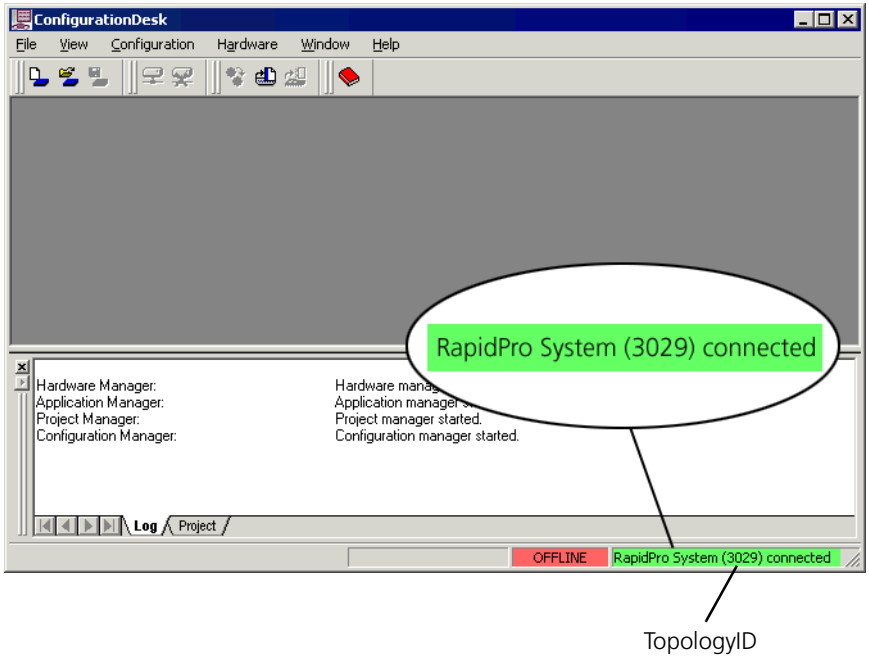
| Detail | Purpose | Access via ... |
|---|---|---|
| TopologyID | To get the TopologyID parameter used in the `ds1603_topology_id_check` function which checks the compatibility of your RapidPro hardware with your application. | Status bar of ConfigurationDesk, see *Getting the TopologyID* on page 38 |
| I/O mapping | To specify parameters in hardware-related RTLib functions for I/O drivers. | Pinout information file, which you can generate using ConfigurationDesk, see *Getting I/O mapping information* on page 39. |
| Installation of available CAN transceivers | To specify parameters in hardware-related RTLib functions for CAN communication. | Hardware Navigator in ConfigurationDesk, see *Getting information about installed CAN transceivers* on page 40 |

If you are not familiar with the specifics and details of the RapidPro hardware, you should read the information given in:

- *RapidPro Hardware Information Required for Implementation* on page 27
- *Effects of Changing Components of a RapidPro System* on page 29

**Getting the TopologyID**

You can get the TopologyID via ConfigurationDesk. It is shown in ConfigurationDesk's status bar if your RapidPro system is connected to the host PC (see below).




For the first steps in ConfigurationDesk, refer to *How to Load RapidPro Hardware Data to ConfigurationDesk* in the *RapidPro System – Getting Started* document.

### Getting I/O mapping information

You can get the necessary I/O mapping from a pinout information file that you can generate using ConfigurationDesk. The output is a list in CSV or XLS format which can be opened with Microsoft Excel(TM).

The list contains the available channels of your RapidPro system. It also shows the mapping of each microcontroller signal (channel) to the corresponding I/O channel of the SC or PS module and its I/O pin. For an example see below:

| F-Connector, Pin | MC Channel | Signal | Signal Description | Channel | Channel Name | Channel Description | Module | Layer | Unit Type | Slot |
|---|---|---|---|---|---|---|---|---|---|---|
| F1, 008 | | FB_OUTA | Driver output A | 2 | Full Bridge Out Ch02 | Full Bridge Output, Ch02 | PS-FBD 2/1 | 3 | Power | 1 |
| | TPU A, Ch27 | FB_CTRL | FB control | 2 | | Full Bridge Output, Ch02 | PS-FBD 2/1 | 3 | Power | 1 |
| | TPU A, Ch28 | FB_DIR | FB direction | 2 | | Full Bridge Output, Ch02 | PS-FBD 2/1 | 3 | Power | 1 |
| | Analog Input, Ch02 | FB_U(I) | Current measurement | 2 | | Full Bridge Output, Ch02 | PS-FBD 2/1 | 3 | Power | 1 |
| | | | | | | | | | | |
| F1, 005 | | HS_OUT | Driver output | 1 | High Side Out Ch01 | High Side Driver, Ch01 | PS-HSD 6/1 | 3 | Power | 3 |
| | TPU B, Ch01 | HS_CTRL | HS control | 1 | | High Side Driver, Ch01 | PS-HSD 6/1 | 3 | Power | 3 |
| | Analog Input, Ch05 | HS_U(I) | Current measurement | 1 | | High Side Driver, Ch01 | PS-HSD 6/1 | 3 | Power | 3 |
| F1, 017 | | HS_OUT | Driver output | 2 | High Side Out Ch02 | High Side Driver, Ch02 | PS-HSD 6/1 | 3 | Power | 3 |
| | TPU B, Ch02 | HS_CTRL | HS control | 2 | | High Side Driver, Ch02 | PS-HSD 6/1 | 3 | Power | 3 |
| | Analog Input, Ch06 | HS_U(I) | Current measurement | 2 | | High Side Driver, Ch02 | PS-HSD 6/1 | 3 | Power | 3 |
| F1, 004 | | HS_OUT | Driver output | 3 | High Side Out Ch03 | High Side Driver, Ch03 | PS-HSD 6/1 | 3 | Power | 3 |
| | TPU B, Ch03 | HS_CTRL | HS control | 3 | | High Side Driver, Ch03 | PS-HSD 6/1 | 3 | Power | 3 |
| F1, 016 | | HS_OUT | Driver output | 4 | High Side Out Ch04 | High Side Driver, Ch04 | PS-HSD 6/1 | 3 | Power | 3 |
| | TPU B, Ch04 | HS_CTRL | HS control | 4 | | High Side Driver, Ch04 | PS-HSD 6/1 | 3 | Power | 3 |
| F1, 003 | | HS_OUT | Driver output | 5 | High Side Out Ch05 | High Side Driver, Ch05 | PS-HSD 6/1 | 3 | Power | 3 |
| | TPU B, Ch21 | HS_CTRL | HS control | 5 | | High Side Driver, Ch05 | PS-HSD 6/1 | 3 | Power | 3 |
| F1, 015 | | HS_OUT | Driver output | 6 | High Side Out Ch06 | High Side Driver, Ch06 | PS-HSD 6/1 | 3 | Power | 3 |
| | TPU B, Ch22 | HS_CTRL | HS control | 6 | | High Side Driver, Ch06 | PS-HSD 6/1 | 3 | Power | 3 |

I/O pins

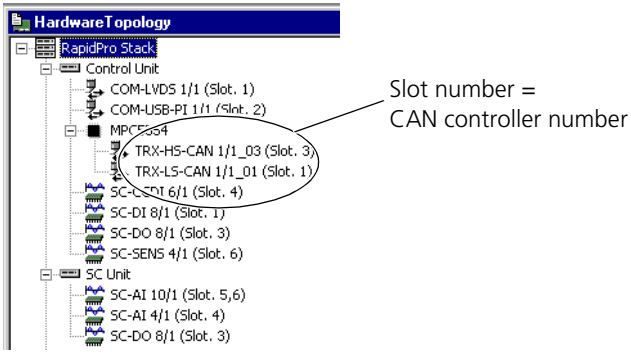
Channels on MPC5554

I/O channel of RapidPro module

For instructions on getting the pinout, refer to *How to Export Pinout Information* in the *RapidPro System – Getting Started* document.

**Getting information about installed CAN transceivers**

You can get the necessary information via the Hardware Navigator in ConfigurationDesk. The navigator displays the available CAN transceivers, their types, and the slots they are installed in. The slot number corresponds to a CAN controller on the MPC5554, which you can configure via RTLib for CAN communication.




**Hardware Navigator**

With the hardware equipment shown above which serves as an example, you can use only controllers (channels) 1 and 3 of the MPC5554 for CAN communication.

The pinout information file does not provide information about the CAN transceivers installed in the RapidPro system.

For instructions on accessing the Hardware Navigator, refer to *How to Load RapidPro Hardware Data to ConfigurationDesk* in the *RapidPro System – Getting Started* document.

# Using the Demo Application

**Objective**

The dSPACE software provides a demo application to show the prototyping capabilities of a RapidPro system using the MPC5554 microcontroller.

**Where to go from here**

Information in this section


| |
|---|
| ***Overview of the Demo Application*** **on page 42** |
| ***Working with the Ready-to-Use Demo Application*** **on page 44** |
| ***Working with the Demo Sources*** **on page 51** |

# Overview of the Demo Application

**Using the demo application**

There are different ways to use the demo application:

- Ready-to-use demo
  Without any experience in programming with the RapidPro system, you can use prepared binaries. If you start with the demo application without I/O access, there are not even any preconditions to be fulfilled by your installed RapidPro system. You can immediately start experimenting with the demo application, using the CalDesk project provided. For further information, refer to *Basics on the Prepared Binaries* on page 44 and *How to Start A Ready-to-Use Demo Application* on page 45.
- Source files for customizing the demo application
  The C source files can be used to modify and build the demo application according to your individual RapidPro system. In the most simple case, you must only specify the TopologyID of your hardware. If you then build the demo application without I/O access, there are no preconditions to be fulfilled by your installed RapidPro system. With some modifications to the prepared CalDesk project, you can start experimenting with the application that was built. For further information, refer to *How to Build and Execute the Demo Application* on page 62 and *How to Prepare Calibration and Measurement* on page 64.

  If your RapidPro system fullfills the preconditions for the I/O access implemented in the demo application, you can build the demo application including I/O access. For information, refer to *Preconditions for Executing the Demo Application* on page 59.
- Source files of the demo used as templates for your own applications

  The source files of the demo can also be used as templates for your own application. The implemented subroutines cover most aspects of implementing prototyping applications. You have only to modify the hardware-dependent configurations.

**Where to find the components of the demo application**

The components of the demo application can be found on `%DSPACE_ROOT%/Demos/DS1603/GettingStarted`. The following folders are installed on your host PC:

| Folder | Contents |
|---|---|
| `.\HandCode\Bin` | Binaries:<br>• Ready-to-use `.ppc` files<br>• Ready-to-use `.mot` and `.a2l` files<br>• Batch files for downloading a ready-to-use object file |
| `.\HandCode` | Source files:<br>• C source files and header files of the demo application<br>• Batch files for building and downloading the demo application with or without I/O access |
| `.\HandCode\CalDeskProject` | CalDesk project:<br>• ZIP file which contains a CalDesk project, immediately usable with the prepared binaries. |

# Working with the Ready-to-Use Demo Application

**Where to go from here**

Information in this section


| |
|---|
| ***Basics on the Prepared Binaries*** **on page 44** |
| ***How to Start A Ready-to-Use Demo Application*** **on page 45** |


## Basics on the Prepared Binaries

**Ready-to-use files**

You can immediately start with the demo application using the ready-to-use `.ppc` files. They are provided by dSPACE for the following use scenarios:

- Flash application without I/O access
- RAM application without I/O access
- Flash application with I/O access
- RAM application with I/O access

When using the demo application with enabled I/O access, you must note some preconditions regarding the required hardware, refer to *Preconditions for Executing the Demo Application* on page 59.

Because of these preconditions, it is recommended to use one of the applications from the `I/O_Disabled` folder.

For each use scenario there are also prepared `.mot` and `.a2l` files, to be used with CalDesk. For further information, refer to *How to Start A Ready-to-Use Demo Application* on page 45.

# How to Start A Ready-to-Use Demo Application

**Objective**

To start a ready-to-use demo application, you must open the prepared CalDesk demo project. This gives you access to the variables of the flash demo application without I/O access.

This How to is written for users who want to take a fast and easy 'first look' at the demo application with CalDesk. To avoid preconditions regarding the components of the RapidPro hardware, the prepared flash demo application without I/O access is used. However, as this has no I/O access, some instruments in CalDesk cannot display the assigned signals.

**Preconditions**

- There are no preconditions for the equipment of your RapidPro hardware.
- CalDesk must be closed.
- ConfigurationDesk must be in offline mode or must be closed.

**Method**

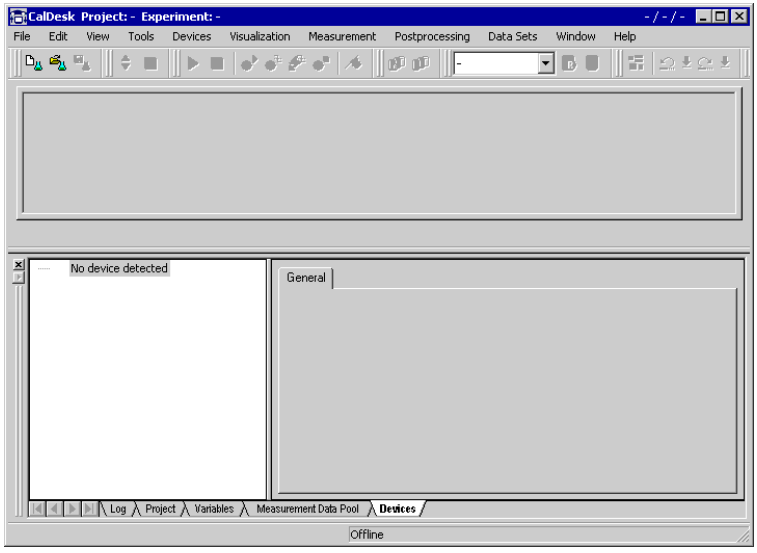
**To start a ready-to-use demo application**

1 In the Demo Pool on `%DSPACE_Root%\Demos\DS1603\GettingStarted\HandCode\Bin\IO_Disabled` start `Download_RTFrameDS1603_FLASH.bat`.

The demo flash application without I/O access is downloaded to your RapidPro hardware.

2 From the Windows **Start** menu, select **Programs – dSPACE Tools – CalDesk**, and click **dSPACE CalDesk.**

CalDesk opens.

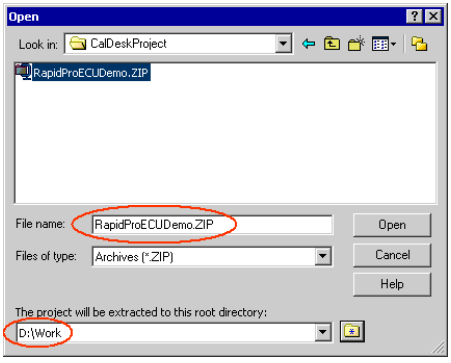



3 From the **File** menu, select Open **Project + Experiment from Backup**.

An **Open** dialog is displayed.

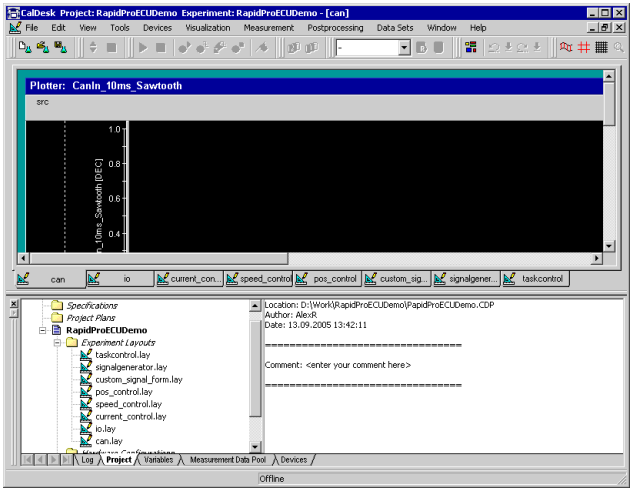
4 In the Open dialog, navigate to `%DSPACE_Root%\Demos\DS1603\GettingStarted\HandCode\CalDeskProject` and select `RapidProECUDemo.ZIP`.

Specify the root folder to extract the project and experiment to.




5 Click **Open**.

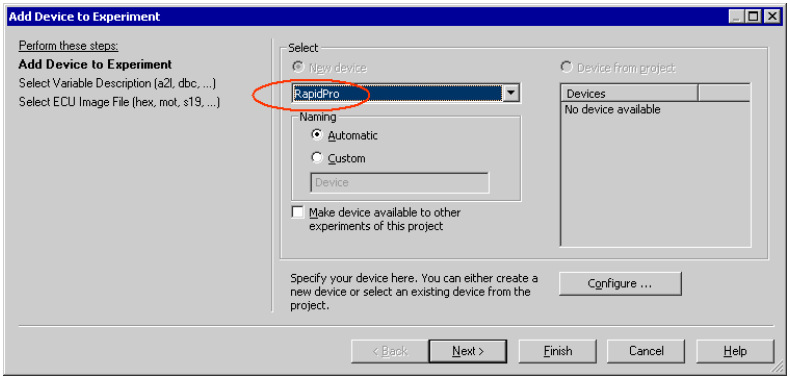
The RapidProECUDemo project is extracted to the selected root folder and opened in CalDesk.




6 From the **Devices** menu, select **Add Device**.

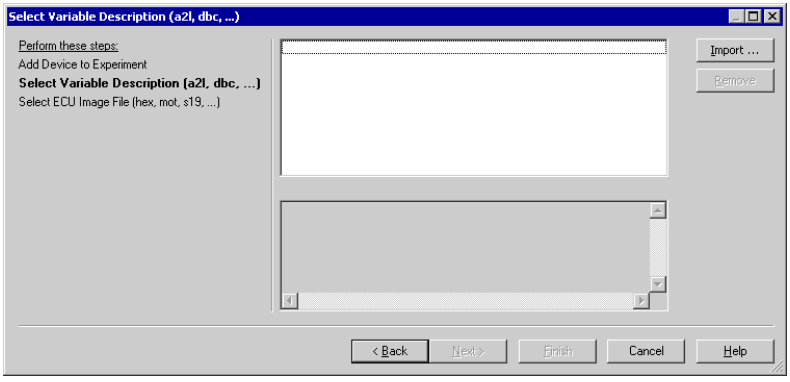
The **Add Device to Experiment** dialog opens.

7 From the list of device types, select the **RapidPro** device.




8 Click **Next**.

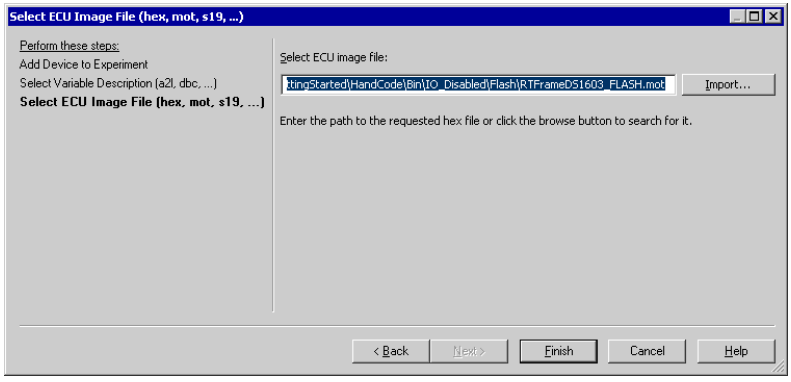
The **Select Variable Description** dialog opens.




9 Click **Import** and select `RTFrameDS1603_FLASH.a2l` from the following folder:
`%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode\Bin\IODisabled`.

10 Click **Next**.

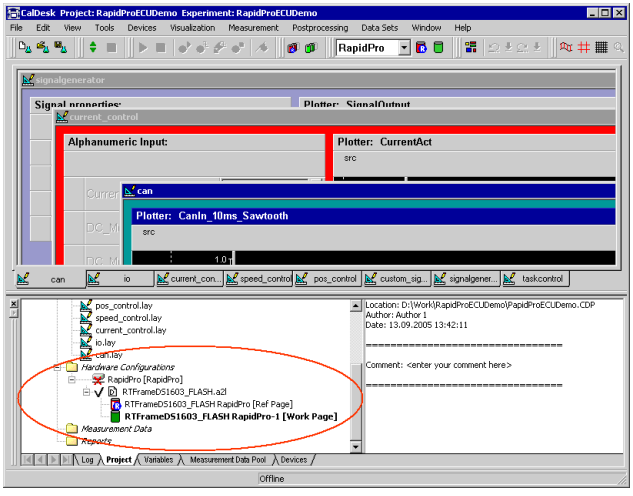
**11** The **Select ECU Image File** dialog opens.




The appropriate ECU Image file is automatically selected.

**12** Click **Finish**.

The RapidPro device and the corresponding variable description and data sets are added to the RapidProECUDemo experiment.




**13** Now you have to establish the connections between the variables and the instruments in the layouts. The fastest way to do this is to save, close, and reopen the experiment:

- From the `File` menu, select **Save Project + Experiment**.

- From the `File` menu, select **Close Project + Experiment**.
- From the `File` menu, select **Open Project + Experiment**.

**Result** You have opened a prepared CalDesk project and added a RapidPro device with access to the flash demo application running on the RapidPro hardware.

**Next steps** For more information on working with experiments in CalDesk and cross-references to the CalDesk documentation, refer to How to Prepare Calibration and Measurement.

# Working with the Demo Sources

**Where to go from here**

Information in this section


| |
|---|
| ***Basics on the Demo Source Files*** **on page 51** |
| ***Overview of Implemented Subroutines and Their Configurations*** **on page 53** |
| ***Preconditions for Executing the Demo Application*** **on page 59** |
| ***How to Build and Execute the Demo Application*** **on page 62** |
| ***How to Prepare Calibration and Measurement*** **on page 64** |


## Basics on the Demo Source Files

**General information**

The `HandCode` folder contains the specific source and header files which were used to build the binaries of the demo application. All the functions are written in the C programming language. You can open the files to see how the features of the demo application have been implemented. Some files contain handcoded source code, others were generated from a TargetLink model using TargetLink as the code generator. For further information on the source files, see *Demo source files*.

**Disabling I/O access**

The C source files contain the preprocessor directive "`#ifndef DISABLE_IO_ACCESS`", which can be interpreted by the compiler to exclude the implemented I/O access in the built application.

For further information on building and executing the demo application, refer to *How to Build and Execute the Demo Application* on page 62.

You should build and execute the application with I/O access only if you are sure that the I/O mapping of your RapidPro system matches the parameterization of hardware-related RTLib functions in the demo application. For further information, refer to *Preconditions for Executing the Demo Application* on page 59.

**Experimenting with a built application**

For information on using the built application in a CalDesk experiment, refer to *How to Prepare Calibration and Measurement* on page 64.

**Demo source files**

The demo application comes with several C source files and header files relating to functional units in your application:

- `RTFrameDS1603`

  Contains the main routine of the demo application and declarations for global variables and functions.

- `DsaRapidProIO`

  Contains all subroutines created for I/O access and CAN support. Some subroutines are called in the main routine, others in the ECU code.

- `ECUCode`

  Contains user code, for example, the algorithms implemented on the ECU. The ECU code of the demo application was generated from a controller model using TargetLink.

- `ECUCode_CalParams`

  While the information in `ECUCode.c` is sufficient for measurement purposes, you must use `ECUCode_CalParams` to provide the calibratable parameters to CalDesk. `ECUCode_CalParams` is also generated by TargetLink.

# Overview of Implemented Subroutines and Their Configurations

**Objective**

The source files provide several prepared subroutines, which are listed below.

### RTFrameDS1603

This source file, the real-time frame, contains all the basic instructions required for a real-time application:

| Instructions For ... | Description |
|---|---|
| Initialization | To initialize hardware and software (RTLib1603, RTK1603). |
| Background service | To set up communication with the host PC. |
| Message handling | To log and display custom messages. |
| Exception handling | To implement defined reactions to exceptions. |
| Time measurement | To implement timing functions, for example, to calculate the execution time. |
| Task handling | To create tasks including interrupt binding and overrun handling:<br>• TaskExt: external triggered task<br>• TaskA, TaskB, TaskC, TaskD: timer tasks<br>• WatchdogTask: watchdog task |
| dSPACE Calibration and Bypassing Service | To set up the dSPACE Calibration and Bypassing Service. You need this service, for example, when you want to experiment with CalDesk. The basic framework is already included in the background service. |
| I/O access | To initialize and start I/O access in a task. For further information, refer to *Implementing I/O Access* on page 151. |
| CAN support | To initialize and trigger CAN messages in a task. For further information, refer to *Implementing Bus Protocols* on page 193. |

### DsaRapidProIO

**A/D conversion**

The following subroutines are implemented and configured for A/D conversion:

| A/D Conversion | Description |
|---|---|
| Configuration function | |
| `DsaAdcInit` | The following channels are configured for A/D conversion:<br>• Analog Input, channel 29 (specified for ADC 2, Queue 1)<br>• Analog Input, channel 30 (specified for ADC 2, Queue 1)<br>• Analog Input, channel 31 (specified for ADC 1, Queue 2)<br>• Analog Input, channel 32 (specified for ADC 1, Queue 2)<br>• Analog Input, channel 33 (specified for ADC 2, Queue 2)<br>• Analog Input, channel 34 (specified for ADC 2, Queue 2)<br>• Analog Input, channel 35 (specified for ADC 1, Queue 1)<br>• Analog Input, channel 36 (specified for ADC 1, Queue 1)<br>All channels are configured to use the same sample rate. |
| Run-time functions | |
| `DsaAdcStartQueue1`<br>…<br>`DsaAdcStartQueue4` | The above combinations of A/D converter and queue specification are started in a task:<br>• `DsaAdcStartQueue1` starts queue 1 specified for A/D converter 1<br>• `DsaAdcStartQueue2` starts queue 1 specified for A/D converter 2<br>• `DsaAdcStartQueue3` starts queue 2 specified for A/D converter 1<br>• `DsaAdcStartQueue4` starts queue 2 specified for A/D converter 2 |
| `DsaGetADC1`<br>…<br>`DsaGetADC8` | The subroutines to read the A/D conversions are used in the ECU code:<br>• `DsaGetADC1` reads the value from A/D channel 29<br>• `DsaGetADC2` reads the value from A/D channel 30<br>• `DsaGetADC3` reads the value from A/D channel 31<br>• `DsaGetADC4` reads the value from A/D channel 32<br>• `DsaGetADC5` reads the value from A/D channel 33<br>• `DsaGetADC6` reads the value from A/D channel 34<br>• `DsaGetADC7` reads the value from A/D channel 35<br>• `DsaGetADC8` reads the value from A/D channel 36 |

**Bit I/O on I/O PLD**

The following subroutines are implemented and configured for bit I/O using the I/O PLD:

| Bit I/O (I/O PLD) | Description |
|---|---|
| Configuration function | |
| `DsaDigIOInit` | The following channels are configured as digital outputs of the I/O PLD:<br>• Channel 1 of the I/O PLD (BIT_IO Ch01)<br>• Channel 2 of the I/O PLD (BIT_IO Ch02)<br>Both channels are configured as outputs with the initial value 0.<br>Initialization includes starting the channels. |
| Run-time functions | |
| `DsaSetBitOut1`<br>`DsaSetBitOut2` | The subroutines to write 0 or 1 to a specified I/O channel are used in the ECU code. |

**Digital input on eTPU**

The following subroutines are implemented and configured for digital input on eTPU:

| Digital Input (eTPU) | Description |
|---|---|
| Configuration function | |
| `DsaTPUIOInit` | The following channels are configured as digital inputs:<br>• Channel 1 of eTPU A (eTPU_A Ch1), generating interrupts on both edges<br>• Channel 2 of eTPU A (eTPU_A Ch2), generating no interrupts<br>Both channels are configured to use the eTPU time base 1 with a prescaling factor of 16.<br>Initialization includes starting the channels. |
| Run-time functions | |
| `DsaGetBitIn1`<br>`DsaGetBitIn2` | The subroutines to read the digital values of the specified channels are used in the ECU code. |

**1-phase PWM signal generation**

The following subroutines are implemented and configured for 1-phase PWM signal generation (PWM out):

| 1-Phase PWM Signal Generation | Description |
|---|---|
| Configuration function | |
| `DsaTPUIOInit` | The following channels are configured as PWM outputs:<br>• Channel 13 of eTPU B (eTPU_B Ch13)<br>• Channel 14 of eTPU B (eTPU_B Ch14)<br>Both channels are configured to use eTPU time base 1 with a prescaling factor of 16. The polarity is set to active high.<br>Initialization includes starting the channels. The initial duty cycle is set to 0 and the initial frequency is set to 1000 Hz. |
| Run-time functions | |
| `DsaSetVecPwmOut1`<br>`DsaSetPwmOut2` | The subroutines to update the values for the duty cycle and the frequency are used in the main routine and in the ECU code.<br>• `DsaSetVecPwmOut1`<br>The duty cycle and the frequency values are stored in a vector and can be set according to a specified index parameter. The subroutine contains the `Ds1603TpuPwmOut_write` function for updating the settings.<br>• `DsaSetPwmOut2`<br>You can use this subroutine to set the duty cycle to the specified value. It contains the `Ds1603TpuPwmOut_write` function for updating the setting. |

**PWM signal measurement**

The following subroutines are implemented and configured for PWM signal measurement (PWM in):

| PWM Signal Measurement | Description |
|---|---|
| Configuration function | |
| `DsaTPUIOInit` | The following channels are configured as PWM inputs:<br>• Channel 3 of eTPU A (eTPU_A Ch3)<br>• Channel 4 of eTPU A (eTPU_A Ch4)<br>Both channels are configured to use TPU time base 1 with a prescaling factor of 16. The polarity is set to active high.<br>Initialization includes starting the channels. |

| PWM Signal Measurement | Description |
|---|---|
| Run-time functions | |
| `DsaGetVecPwmIn1`<br>`DsaGetPwmIn2` | The subroutines to get the duty cycle and the frequency values of the specified channels are used in the ECU code.<br>• `DsaGetVecPwmIn1`<br>This subroutine reads from the PWM inputs and stores the values of the duty cycle and the frequency in a vector according to the specified index parameter.<br>• `DsaGetPwmIn2`<br>You can use this subroutine to read from the PWM inputs and store the duty cycle in the specified parameter. |

**CAN support** The following subroutines are implemented and configured for CAN support:

| CAN Handling | Description |
|---|---|
| Configuration function | |
| `DsaCANInit` | Three CAN channels are created and specified with a baud rate of 250 kbit/s.<br>• CAN channel 1:<br>Configuration of a CAN receive (RX) message in STD format (identifier "0x10")<br>• CAN channel 2:<br>Configuration of a CAN transmit (TX) message in STD format (identifier "0x10", 8 byte data length, variable for the message data)<br>• CAN channel 3:<br>Configuration of a CAN receive (RX) message in STD format (identifier "0x10") |

| CAN Handling | Description |
|---|---|
| Run-time functions | |
| `DsaCanTriggerTxCh2` | The CAN transmit channel is triggered to transmit a message. This subroutine is called in timer task B. |
| `DsaGetCanIn10msSawtooth` | To receive a specific signal on CAN channel 1. |
| `DsaGetCanIn20msSawtooth` | To receive a specific signal on CAN channel 3. |
| `DsaGetCanIn10msSine1` | To receive a specific signal on CAN channel 1. |
| `DsaGetCanIn10msSine2` | To receive a specific signal on CAN channel 1. |
| `DsaGetCanIn20msSine1` | To receive a specific signal on CAN channel 3. |
| `DsaGetCanIn20msSine2` | To receive a specific signal on CAN channel 3. |
| `DsaSetCanOutSawtooth`<br>`DsaSetCanOutSine1`<br>`DsaSetCanOutSine2` | To specify the data to be transmitted on CAN channel 2 |

# Preconditions for Executing the Demo Application

**Objective**

To execute the demo application, there are some preconditions regarding the components of your RapidPro system.

**Preconditions**

The preconditions depend on the use scenario:

- Using **source files** without I/O access

  The TopologyID of your RapidPro system must be identical to the TopologyID specified in the initialization section of the demo application. Refer to *Details on the TopologyID* on page 60.

- Using **source files** with I/O access

  To execute source files with I/O access:

  - The connected hardware must comply with TopologyID 8984. For details, refer to *Matching TopologyID 8984* on page 60.
  - The TopologyID specified in the initialization section of the demo application must be 8984. Refer to *Details on the TopologyID* on page 60.

- Using prepared binaries (**ready-to-use demo**) without I/O access

  To execute the prepared binaries without I/O access, there are no preconditions regarding the components of your RapidPro hardware.

- Using prepared binaries (**ready-to-use demo**) with I/O access

  To execute prepared binaries with I/O access, the connected hardware must comply with TopologyID 8984. For details, refer to *Matching TopologyID 8984* on page 60.

**Details on the TopologyID**

TopologyID 0 is the default setting used in the source files of your demo application. For information on accessing the TopologyID of your RapidPro system, refer to *Getting the TopologyID* on page 38.

If the TopologyID of your connected RapidPro system differs from TopologyID 8984, you can execute only demo applications without I/O access. You also have to change the TopologyID in the declaration of the variable that holds the TopologyID of your RapidPro system:

```
UInt32 TopoRequired=8984;
```

This declaration is included in the `RTFrameDS1603.c` file of your demo application (located in `%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode\`)

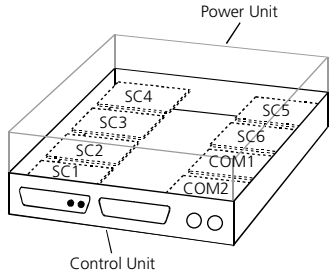
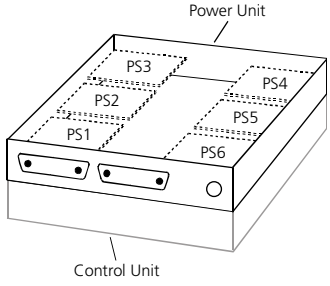
The TopologyID is checked by the `ds1603_topology_id_check` function, which checks the consistency of the connected hardware topology and the ID specified in the source code.

**Matching TopologyID 8984**

A RapidPro system must contain the following hardware components to comply with TopologyID 8984:

- RapidPro Control Unit with:
  - SC-AI 10/1 (DS1633)
  - SC-DI 8/1 (DS1642)
  - SC-DO 8/1 (DS1646)
  - COM-USB-PI 1/1 (DS1607)
- RapidPro Power Unit
  - PS-LSD 6/1 (DS1662)

The table below shows the unit assembly, the modules, and their installation slots required to match TopologyID 8984:

| Units and Their Slots | Slot | Module |
|---|---|---|
|  | SC1 | SC-DI 8/1 (DS1642) |
| | SC2 | – |
| | SC3 | – |
| | SC4 | SC-DO 8/1 (DS1646) |
| | SC5 | SC-AI 10/1 (DS1633) 1) |
| | SC6 | SC-AI 10/1 (DS1633) 1) |
| | COM1 | COM-USB-PI 1/1 (DS1607) |
| | COM2 | – |
|  | PS1 | – |
| | PS2 | – |
| | PS3 | PS-LSD 6/1 (DS1662) |
| | PS4 | – |
| | PS5 | – |
| | PS6 | – |

1) The SC-AI 10/1 has 10 channels and requires two adjacent slots.

- You should execute an application with I/O access only if you are sure that the I/O mapping of your RapidPro system matches the parameterization of hardware-related RTLib functions in the demo application.
- If you want to use the CAN support of the demo application, the MPC5554 microcontroller module must be equipped with CAN transceiver modules. For further information, refer to *Installing of CAN transceivers* on page 28.

# How to Build and Execute the Demo Application

**Objective**

To execute the demo application on the RapidPro system, you need to build and download it to your hardware.

**Folder structure**

The demo application can be found on `%DSPACE_ROOT%/Demos/DS1603/GettingStarted`. For details on its folder structure, refer to *Overview of the Demo Application* on page 42.

**Customizing source files**

You can customize the source files of the demo (or at least parts of them) so that they can serve as a starting point for your own real-time applications.

**Default download setting**

By default, the flash demo application is downloaded to the RapidPro hardware.

**Preconditions**

- To execute the demo application, some preconditions concerning the RapidPro hardware and the TopologyID must be fulfilled. For details, refer to *Preconditions for Executing the Demo Application* on page 59.
- During download, the RapidPro device in CalDesk must be in a disconnected state, or CalDesk must be closed.
- During download, ConfigurationDesk must be in offline mode or closed.

**Method**

**To build and execute the demo application**

➤ Build the demo by calling either `Build_RTFrameDS1603_IO_disabled.bat` or `Build_RTFrameDS1603_IO_enabled.bat`, which are both located on `%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode`.

You should use the demo application without I/O access (`..._IO_disabled.bat`), if you are not sure if the I/O mapping of your RapidPro system matches the parameterization in the demo application.

**Result**

The source files are compiled and linked and the generated files are stored on `%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode\.` The application is downloaded to the flash memory of your RapidPro system. It is named `RTFrameDS1603_FLASH` and starts running on the MPC5554 microcontroller immediately after download. Depending on the batch file you used, the application runs either with or without I/O access.

- If the TopologyID specified in the initialization section of your application is not identical to the TopologyID of your RapidPro system, your application starts briefly, then immediately stops running, and an error message is output during the TopologyID check.
- If you want to work with the demo application in CalDesk, you need an A2L (ASAM-MCD 2MC) file which contains the variable description of the demo application.

**Next step**

If you have downloaded the demo application to the RapidPro hardware and have an A2L file which matches your hardware, you can load the demo application in CalDesk. For instructions, refer to *How to Prepare Calibration and Measurement* on page 64.

# How to Prepare Calibration and Measurement

**Objective**

CalDesk provides access to the variables of the demo application. You can see information on variables (addresses, limits, etc.), change the values of parameters, perform measurements or recordings, and print reports of parameter data sets, for example.

The instructions in this How to are written for users who might be unfamiliar with CalDesk. Brief definitions of basic CalDesk terms are therefore given below.

**Basic terms in CalDesk**

The following list contains brief definitions of CalDesk terms that are used in the instructions that follow.

**Project** In CalDesk, a project manages different experiments that belong together, such as the tasks for calibrating a specific engine variant. It holds the experiments relating to these tasks, and documents relevant to the entire project.

**Experiment** An experiment is the basis for carrying out a specific task, for example, adjusting the values of an idle speed control. An experiment allows you to manage all the items relating to the task, such as: devices, layouts, variable descriptions, data sets.

**Device** In CalDesk, devices are used to access hardware components like electronic control units (ECUs). CalDesk provides the RapidPro device to access the Rapid Pro hardware.

**Variable description file** The variable description specifies the parameters and measurement variables that are available on a device. It also contains information on the device's memory segments. The variable description of an ECU RapidPro system is stored as an ASAM-MCD 2MC (A2L) file.

**ECU Image file** The ECU Image file usually contains the code of an ECU application and the data of its parameters. The ECU Image file of the demo application is stored as a `mot` file.

**Data set** A data set contains a complete set of parameters. The RapidPro device has two memory pages: a reference and a working page. The reference data set is assigned to the reference page, the working data set to the working page. The working data set must be activated if you want to change parameter values.

**Layout** Layouts are used for visualizing variables: To calibrate parameters, or measure and record data, the corresponding variables must be placed on a layout. In the layout, calibration and recording of these variables is done graphically with instruments

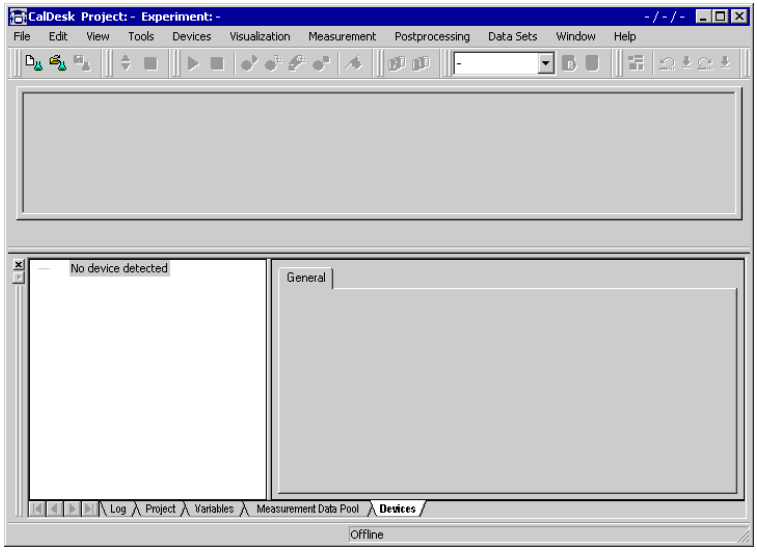
**Preconditions**

- The TopologyID of your RapidPro system must be identical to the TopologyID specified in the initialization section of your application.
- You must have created and downloaded the demo application as described in *How to Build and Execute the Demo Application* on page 62.
- You must have created an A2L file for the demo application, and stored it on `%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode\`. If you need an A2L file generator (editor), contact dSPACE for further information.

**Method** **To prepare calibration and measurement**

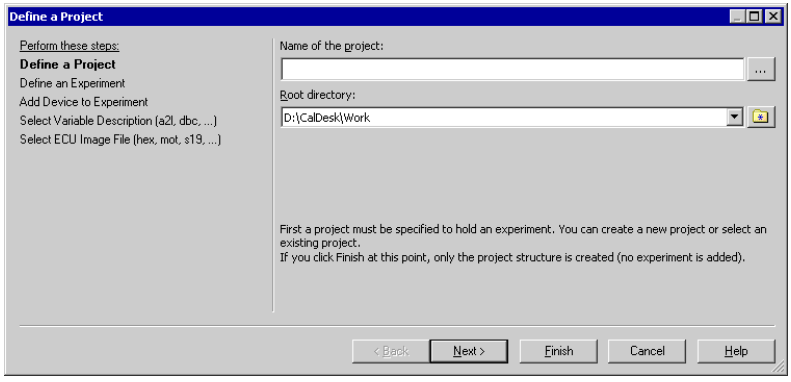
1 From the Windows **Start** menu, select **Programs – dSPACE Tools – CalDesk**, and click **dSPACE CalDesk**.

CalDesk opens.



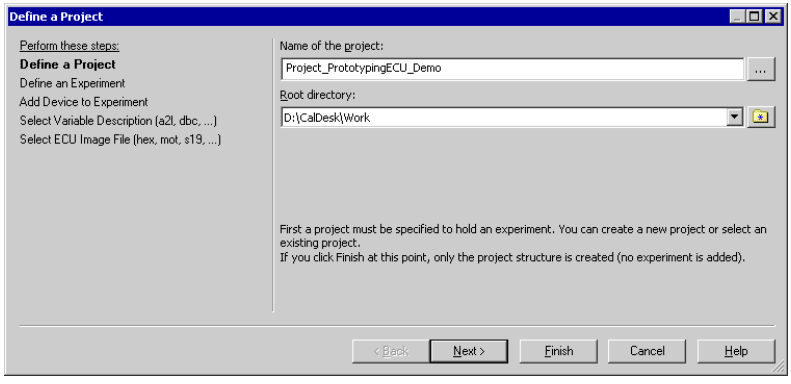
2 From the **File** menu, select **New Project + Experiment**.

The **Define a Project** dialog opens.




The **Define a Project** dialog is the first dialog of CalDesk's Project Wizard. The Project Wizard consists of a sequence of 5 dialogs, which let you create an experiment with access to the variables of the demo application.

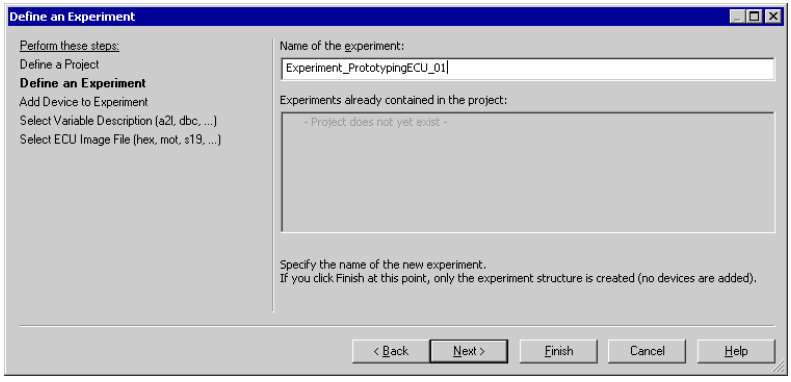
3 Enter a name for the project.




4 Click **Next.**

The **Define an Experiment** dialog opens.

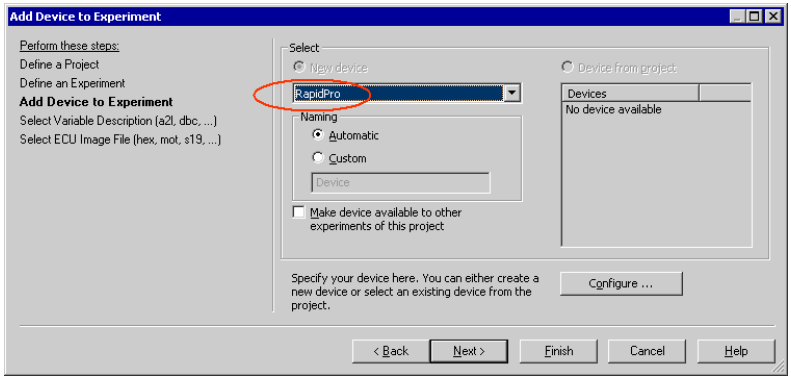
**5** Enter a name for the experiment.




**6** Click **Next.**

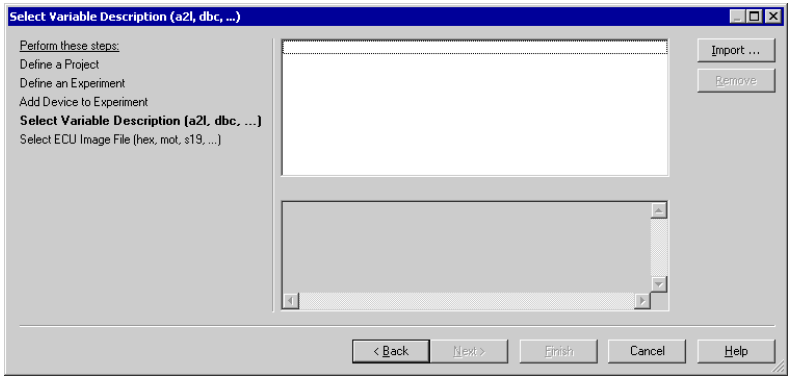
The **Add Device to Experiment** dialog opens.

**7** From the list of device types, select the **RapidPro** device.

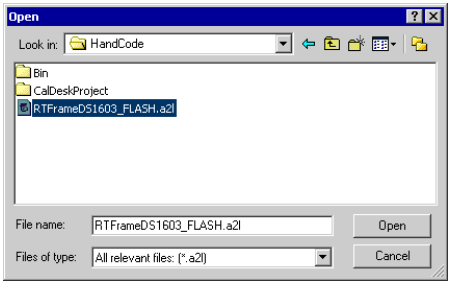


**8** Click **Next**.

The **Select Variable Description** dialog opens.

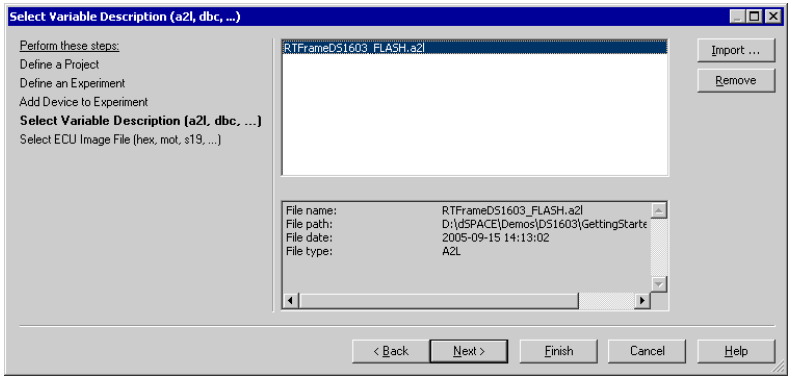



**9** Click **Import** and select the variable description (A2L) file you created for the demo application from the following folder: `%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode`.



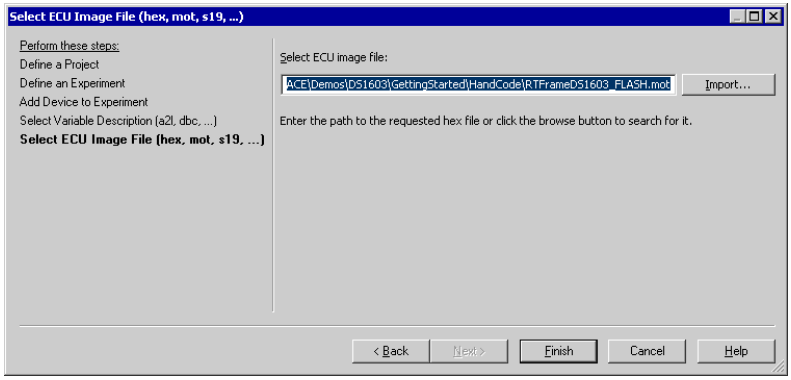
If the preconditions for executing the demo application are fulfilled, you can use one of the prepared A2L files, instead. Select an A2L file that meets your requirements (flash or RAM application with or without I/O access). For details on the preconditions, refer to *Preconditions for Executing the Demo Application* on page 59. For details on the prepared files, refer to *Working with the Ready-to-Use Demo Application* on page 44.

You have selected the variable description to be imported to the project.



**10** Click **Next.**

The **Select ECU Image File** dialog opens.



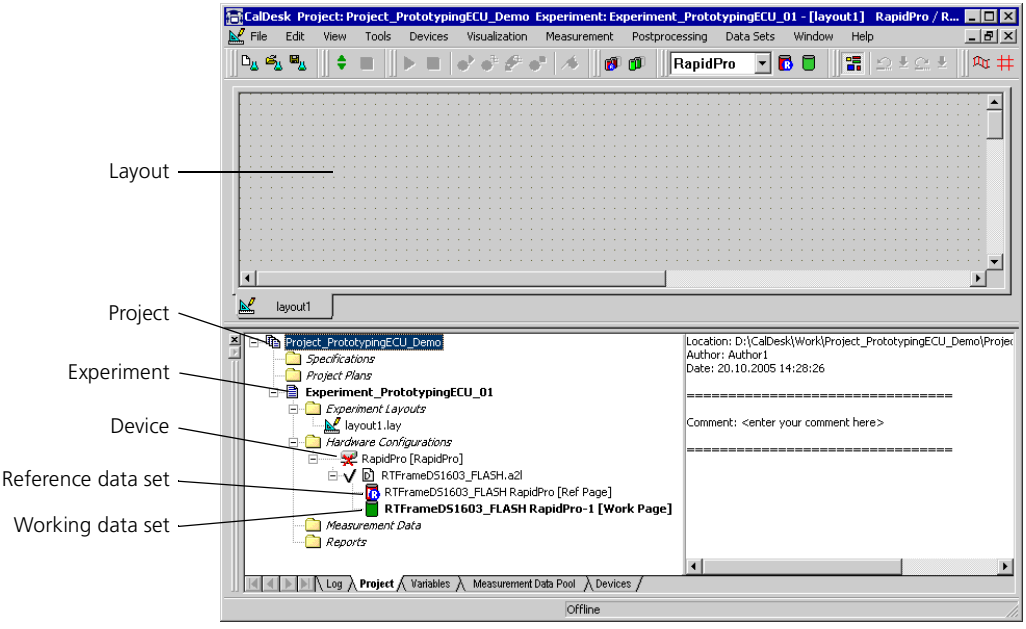

CalDesk identifies the names of the ECU Image files in the folder of the selected `.a2l` file and preselects the corresponding `.mot` file.

**11** Click **Finish** to close the Project Wizard.

**Result** You have created a new CalDesk project and experiment. The experiment contains the hardware configuration needed to access the variables of the real-time application running on the RapidPro system:

- A RapidPro device
- A variable description
- A reference data set and a working data set
- A CalDesk layout to visualize variables in instruments

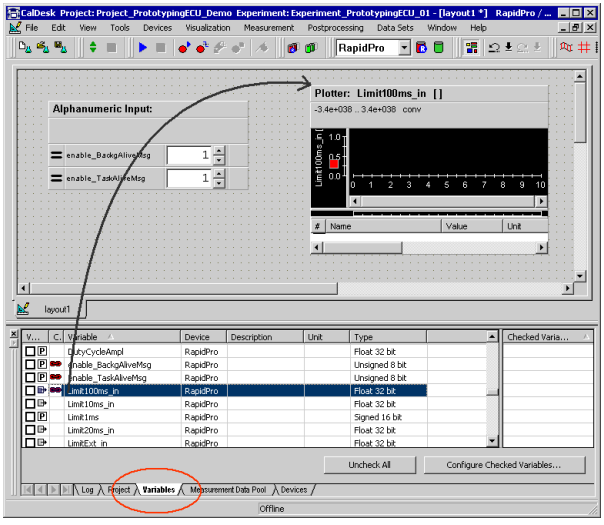
The following illustration shows what CalDesk should look like after you have finished the Project Wizard.



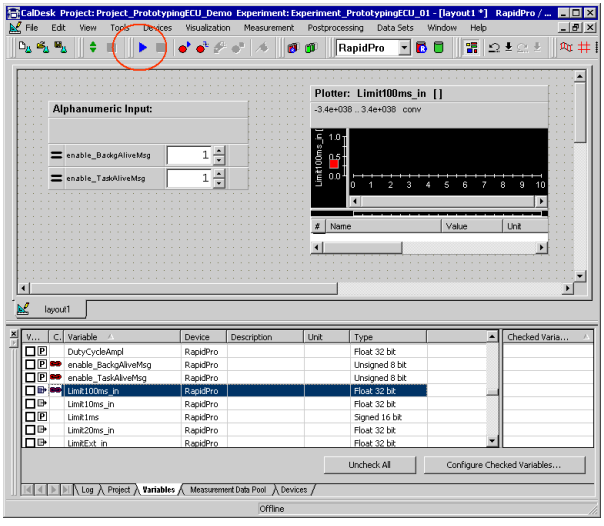

**Next steps**

You have now created the basis for accessing the real-time application on the RapidPro system. Some of the steps you can now take in CalDesk are shown below.

- From the variable browser, drag variables to the layout to visualize them in instruments.

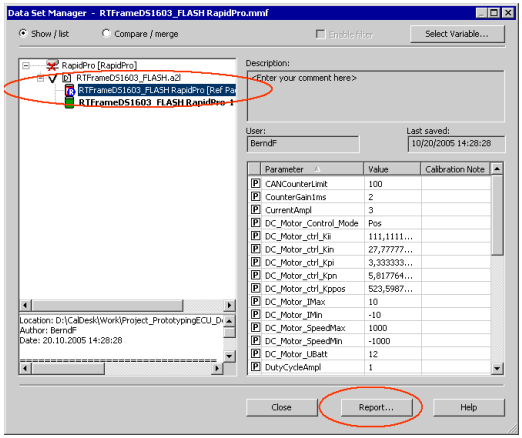



- Start a measurement.




CalDesk starts online calibration and measuring.

- From the **Data Sets** menu, select **Show/List** to open the Data Set Manager. Click a data set to list its values. Click Report to create an HTML or PDF report.




The following table guides you to more information on basic steps in working with CalDesk.

| Steps in CalDesk | Information in the CalDesk Calibration Guide |
|---|---|
| Select the variables (parameters and measurement variables) you want to work with. | *Basics of Viewing and Searching for Variables* |
| Visualize the variables in instruments on a layout. | *Basics of Visualizing Variables* |
| Change the values of parameters. | *Instruments, Data Sets and Calibration Tasks*<br>*How to Activate the Working or Reference Data Set* |
| Configure a measurement or recording. | *Measurement Configuration and Measurement Data Pool* |
| Start a measurement or recording. | *How to Start Measuring*<br>*How to Perform an Immediate Recording* |
| Analyze the results of a measurement or recording. | *Postprocessing Measured and Recorded Data in CalDesk* |
| Show or compare the values of parameters. | *How to Generate a Report* |

# Creating the Main Routine of Your Application

**Objective**

The main routine is the core of your real-time application. As usual for C applications, it is the main routine that is called by the application's start-up code and thus is the C entry point of the application.

**Where to go from here**

Information in this section

| |
|---|
| ***Structuring an Application*** **on page 76**<br>To make your application more transparent, you can split it into several parts. The structure of the demo application is described. |
| ***How to Create the Main Routine from Scratch*** **on page 78**<br>The main routine of the dSPACE demo application is rather complex. The first steps involved in creating it are described. |

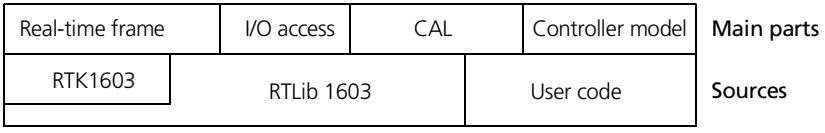
# Structuring an Application

**Objective**

To make your application more transparent, you can split it into several parts according to functional units.

**Recommended application structure**

The illustration below shows a possible way of structuring an application. The application consists of four main parts, each represented by a C file. These C files reference several sources of C code.




You can use the dSPACE demo application as a template to develop your own application, refer to the example at the end of this chapter.

**Real-time frame**

The real-time frame for the MPC5554 microcontroller module (DS1603) of the RapidPro system, or more precisely, its main routine is the core of the demo application. It is based on functions of the dSPACE Real-Time Kernel (RTK1603) and the DS1603 Real-Time Library (RTLib1603).

The RTK1603 provides functions for handling interrupts and scheduling tasks. It lets you implement the basic frame of a real-time operating system. For detailed information on the RTK1603, refer to *dSPACE Real-Time Kernel* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

The RTLib1603 provides functions for accessing the MPC5554 microcontroller module, for example, functions for initialization, time-stamping, and message and exception handling. For detailed information on the RTLib1603, refer to *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**I/O access**

Commonly, your RapidPro hardware is connected to sensors and actuators, which are driven by I/O accesses. The I/O access software module contains the required hardware-specific I/O functions which are called by the tasks.

The I/O functions are taken from the RTLib1603, which provides standard I/O functions (A/D conversion, bit I/O access, and timing I/O). For detailed information on the standard I/O functions, refer to *Standard I/O* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

**Controller model**

This software module contains the C code of your controller model. The C code can be handcoded or generated, for example, by using TargetLink, dSPACE's production code generation tool.

**CAL**

This software module defines the calibratable parameters (reference page and working page) of your application. You can access calibratable parameters of handwritten and generated C code via CalDesk, dSPACE's calibration tool.

The demo application comprises the following main parts and C files:

| Main Part | C File |
|---|---|
| Real-time frame | `RTFrameDS1603.c` |
| I/O access | `DsaRapidProIO.c` |
| Controller model | `ECUCode.c` |
| CAL | `ECUCode_CalParams.c` |

The C files of the demo application and their header files reside in `%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode`.

# How to Create the Main Routine from Scratch

**Objective**

The main routine of the dSPACE demo application is rather complex. To give you a better understanding of it, the first steps involved in creating it are described here.

**Implementation**

The instructions below describe the implementation of the following functionality:

- Initialization of the RapidPro system including a validity check on the connected RapidPro hardware
- Background service (communication between the RapidPro hardware and the host PC).

**Method**

**To create the main routine from scratch**

1. Create a new C file, for example, `MyRealTimeFrame.c`.
2. Type the following `include` in your source code:

   ```
   #include <Brtenv.h>
   ```

   Defines the basic real-time environment (part of the dSPACE software package)
3. Create a main routine.

   ```
   void main(void)
   {
   }
   ```
4. Type the declaration of the variable that holds the topology ID of your RapidPro system, for example:

   ```
   UInt32 TopoRequired=12345;
   ```

   The topology ID specified here is used later by the function that checks the topology ID (step 6).
5. Type the function that initializes the RapidPro system (hardware and software):

   ```
   init();
   ```
6. Type the function that checks the topology ID of the connected RapidPro hardware:

```
ds1603_topology_id_check(TopoRequired);
```

The topology ID specified in your C code (step 4) must match the topology ID that is read from the connected RapidPro hardware. Otherwise the validity check fails and the application stops.

**7** Type the function that activates all the output channels globally:

```
ds1603_output_enable();
```

At this point the part of the main routine that is concerned with initialization ends, and the part that is concerned with the background service begins.

**8** Type the function that implements the background service:

```
while( 1 )
{
    RTLIB_BACKGROUND_SERVICE();
}
```

The background service is called during the idle time of the application. It performs communication between the RapidPro system and the host PC.

**Result** You have created the main routine from scratch. The implemented functionality is still rudimentary, comprising the initialization of the RapidPro system, and the implementation of the background service.

The main routine can look like this:

```
#include <Brtenv.h>

void main(void)
{
    // Topology ID of your RapidPro system
    UInt32 TopoRequired=12345;

    // Initialize real-time library of the RapidPro system (RTLib1603)
    init();

    // Check whether the connected RapidPro hardware is appropriate
    ds1603_topology_id_check(TopoRequired);

    // enable output drivers
    ds1603_output_enable();

    while( 1 )
    {
        // call RTLib background service
        RTLIB_BACKGROUND_SERVICE();
    }
}
```

**Next steps** You can implement message handling to make the application talk to you, refer to *Implementing Messages in Your Application* on page 81.

# Implementing Messages in Your Application

**Objective**

To make your application give you information, for example, about important results or a progress status, you can use the message handling functions provided by the DS1603 Real-Time Library.

**Where to go from here**

Information in this section


| |
|---|
| ***Basic Information on Message Handling*** **on page 82** |
| ***How to Implement Messages in Your Application*** **on page 85** |
| ***How to Flush the Message Buffer*** **on page 88** |

# Basic Information on Message Handling

**Message output**

The DS1603 RTLib provides functions to generate messages that are displayed in the dSPACE software and stored in the dSPACE log file on your host PC. The messages are displayed in the Log Viewer of ConfigurationDesk and CalDesk. If you are working with both software programs, the messages are displayed in the program which is in online mode.

**Communication method**

The message is generated by the application running on the processor board and written to a message buffer. In the application's background loop, the message buffer is transferred to the connected host PC via USB link.

**Function module**

The message handling functions are provided by the DSMSG function module. The `DSMSG.h` header file contains all function declarations, global variables, and required data types.

**Message types**

The message module provides functions to generate error, warning, and information messages to be displayed by the dSPACE software and stored in the dSPACE log file. Additionally, you can specify a log message that is stored only in the dSPACE log file. You can use generic functions for implementing a freely configurable message during run time, for example, the `msg_set` function, or functions which are predefined for a specific message type, for example, the `msg_warning_set` function. The following four message types are defined:

| Message Type | Representation in the dSPACE Software |
|---|---|
| ERROR | The specified message text appears in:<br>• Log Viewer beginning with "ERROR"<br>• Log file |
| WARNING | The specified message text appears in:<br>• Log Viewer beginning with "WARNING"<br>• Log file |

| Message Type | Representation in the dSPACE Software |
|---|---|
| INFO | The specified message text appears in:<br>• Log Viewer with no further assignment<br>• Log file |
| LOG | The specified message text appears in:<br>• Log file |

**Content of a message**

The message entry in the Log Viewer contains information for identifying it. It consists of:

- Name of the software module (generated)
- Message type (as specified by the user)
- Board name (generated)
- Submodule name (as specified by the user, always `MSG_SM_USER`)
- Message text (as specified by the user)

An entry in the log file contains the time and system information for each started software session. The module name, the board name, and the submodule name are added to your specified message automatically. You must only specify the message type, the message text, and the submodule name (always `MSG_SM_USER`) in the message handling functions in your application.

**Global settings for message handling**

The memory that is used for the message buffer is specified by the maximum number of messages and the maximum length of a message text. These are the defaults for the DS1603:

| Parameter | Default Value |
|---|---|
| Maximum length of a message text | 256 characters |
| Maximum number of messages | 64 messages |

If the specified message exceeds the maximum length, it is truncated.

If the number of messages exceeds the specified limit, the latest message is lost or overwrites the earliest one in the buffer. This behavior depends on the buffer mode which you can specify via the `msg_mode_set` function.

### Changing the global settings

You can find the values of the default message length and the message buffer in the `Msg1603.h` file:

```
#define MSG_STRING_LENGTH 256  /* message string length */
#define MSG_BUFFER_LENGTH 64   /* buffer length */
```

Generally, the default values are sufficient for most messages. You can change the values for solving the following problems:

- The time to generate messages is too long.
- The message module needs too much memory.

To make changes work, call `Bldlib.bat` to regenerate the DS1603 real-time library. For further information, refer to *Building Utilities for the Real-Time Library* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

### Function overview

For detailed information on the message handling functions and specific data types, refer to *Message Handling* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

# How to Implement Messages in Your Application

**Objective**

You can use messages to log the activities of your application, for example, its current status or important results. Each message is stored in the dSPACE log file and displayed according to the message type in the Log Viewer of your dSPACE software, for example, ConfigurationDesk.

**Message function used in the demo application**

The following instruction is for the `msg_info_printf` function which is used in the demo application. It contains three parameters:

- Module identification
- Message number
- Message text
  The message text can be formatted like a standard C `printf` function text.

For detailed information about this and all other functions of the message module, refer to *Message Handling* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**Restrictions**

Note the following restriction:

- Creating and transferring messages are time-consuming activities. Although the message transfer function is called in the background loop with a low priority, the message creation can block other processes, for example, when called in a loop construct.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Your application must contain at least the definitions required for the main routine, refer to *Creating the Main Routine of Your Application* on page 75.
- The message module must be initialized before any message function is used. Because the `msg_init` function is internally called by the `init` function, the message module does not have to be initialized explicitly.

- For building this application, the `DSMSG.h` header file must be included in the source code. This is done automatically if you include the `BRTENV.h` header file, which contains all the header files used for a standard application.

For detailed information, refer to *Creating the Main Routine of Your Application* on page 75.

**Method** **To implement messages in your application**

1. Type the `msg_info_printf` function in your source code, taking the restrictions into account.
2. Parameterize the **module identification**.
   For a user-specific message, you must always use the predefined symbol `MSG_SM_USER`.
3. Parameterize the **message number**.
   The message number is used to identify the message in your application. You can choose a number in the range $-2^{31}$ ... $2^{31}-1$, in either decimal or hexadecimal format.
4. Parameterize the **message text**.
   The syntax of the message text follows the rules for the standard C `printf` function. It can be a simple string, or a string with formatted placeholders for dynamic replacement by the specified parameters.

**Result** When you execute the rebuilt application, the specified message is displayed in the Log Viewer of the connected dSPACE software and written to the dSPACE log file.

The following source code excerpt shows how to implement a message with a simple string and parameterize the message number in hexadecimal format:

```
msg_info_printf(MSG_SM_USER, 0x16030102, "RTK: Timer tasks
started");
```

The following source code excerpt shows how to implement a message with placeholders and parameterize the message number in decimal format:

```
msg_info_printf(MSG_SM_USER, 10,
      "User Application: Loop has been executed %d times.",
      CurrentLoop);
```

**Next steps**

- *How to Flush the Message Buffer* on page 88

# How to Flush the Message Buffer

**Objective**

Message transfer to the host PC is managed by the background service of your application. To guarantee that the message buffer is read out if the background service has stopped, for example, because the program crashed, you can implement a subroutine to flush the message buffer.

**Restrictions**

Note the following restriction:

- Flushing the message buffer is a time-consuming activity. It should be used only if you expect a program termination, for example, because of an exception.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Your application must contain at least the definitions required for the main routine, refer to *Creating the Main Routine of Your Application* on page 75.
- For building your application, the following header files must be included in the source code:

| Header File | Used For ... |
|---|---|
| `BRTENV.h` including ... | |
| `Dsmsg.h` | ... Getting the number of buffer entries. |
| `DsECUcmdRP.h` | ... Transferring the messages from the buffer to the host PC. |

**Method**

**To flush the message buffer**

1. Use the `dci_ecu_rapidpro_bckgrnd` function to transfer the latest message from the message buffer to the host PC.
2. Use the `msg_buffer_get_no_of_entries` function to check whether the message buffer is empty. If not, you must repeat these two steps.

**Result** If you call these two functions until the message buffer is empty, you can guarantee that all messages are transferred to the host PC.

The following source code shows how to implement a subroutine for flushing the message buffer. You can also find it in the demo application (`RTFrameDS1603.c`).

```
static void DsaSendMsgBuffer(void)
{
   UInt32 BufferEntryCnt;

   do
   {
      dci_ecu_rapidpro_bckgrnd();
      BufferEntryCnt = msg_buffer_get_no_of_entries();
   } while(BufferEntryCnt > 0);
}
```

# Implementing Time-Stamping

**Objective**

The RTLib1603 provides functions which can be used to take absolute time stamps from a highly accurate, absolute time base.

**Where to go from here**

Information in this section

| |
|---|
| ***Basics of Time-Stamping*** **on page 92**<br>When an application is running, you can measure absolute times using specific functions. |
| ***How to Implement Time-Stamping*** **on page 93**<br>If you want to monitor, for example, the execution time of a real-time application, you must implement time-stamping. |

# Basics of Time-Stamping

**Access of the absolute time**

The RTLib1603 provides functions which can be used to take the absolute time from a highly accurate, absolute time base. These functions provide sufficiently accurate samples of the independent variable Time. If signals and events have been recorded together with the associated time stamps, it is possible to reconstruct their temporal order.

**Function module**

The time-stamping functions are provided by the DSTS function module. The `DSTS.h` header file contains all function declarations, global variables, and required data types.

**Time base of the DS1603 board**

The local clock of the MPC5554 processor acts as the time base of the DS1603 board. The clock rate is 112 MHz (resolution: 8.928 ns).

Each tick of the MPC5554 processor clock increments a microtick counter (32 bit). Due to the processor's clock rate, one microtick lasts `dsts_mit_period` = 8.928 ns. After $2^{32}$ microticks a macrotick counter (32 bit) is incremented, and the microtick counter is reset to 0. One macrotick lasts `dsts_mat_period` = 38.348 s. The macrotick counter would theoretically overrun after 5175 years.

**Time-stamping method**

The values of the microticks and macrotick counters are written to a predefined time stamp structure (64 bit). You can read the time stamp structure and calculate the absolute time in seconds using predefined time-stamping functions.

**Function overview**

For detailed information on the time-stamping functions, specific data types, and global variables, refer to *Time-Stamping* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

# How to Implement Time-Stamping

**Objective**

Time-stamping enables you to monitor, for example, the execution time of a real-time application.

**Preconditions**

Before you can implement time-stamping, the `main` routine of the real-time frame must be set up, as described in *How to Create the Main Routine from Scratch* on page 78. This calls the `ts_init()` time-stamping initialization function via the `init()` function, and includes the `dsts.h` header file via the `Brtenv.h` header file.

**Method**

**To implement time-stamping**

1. Specify a pointer to a time-stamp structure using the `ts_timestamp_type` data type.
2. Read the microtick counter:

   `ts_timestamp_read();`

   The function writes the current number of microticks to the time stamp structure specified in step 1.
3. If you want to calculate the absolute time in seconds from the number of microticks, use the `ts_time_calculate()` function.

**Result**

You have implemented time-stamping.

The source code fragment shown below is part of the `main` routine in the `RTFrameDS1603.c` file (Demo Application).

As soon as the RTLib1603 is initialized, a first time stamp is measured via `ts_timestamp_read(&ts1)`. This is the start time. Further time stamps are measured at the beginning of the background loop via `ts_timestamp_read(&ts2)`. When 5 seconds have passed, a new time interval is started and the system checks whether the **background alive** message is to be output.

```
...
void main(void)
{
    ...
    // Two pointers to a timestamp structure
    ts_timestamp_type ts1, ts2;
    // Variable that holds time values in seconds
    Float32 time;
...
    while( 1 )
    {
        // Measurement of the current time
        ts_timestamp_read(&ts2);

        // Check every very 5 seconds whether the 'background alive' message should be output
        if( 5.0f <= ts_timestamp_interval(&ts1, &ts2) )
        {
            // start new time interval
            ts_timestamp_read(&ts1);

            if( BACKGROUND_ALIVE_MSG )
            {
                msg_info_printf(0, 0x16030110, "RTK: background alive (called %d times)", (int) s_BackgroundCalls);
                s_BackgroundCalls = 0;
            }
         ...
    }
}
...
```

# Implementing Exception Handling

**Objective**

One aspect of good programming is ensuring the application has robust error handling. One requirement for this is exception handling for a defined reactions to erroneous program states.

**Where to go from here**

Information in this section


| |
|---|
| ***Basics on Exception Handling*** **on page 96** |
| ***How to Implement Exception Handling*** **on page 98** |

# Basics on Exception Handling

**Exception types**

The exceptions which can be handled by the RTLib1603 are predefined exceptions of the MPC5554 microcontroller. Some of them are critical. This means that the erronous state cannot be corrected by the application, and the application terminates. Other exceptions are uncritical, and can be corrected by algorithms implemented in your application.

| Predefined Exception[1)] | Description |
|---|---|
| Critical exception (program terminates) | |
| EXCEPTION_CRITICAL_INPUT_NUMBER | Critical input exception |
| EXCEPTION_MACHINE_CHECK_NUMBER | Machine check exception |
| EXCEPTION_DATA_STORAGE_NUMBER | Data storage exception |
| EXCEPTION_INSTR_STORAGE_NUMBER | Instruction storage exception |
| EXCEPTION_ALIGN_ERR_NUMBER | Alignment error exception |
| EXCEPTION_FP_UNAVAIL_NUMBER | Floating-point unavailable exception |
| EXCEPTION_APU_UNAVAILABLE_NUMBER[2)] | Auxiliary processing unit (APU) unavailable exception |
| EXCEPTION_DATA_TLB_ERROR_NUMBER | Data TLB error exception |
| EXCEPTION_INSTR_TLB_ERROR_NUMBER | Instruction TLB error exception |
| EXCEPTION_DEBUG_NUMBER | Debug exception |
| EXCEPTION_SPE_UNAVAILABLE_NUMBER | Signal processing engine (SPE) unavailable exception |
| EXCEPTION_PROG_ERR_NUMBER | Program error exception |
| Uncritical exception (program can continue) | |
| EXCEPTION_SYSTEM_CALL_NUMBER | System call exception |
| EXCEPTION_DECREMENTER_NUMBER | Decrementer exception |
| EXCEPTION_FIXED_INTERVAL_TMR_NUMBER | Fixed interval timer exception |
| EXCEPTION_WATCHDOG_TMR_NUMBER | Watchdog timer exception |
| EXCEPTION_SPE_DATA_EXCEPTION_NUMBER | SPE data exception |
| EXCEPTION_SPE_ROUND_EXCEPTION_NUMBER | SPE round exception (deactivated) |

1) For further information, refer to the MPC5554 Data Sheet from Freescale

2) This exception does not occur for the MPC5554 but is specified for it. For further information, refer to the MPC5554 Data Sheet from Freescale.

**Required functions**

For a basic implementation of the exception handling, you need the following functions:

- A hook function for each exception type.
- An initialization function
  With the initialization function, you install the hook function which is to be executed for the specified exception. For each exception type, you want to handle by a hook function, you must call the initialization function.

**Refinement of exception handling**

The RTLib1603 provides two additional functions for obtaining information about an exception:

- Using the `dsmpc5554_except_pc_get` function, you can get the program address at which an exception occurred.
- Exceptions of `EXCEPTION_SPE_DATA_EXCEPTION_NUMBER` type can be refined by using the `dsmpc5554_arithm_operation_status_get` function. This returns the cause of an exception triggered by the last arithmetic operation, for example, overflow, underflow, division by zero, or invalid operator.

**Function overview**

For detailed information on the exception handling functions, refer to *Exception Handling* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

# How to Implement Exception Handling

**Objective**

Robust application behavior can be achieved by implementing exception handling for a defined reaction to an erroneous program state.

**Functions used in the demo application**

There are many ways to implement exception handling. It mainly depends on your requirements regarding exception handling. The following instruction describes a basic implementation of exception handling used in the demo application (`RTFrameDS1603.c`). It consists of one hook function for critical exceptions that results in termination of your application, and one hook function for uncritical exceptions. The hook functions are assigned in one initialization function.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Your application must contain at least the definitions required for the main routine, refer to *Creating the Main Routine of Your Application* on page 75.
- For building your application, the `Ds1603Init.h` and `DsMPC5554Def.h` header files must be included in the source code. This is done automatically if you include the `BRTENV.h` header file, which contains all the header files used for standard applications.

**Method**

**To implement exception handling**

1. Declare the global variables required for your exception handling, for example, a counter for uncritical exceptions that occur.
2. Create functions that you want to be executed if a specific exception occurs.
   The syntax of a hook function is:

```
void <hook_function_name>(UInt16 Number){
…
/* Custom source code */
}
```

For example, you can create one function for critical exceptions that results in termination of your application, and one function for uncritical exceptions after which the application can continue.

The `Number` parameter is automatically set by the exception handling module to the ID of the exception type.

3 Assign these functions to the possible exception handlers by using the `ds1603_exception_hook_fcn_init` function.
The exception hook functions must be initialized as soon as possible in the main routine of your application to catch all possible exceptions.

**Result** If an exception occurs in your application, the installed hook function is executed.

The following source code shows how to implement hook functions for exception handling. You can also find this source code in the application template (`%DSPACE_ROOT%\Demos\DS1603\GettingStarted\HandCode\RTFrameDs1603.c`).

- The first hook function defines the behavior of the application before it terminates because of a critical exception. It creates a warning message in the message buffer. To guarantee that this message is displayed, for example, in the ConfigurationDesk Log Viewer, the message buffer is flushed by using the `DsaSendMsgBuffer` function (refer to *How to Flush the Message Buffer* on page 88).

```
static void DsaExceptionHookFcnGoodbye(UInt16 ExceptionID)
{
   // program execution stops after these exceptions
   msg_warning_printf(0, 0x16030E00+ExceptionID,
      "RTLib: exception (ID =0x%x) detected", ExceptionID);
   DsaSendMsgBuffer();
}
```

- The next hook function defines the behavior of the application if an uncritical exception occurs. The application continues execution. The exceptions that occur are counted and the latest exception is stored by its ID. The number of exceptions that occured and the ID of the last exception can be displayed, for example, in the background loop of your main routine, by using the `msg_printf` function.

```
static void DsaExceptionHookFcnContinue(UInt16 ExceptionID)
{
   // program execution can continue after these exceptions,
   // remember that an exception has occured
   s_ExceptionCnt++;
   s_LastException = ExceptionID;
}
```

- When you have defined the hook functions, you can assign them to the available exceptions using the `ds1603_exception_hook_fcn_init` function. To simplify the initialization of the exception hook functions, you can list them all in one function, as shown below.

```
static void DsaInitExceptionHooks(void)
{
    // exceptions after which continuation is not possible
    ds1603_exception_hook_fcn_init(EXCEPTION_CRITICAL_INPUT_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_MACHINE_CHECK_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_DATA_STORAGE_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_INSTR_STORAGE_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_ALIGN_ERR_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_FP_UNAVAIL_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_APU_UNAVAILABLE_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_DATA_TLB_ERROR_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_INSTR_TLB_ERROR_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_DEBUG_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_SPE_UNAVAILABLE_NUMBER, DsaExceptionHookFcnGoodbye);
    ds1603_exception_hook_fcn_init(EXCEPTION_PROG_ERR_NUMBER, DsaExceptionHookFcnContinue);

    // exceptions after which continuation is possible
    ds1603_exception_hook_fcn_init(EXCEPTION_SYSTEM_CALL_NUMBER, DsaExceptionHookFcnContinue);
    ds1603_exception_hook_fcn_init(EXCEPTION_DECREMENTER_NUMBER, DsaExceptionHookFcnContinue);
    ds1603_exception_hook_fcn_init(EXCEPTION_FIXED_INTERVAL_TMR_NUMBER, DsaExceptionHookFcnContinue);
    ds1603_exception_hook_fcn_init(EXCEPTION_WATCHDOG_TMR_NUMBER, DsaExceptionHookFcnContinue);
    ds1603_exception_hook_fcn_init(EXCEPTION_SPE_DATA_EXCEPTION_NUMBER, DsaExceptionHookFcnContinue);
    ds1603_exception_hook_fcn_init(EXCEPTION_SPE_ROUND_EXCEPTION_NUMBER, DsaExceptionHookFcnContinue);
}
```

# Creating Tasks

**Objective**

Tasks are required if you want to calculate parts of your controller model at specific rates and in a specific execution order.

**Where to go from here**

Information in this section

| |
|---|
| ***Task Types*** **on page 103**<br>A real-time application commonly consists of periodic and/or aperiodic real-time tasks, and one background task. Their main characteristics are described. |
| ***Task Execution Order*** **on page 108**<br>Real-time tasks are executed according to the priority-based preemptive task scheduler of the dSPACE Real-Time Kernel, taking priorities and current states into account. |
| ***How to Implement One Periodic Task*** **on page 110**<br>If a task is to be executed at equidistant intervals, you must implement it as a periodic task. |
| ***How to Implement Multiple Periodic Tasks*** **on page 115**<br>You must note some points on synchronization when implementing multiple periodic tasks. |

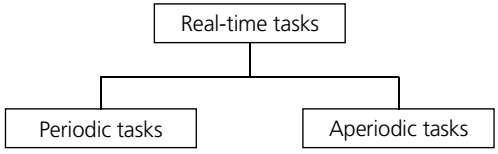
| |
|---|
| ***How to Implement Aperiodic Tasks*** **on page 121**<br>If a task is to be executed as a reaction to a specific condition, you must implement it as an aperiodic task. |
| ***How to Implement Inherited Tasks*** **on page 126**<br>If a task is to be triggered from within another task, you must implement it as an inherited task. |
| ***Task Overrun Handling*** **on page 131**<br>If a task is requested to start even though it is still running, an overrun situation occurs. The RTK1603 provides several options for reacting to this. |
| ***How to Implement Overrun Handling*** **on page 133**<br>If you want a certain reaction to a task overrun, you must implement task-specific overrun handling. |

# Task Types

**Use of tasks**

Your controller model usually consists of parts that have to be calculated periodically and others that have to be calculated as a reaction to an external event, for example, a rising edge of an input signal. These parts must be assigned to tasks.

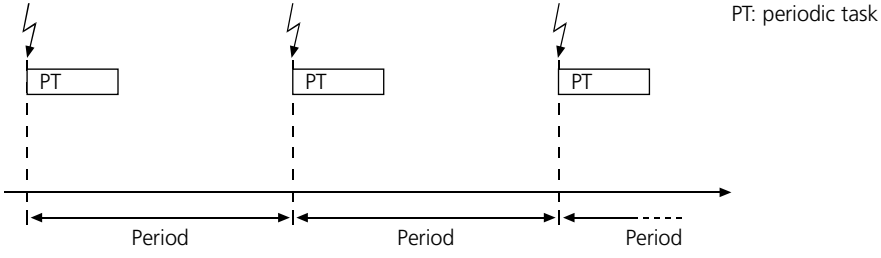
Real-time tasks can be grouped according to usage, as follows:




**Periodic real-time tasks**

Periodic real-time tasks are functions to be executed at defined time intervals, for example, the sampling of an input at equidistant intervals. A real-time task becomes periodic if it is bound to a periodic interrupt service, for example, to a timer.

To schedule periodic tasks, you must specify their task periods.

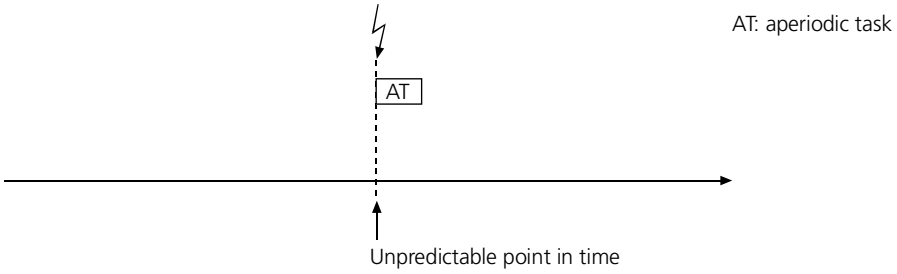



Periodic tasks need to be controlled by a designated timer, otherwise equidistant intervals cannot be achieved.

For instructions on implementing a periodic task, refer to *How to Implement One Periodic Task* on page 110 or *How to Implement Multiple Periodic Tasks* on page 115.

**Aperiodic real-time tasks**

Aperiodic tasks do not follow a certain time scheme, but are executed as required. For example, an external trigger signal via TPU channel indicates that a temperature limit has been exceeded.




To determine the timing characteristics (absolute time) of an aperiodic task, it must be linked to a periodic reference task. The RTK1603 calculates the timing characteristics of an aperiodic task with respect to the trigger time of its periodic reference task. Commonly, the base rate task of the controller model is referenced. If no task is referenced, the RTK1603 references the hardware time by reading the absolute time from the time-stamping module.

For instructions on implementing an aperiodic task, refer to *How to Implement Aperiodic Tasks* on page 121.

The services for triggering aperiodic tasks are provided by the I/O drivers.

**Inherited tasks** Inherited tasks are a variant of real-time tasks. If a task is triggered from within a real-time task (parent task) by a software-based interrupt service, it is called inherited. The parent task can be periodic or aperiodic.

For example, a software-based interrupt that triggers a specific task (the inherited task) is generated according to the result of a logical operation performed in a periodic real-time task.

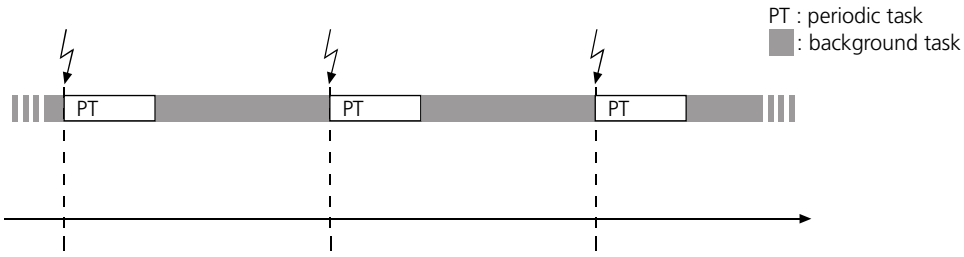
To determine the timing characteristics of an inherited task, the system assumes the inherited task and its parent task have the same trigger time and duration.

If an inherited task is not triggered from within another task, its timing characteristics become invalid.

For instructions on implementing an inherited task, refer to *How to Implement Inherited Tasks* on page 126.

**Background task**

The background task is not a task in the common sense. It is implemented as an endless loop at the end of the main routine. It has no priority. The background task is executed whenever no real-time task is active. However, you cannot predict when or how often the background task is executed. The calculations made in it do not have specific timing, but are either distributed over several base sampling steps or executed several times between two base sampling steps.




Because the background task is interrupted by all real-time tasks, it is not suitable for real-time jobs. The background task mainly deals with administrative jobs, for example, it performs communication between the RapidPro system and the host PC.

**Interrupts as task triggers**

A real-time task is executed when it has been triggered by an interrupt. Thus, a real-time task must be bound to an interrupt source, also called a service. Interrupts can be generated by different services, for example, the timer service `S_INTERVAL_A`, a service for periodic timer interrupts. Services can be hardware-based or software-based.

One service can trigger multiple real-time tasks by delivering different subinterrupts. One subinterrupt triggers one real-time task. (Thus, a real-time task is actually triggered by a subinterrupt.) Each subinterrupt is defined by the service it belongs to and a subinterrupt ID number. The timer service, for example, provides $2^{32}$ subinterrupts, that is, you could trigger $2^{32}$ different real-time tasks with it.

The following table shows the available services and their identifiers.

| Interrupt Service | Service Identifier |
|---|---|
| Generic RTK service | `S_NONE` |
| Generic RTK service, software triggered tasks | `S_SOFTTASK` |
| TPU channels (2 services) | `S_ETPU_…` |
| Periodic timer interrupts | `S_INTERVAL_A` |
| ADC channels/queues (6 services) | `S_EQADC…` |

For detailed information on the interrupt services, refer to the *Interrupt Service Handling* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

**Task priorities**

A real-time task must have a priority assigned to it. This task parameter is important for scheduling the execution order of multiple real-time tasks in a running application. The possible range for tasks periods is 1 ... 128. A low value means a high priority.

You can assign the same priority to several real-time tasks.

**Storage of task data**

Each real-time task is represented by a task control block (TCB). A TCB is a data structure that contains all the necessary information on scheduling, overrun handling, and interrupt services. A TCB is created when the `rtk_create_task()` function has been called. Task control blocks are sorted by the tasks' priorities. They are linked together to form a task list.

For a detailed description of the `rtk_create_task()` function, refer to *rtk_create_task* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

# Task Execution Order

**Task controlling via RTK1603**

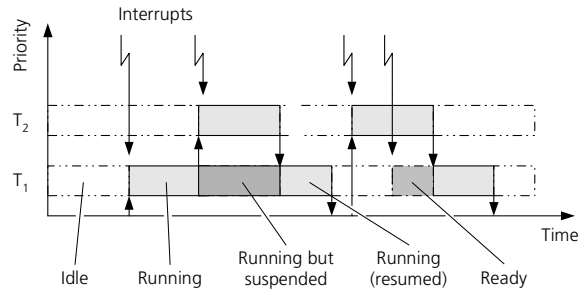
Only one task can be executed at a time (including the background task). The execution of multiple real-time tasks requires sophisticated task controlling. The RTK1603 manages tasks, handles interrupts, and schedules application tasks efficiently.

**Task-switching time**

A crucial value in task scheduling is the task-switching time, which defines the delay between the occurrence of an interrupt on the hardware and the execution of the task's first statement.

**Task states**

To identify which task is active, which ones are suspended, and so on, the scheduler of the RTK1603 assigns a state to each task as shown in the following illustration:




The dSPACE Real-Time Kernel uses the definitions listed in the table below:

| State | Value | Meaning |
|---|---|---|
| **Idle** | 0 | The task is inactive, waiting to be triggered. |
| **Ready** | 1 | The task has been triggered but could not start due to a high-priority task that is currently running. It is waiting to start execution. |
| **Running** | 2 | The task has started running. This state is true until the task finishes running, regardless of whether it is suspended by a high-priority task or not. |

**Task execution order**

Depending on the priorities and current states of the tasks, the scheduler of the RTK1603 executes them according to the following rules:

- A high-priority task that is triggered always suspends a low-priority task that is currently running.
- If the high-priority task has been executed, the suspended low-priority task resumes execution.
- Tasks of the same priority do not suspend each other if they are triggered, but follow a first come, first served policy.
- As long as all other tasks are idle, the background task is executed.

# How to Implement One Periodic Task

**Objective**

If a task is to be executed at equidistant intervals, you must implement it as a periodic task.

**RTK1603**

The following C functions of the RTK1603 are used for implementation:

| Function | Used For ... |
|---|---|
| `rtk_create_task()` | Task handling |
| `rtk_task_name_set()` | Task handling |
| `rtk_bind_interrupt()` | Interrupt service handling |
| `rtk_set_task_type()` | Task handling |
| `rtk_global_int_enable()` | Interrupt service handling |
| `rtk_enable_services()` | Interrupt service handling |
| `rtk_idle_function()` | Interrupt service handling |

For a description of the task handling and interrupt service handling functions, refer to *dSPACE Real-Time Kernel* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

**Scope of instructions**

Functionalities that enhance task handling but are not actually necessary for its implementation are not considered, for example, message and exception handling.

**Restrictions**

Implementing multiple periodic tasks involves more than simply repeating the steps in the following instructions. There are also task synchronization aspects to consider, refer to *How to Implement Multiple Periodic Tasks* on page 115.

**Preconditions**

You must have at least implemented the main routine, refer to *Creating the Main Routine of Your Application* on page 75.

**Method**

**To implement one periodic task**

1. Include the `rtkernel.h` header file. The file contains the definitions of the RTK1603 which is required for task handling.
2. Define a pointer to a task control block (TCB) which is used to reference the task and its data.

   ```
   rtk_p_task_control_block pMyTask;
   ```
3. Declare the required functions (depends on `rtk_create_task()` in step 5).
4. Initialize the RTK1603 by calling the following function in the main routine:

   ```
   rtk_initialize();
   ```
5. Create a new task:

   ```
   pMyTask = rtk_create_task(fcn, priority, ovc_check,
   ovc_fcn, ovc_maxcnt, user_data);
   ```
   The function allocates and initializes a task control block (TCB). It returns the address of the TCB or NULL, if there is not enough space on the heap to allocate it.
6. Specify the name of the task:

   ```
   rtk_task_name_set(pMyTask, name);
   ```
   The name can be referenced by messages, for example.
7. Bind a subinterrupt with the new task:

   ```
   rtk_bind_interrupt(service, subentry, pMyTask, sample_time,
   channel, log_int_nr, hook_fcn);
   ```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| `service` | `S_INTERVAL_A` | Mandatory for periodic tasks |
| `channel` | `C_LOCAL` | Mandatory for RapidPro systems |
| `log_int_nr` | `0` | Mandatory for RapidPro systems |

**8** Specify the type of the task, that is, periodic:

```
rtk_set_task_type(service, subentry, subsubint, task_type,
baserate_task, sample_rate_offset, step_multiple);
```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| `service` | `S_INTERVAL_A` | Mandatory for periodic tasks |
| `subsubint` | `RTK_NO_SINT` | Mandatory for RapidPro systems |
| `task_type` | `rtk_tt_periodic` | Mandatory for periodic tasks |
| `baserate_task` | `NULL` | Mandatory for periodic tasks |

**9** Enable the interrupt source (hardware):

```
rtk_global_int_enable();
```

The function globally enables all the interrupts of the RapidPro system.

**10** Enable the interrupt services:

```
rtk_enable_services();
```

The function activates all the interrupt services that have been initialized by calling the `rtk_bind_interrupt()` function.

**11** Implement the task function.

```
void fcn(rtk_p_task_control_block pTCB)
```

The task function defines what actions are triggered when the task is executed. `fcn` has been registered by the `rtk_create_task()` function (step 5).

**12** Implement the idle function:

```
void rtk_idle_function(void)
{
}
```

**Result** You have implemented one periodic task.

Suppose you want to implement a periodic task as follows:

| Task Parameter | Implementation |
|---|---|
| Subentry of the service | 0 |
| Pointer to the task control block (TBC) | s_pTaskATCB |
| Pointer to the task function | DsaTaskA |
| Priority | 12 |
| Overrun handling method | ocv_fcn |
| Overrun function | DsaTaskAOverrunFcn |
| Number of permitted overruns | INT_MAX |
| User data | – |
| Task name | PWM_Read |
| Task period | 0.002f |
| Sample rate offset | 0.0f |
| Subinterrupt hook function | NULL |

The resulting C code can look like the code fragment shown below, which is an adapted excerpt from the `RTFrameDS1603.c` dSPACE demo file.

```
...
#include <rtkernel.h>
...
// global variables
rtk_p_task_control_block s_pTaskATCB;
...
void main(void)
{
...
    rtk_initialize();
...
    s_pTaskATCB = rtk_create_task(DsaTaskA, 12, ovc_fcn, DsaTaskAOverrunFcn, INT_MAX, 0);
    rtk_task_name_set(s_pTaskATCB, "PWM_Read");
    rtk_bind_interrupt(S_INTERVAL_A, 0, s_pTaskATCB, 0.002f, C_LOCAL, 0, NULL);
    rtk_set_task_type(S_INTERVAL_A, 0, RTK_NO_SINT, rtk_tt_periodic, NULL, 0.0f, 1);
...
    rtk_global_int_enable();
    rtk_enable_services();
...
}
...
void DsaTaskA(rtk_p_task_control_block pTCB)
{
...
    /* task code */
    TASKA_STEP_FUNCTION();
...
}
...
void rtk_idle_function(void)
{
}
...
```

# How to Implement Multiple Periodic Tasks

**Objective**

If several tasks are to be executed at equidistant intervals, you must implement them as periodic tasks. The rates of the tasks can differ.

**RTK1603**

The following C functions of the RTK1603 are used for implementation:

| Function | Used For ... |
|---|---|
| `rtk_create_task()` | Task handling |
| `rtk_task_name_set()` | Task handling |
| `rtk_bind_interrupt()` | Interrupt service handling |
| `rtk_set_task_type()` | Task handling |
| `rtk_it_task_register_rel()` | Task handling (synchronization of multiple periodic tasks) |
| `rtk_global_int_enable()` | Interrupt service handling |
| `rtk_enable_services()` | Interrupt service handling |
| `rtk_idle_function()` | Interrupt service handling |

For a description of the task handling and interrupt service handling functions, refer to *dSPACE Real-Time Kernel* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

**Scope of instructions**

Functionalities that enhance task handling but are not actually necessary for its implementation are not considered, for example, message and exception handling.

**Preconditions**

You must have at least implemented the main routine, refer to *Creating the Main Routine of Your Application* on page 75.

**Method** **To implement multiple periodic tasks**

1. Include the `rtkernel.h` header file. The file contains the definitions of the RTK1603 which is required for task handling.
2. Define a pointer to a task control block (TCB) which is used to reference the task and its data.

   ```
   rtk_p_task_control_block pMyTask;
   ```
3. Declare the required functions (depends on `rtk_create_task()` in step 5):
4. Initialize the RTK1603 by calling the following function in the main routine (only once for multiple tasks):

   ```
   rtk_initialize();
   ```
5. Create a new task:

   ```
   pMyTask = rtk_create_task(fcn, priority, ovc_check,
   ovc_fcn, ovc_maxcnt, user_data);
   ```
   The function allocates and initializes a task control block (TCB). It returns the address of the TCB or NULL, if there is not enough space on the heap to allocate it
6. Specify the name of the task:

   ```
   rtk_task_name_set(pMyTask, name);
   ```
   The name can be referenced by messages, for example.
7. Bind a subinterrupt with the new task:

   ```
   rtk_bind_interrupt(service, subentry, pMyTask, sample_time,
   channel, log_int_nr, hook_fcn);
   ```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| `service` | `S_INTERVAL_A` | Mandatory for periodic tasks |
| `sample_time` | `0.0` (ignored) | Set via `rtk_it_task_register_rel()` in step 12 |
| `channel` | `C_LOCAL` | Mandatory for RapidPro systems |
| `log_int_nr` | `0` | Mandatory for RapidPro systems |

**8** Specify the type of the task, that is, periodic:

```
rtk_set_task_type(S_INTERVAL_A, subentry, subsubint,
rtk_tt_periodic, baserate_task, sample_rate_offset,
step_multiple);
```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| `service` | `S_INTERVAL_A` | Mandatory for periodic tasks |
| `subsubint` | `RTK_NO_SINT` | Mandatory for RapidPro systems |
| `task_type` | `rtk_tt_periodic` | Mandatory for periodic tasks |
| `baserate_task` | `NULL` | Mandatory for periodic tasks |

**9** If you want to implement an additional periodic task, go to step 2, otherwise proceed with step 10.

**10** Enable the interrupt source (hardware):

```
rtk_global_int_enable();
```

The function globally enables all the interrupts of the RapidPro system.

**11** Enable the interrupt services:

```
rtk_enable_services();
```

The function activates all the interrupt services that have been initialized by calling the `rtk_bind_interrupt()` function.

**12** Read the absolute time (time stamp):

```
ts_timestamp_read(&ts1);
```

The function returns the absolute time as a time stamp structure.

**13** Register all tasks in the interval timer (IT) queue. You must specify a separate register function for each task. However, all registered functions refer to the same time stamp (absolute time).

```
rtk_it_task_register_rel(service, subentry, subsubint,
interval, deadline, sample_time, &ts1);
```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| `service` | `S_INTERVAL_A` | Mandatory for periodic tasks |
| `subsubint` | `RTK_NO_SINT` | Mandatory for RapidPro systems |
| `task_type` | `rtk_tt_periodic` | Mandatory for periodic tasks |

**14** Implement the task function for each task as a subroutine after the main routine.

```
void task(rtk_p_task_control_block pTCB)
```

The task function defines what actions are triggered when the task is executed. `fcn` has been registered by the `rtk_create_task()` function (step 4).

**15** Implement the idle function:

```
void rtk_idle_function(void)
{
}
```

**Result** You have implemented multiple periodic tasks.

Suppose you want to implement two periodic tasks as follows:

| Task Parameter | Task A | Task B |
|---|---|---|
| Subentry of the service | 0 | 0 |
| Pointer to the task control block (TBC) | s_pTaskATCB | s_pTaskBTCB |
| Pointer to the task function | DsaTaskA | DsaTaskB |
| Priority | 12 (higher priority) | 13 (lower priority) |
| Overrun handling method | ocv_fcn | ocv_fcn |
| Overrun function | DsaTaskAOverrunFcn | DsaTaskBOverrunFcn |
| Number of permitted overruns | INT_MAX | INT_MAX |
| User data | – | – |
| Task name | PWM_Read | Servo_Read |
| Task period | 0.002f | 0.01f |
| Sample rate offset | 0.0f | 0.0f |
| Subinterrupt hook function | NULL | NULL |
| `interval [rtk_it_task_register_rel()]` | 0.002f (task period) | 0.01f (task period) |
| `deadline [rtk_it_task_register_rel()]` | 0 (no deadline) | 0 (no deadline) |

The resulting C code can look like the code fragment shown below, which is an adapted excerpt from the `RTFrameDS1603.c` dSPACE demo file.

```
...
#include <rtkernel.h>
...
// global variables
rtk_p_task_control_block s_pTaskATCB;
rtk_p_task_control_block s_pTaskBTCB;
...
void main(void)
{
...
    rtk_initialize();
...
    // Task A
    s_pTaskATCB = rtk_create_task(DsaTaskA, 12, ovc_fcn, DsaTaskAOverrunFcn, INT_MAX, 0);
    rtk_task_name_set(s_pTaskATCB, "PWM_Read");
    rtk_bind_interrupt(S_INTERVAL_A, 0, s_pTaskATCB, 0.0f, C_LOCAL, 0, NULL);
    rtk_set_task_type(S_INTERVAL_A, 0, RTK_NO_SINT, rtk_tt_periodic, NULL, 0.0f, 1);
...
    // TaskB
    s_pTaskBTCB = rtk_create_task(DsaTaskB, 13, ovc_fcn, DsaTaskBOverrunFcn, INT_MAX, 0);
    rtk_task_name_set(s_pTaskBTCB, "Servo_Read");
    rtk_bind_interrupt(S_INTERVAL_A, 1, s_pTaskATCB, 0.0f, C_LOCAL, 0, NULL);
    rtk_set_task_type(S_INTERVAL_A, 1, RTK_NO_SINT, rtk_tt_periodic, NULL, 0.0f, 1);
...
    rtk_global_int_enable();
    rtk_enable_services();
...
    ts_timestamp_read(&ts1);
    rtk_it_task_register_rel(S_INTERVAL_A, 0, RTK_NO_SINT, 0.002f, 0, 0.002f, &ts1);
    rtk_it_task_register_rel(S_INTERVAL_A, 1, RTK_NO_SINT, 0.01f, 0, 0.01f, &ts1);
...
}
...
// Task A
void DsaTaskA(rtk_p_task_control_block pTCB)
{
...
    /* task code */
    TASKA_STEP_FUNCTION();
...
}
...
// Task B
void DsaTaskB(rtk_p_task_control_block pTCB)
{
...
    /* task code */
    TASKB_STEP_FUNCTION();
...
}
...
void rtk_idle_function(void)
{
}
...
```

# How to Implement Aperiodic Tasks

**Objective**

If a task is to be triggered by asynchronous trigger sources, you must implement it as an aperiodic task.

**RTK1603**

The following C functions of the RTK1603 are used for implementation:

| Function | Used For ... |
|---|---|
| `rtk_create_task()` | Task handling |
| `rtk_task_name_set()` | Task handling |
| `rtk_bind_interrupt()` | Interrupt service handling |
| `rtk_set_task_type()` | Task handling |
| `rtk_global_int_enable()` | Interrupt service handling |
| `rtk_enable_services()` | Interrupt service handling |
| `rtk_idle_function()` | Interrupt service handling |

For a description of the task handling and interrupt service handling functions, refer to *dSPACE Real-Time Kernel* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

**Scope of instructions**

Functionalities that enhance task handling but are not actually necessary for its implementation are not considered, for example, message and exception handling.

**Preconditions**

You must have at least implemented the main routine, refer to *Creating the Main Routine of Your Application* on page 75.

**Method** **To implement an aperiodic task**

1. Include the `rtkernel.h` header file. The file contains the definitions of the RTK1603 which is required for the task handling.
2. Define a pointer to a task control block (TCB) which is used to reference the task and its data.

   ```
   rtk_p_task_control_block pMyTask;
   ```
3. Declare the required functions (depends on `rtk_create_task()` in step 5):
4. Initialize the RTK1603 by calling the following function in the main routine:

   ```
   rtk_initialize();
   ```
5. Create a new task:

   ```
   pMyTask = rtk_create_task(fcn, priority, ovc_check,
   ovc_fcn, ovc_maxcnt, user_data);
   ```
   The function allocates and initializes a task control block (TCB). It returns the address of the TCB or NULL, if there is not enough space on the heap to allocate it
6. Specify the name of the task:

   ```
   rtk_task_name_set(pMyTask, name);
   ```
   The name can be referenced by messages, for example.
7. Bind a subinterrupt with the new task:

   ```
   rtk_bind_interrupt(service, subentry, pMyTask, sample_time,
   channel, log_int_nr, hook_fcn);
   ```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| `sample_time` | `0.0` (ignored) | Mandatory for aperiodic tasks |
| `channel` | `C_LOCAL` | Mandatory for RapidPro systems |
| `log_int_nr` | `0` | Mandatory for RapidPro systems |

8 Specify the type of the task, that is, aperiodic:

```
rtk_set_task_type(service, subentry, subsubint, task_type,
baserate_task, sample_rate_offset, step_multiple);
```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| subsubint | RTK_NO_SINT | Mandatory for RapidPro systems |
| task_type | rtk_tt_aperiodic | Mandatory for aperiodic tasks |
| sample_rate_offset | 0.0f | Mandatory for aperiodic tasks |

9 Enable the interrupt source (hardware):

```
rtk_global_int_enable();
```

The function globally enables all the interrupts of the RapidPro system.

10 Enable the interrupt services:

```
rtk_enable_services();
```

The function activates all the interrupt services that have been initialized by calling the `rtk_bind_interrupt()` function.

11 Implement the task function.

```
void fcn(rtk_p_task_control_block pTCB)
```

The task function defines what actions are triggered when the task is executed. `fcn has been registered` by the `rtk_create_task()` function (step 5).

12 Implement the idle function:

```
void rtk_idle_function(void)
{
}
```

**Result** You have implemented an aperiodic task.

Suppose you want to implement an aperiodic task as follows:

| Task Parameter | Implementation |
|---|---|
| Service | S_ETPU_A (TPU A) |
| Subentry of the service | 0 (interrupt channel 1) |
| Pointer to the task control block (TBC) | s_pTaskExtTCB |
| Pointer to the task function | DsaTaskExt |
| Priority | 11 |
| Overrun handling method | ocv_fcn |
| Overrun function | DsaTaskExtOverrunFcn |
| Number of permitted overruns | INT_MAX |
| User data | – |
| Task name | Synchronization_Lost |
| Base rate task | NULL<br>(reference microprocessor clock) |
| Sample rate offset | 0.0f |
| Subinterrupt hook function | NULL |

The resulting C code can look like the code fragment shown below, which is an adapted excerpt from the `RTFrameDS1603.c` dSPACE demo file.

```
...
#include <rtkernel.h>
...
// global variables
rtk_p_task_control_block s_pTaskExtTCB;
...
void main(void)
{
...
    rtk_initialize();
...
    s_pTaskExtTCB = rtk_create_task(DsaTaskExt, 11, ovc_fcn, DsaTaskExtOverrunFcn, INT_MAX, 0);
    rtk_task_name_set(s_pTaskExtTCB, "Synchronization_Lost");
    rtk_bind_interrupt(S_ETPU_A, 0, s_pTaskExtTCB, 0.0f, C_LOCAL, 0, NULL);
    rtk_set_task_type(S_ETPU_A, 0, RTK_NO_SINT, rtk_tt_aperiodic, NULL, 0.0f, 1);
...
    rtk_global_int_enable();
    rtk_enable_services();
...
}
...
void DsaTaskExt(rtk_p_task_control_block pTCB)
{
...
    /* task code */
    TASKExt_STEP_FUNCTION();
...
}
...
void rtk_idle_function(void)
{
}
...
```

# How to Implement Inherited Tasks

**Objective**

If a task is to be triggered from within another task by a software-based interrupt, you must implement it as an inherited task. The parent task can be periodic or aperiodic.

**RTK1603**

The following C functions of the RTK1603 are used for implementation:

| Function | Used For ... |
|---|---|
| `rtk_create_task()` | Task handling |
| `rtk_task_name_set()` | Task handling |
| `rtk_bind_interrupt()` | Interrupt service handling |
| `rtk_set_task_type()` | Task handling |
| `rtk_trigger_interrupt()` | Interrupt service handling |

For a description of the task handling and interrupt service handling functions, refer to *dSPACE Real-Time Kernel* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

**Scope of instructions**

Instructions for implementing the parent task are not described.

Functionalities that enhance task handling but are not actually necessary for its implementation are not considered, for example, message and exception handling.

**Preconditions**

You must have implemented the parent task, refer to *How to Implement One Periodic Task* on page 110, for example.

**Method** **To implement an inherited task**

1. Define a pointer to a task control block (TCB) which is used to reference the task and its data.

   ```
   rtk_p_task_control_block pMyTask;
   ```

2. Declare the required functions (depends on `rtk_create_task()` in step 5):

3. Create a new task:

   ```
   pMyTask = rtk_create_task(fcn, priority, ovc_check,
   ovc_fcn, ovc_maxcnt, user_data);
   ```

   The function allocates and initializes a task control block (TCB). It returns the address of the TCB or NULL, if there is not enough space on the heap to allocate it

4. Specify the name of the task:

   ```
   rtk_task_name_set(pMyTask, name);
   ```

   The name can be referenced by messages, for example.

5. Bind a subinterrupt with the new task:

   ```
   rtk_bind_interrupt(service, subentry, pMyTask, sample_time,
   channel, log_int_nr, hook_fcn);
   ```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| `service` | `S_SOFTTASK` | Mandatory for inherited tasks |
| `sample_time` | `0.0` (ignored) | Mandatory for inherited tasks |
| `channel` | `C_LOCAL` | Mandatory for RapidPro systems |
| `log_int_nr` | `0` | Mandatory for a RapidPro systems |

6 Specify the type of the task, that is, inherited:

```
rtk_set_task_type(service, subentry, subsubint, task_type,
baserate_task, sample_rate_offset, step_multiple);
```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| service | S_SOFTTASK | Mandatory for inherited task |
| subsubint | RTK_NO_SINT | Mandatory for RapidPro systems |
| task_type | rtk_tt_inherited | Mandatory for inherited task |
| sample_rate_offset | 0.0f | Mandatory for inherited task |

7 Implement the task function.

```
void fcn(rtk_p_task_control_block pTCB)
```

The task function defines what actions are triggered when the task is executed. `fcn has been registered` by the `rtk_create_task()` function (step 3).

8 Implement a software trigger for the inherited task in the task function of the parent task as needed:

```
rtk_trigger_interrupt(service, subentry);
```

| Parameter | Mandatory Setting | Reason |
|---|---|---|
| service | S_SOFTTASK | Mandatory for inherited tasks |

The subentry must equal the subentry specified in the `rtk_bind_interrupt()` function (step 4).

**Result** You have implemented an inherited task. The inherited task inherits its timing characteristics from its parent task.

Suppose you want to implement an inherited task as follows:

| Task Parameter | Implementation |
|---|---|
| Parent task | DsaTaskA |
| Subentry of the service | 0 (interrupt channel 1) |
| Pointer to the task control block (TBC) | s_pTaskSoftTCB |
| Pointer to the task function | DsaTaskSoft |
| Priority | 10 |
| Overrun handling method | ocv_fcn |
| Overrun function | DsaTaskSoftOverrunFcn |
| Number of permitted overruns | INT_MAX |
| User data | – |
| Task name | Speed_Restrict |
| Base rate task | NULL<br>(reference microprocessor clock) |
| Sample rate offset | 0.0f |
| Subinterrupt hook function | NULL |

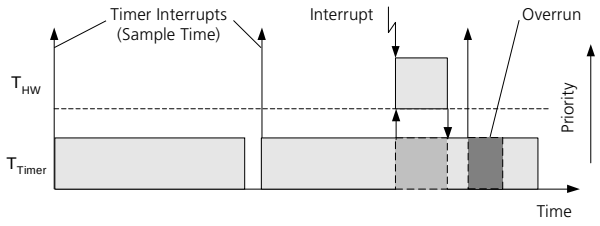
The resulting C code can look like the code fragment shown below.

```
...
// global variables
...
rtk_p_task_control_block s_pTaskSoftTCB;
...
void main(void)
{
...
    s_pTaskSoftTCB = rtk_create_task(DsaTaskSoft, 10, ovc_fcn, DsaTaskSoftOverrunFcn, INT_MAX, 0);
    rtk_task_name_set(s_pTaskSoftTCB, "Speed_Restrict");
    rtk_bind_interrupt(S_SOFTTASK, 0, s_pTaskSoftTCB, 0.0f, C_LOCAL, 0, NULL);
    rtk_set_task_type(S_SOFTTASK, 0, RTK_NO_SINT, rtk_tt_inherited, NULL, 0.0f, 1);
...
}
...
// parent task
void DsaTaskA(rtk_p_task_control_block pTCB)
{
...
    rtk_trigger_interrupt(S_SOFTTASK, 0);
...
// inherited task
void DsaTaskSoft(rtk_p_task_control_block pTCB)
{
...
    /* task code */
    TASKSOFT_STEP_FUNCTION();
...
}
...
```

# Task Overrun Handling

**Overrun Situation**

An overrun situation occurs if a task is requested to start but has not finished its previous execution yet. Consider the situation shown below:




There is no overrun between the first and the second timer interrupt, but if aperiodic task $T_{HW}$ (higher priority) needs to be calculated, there is not enough time for periodic timer task $T_{Timer}$ to finish before it is requested to start again. The scheduler detects if such an overrun situation occurs for a task.

**Overrun handling**

The processing time of a task is limited to the reoccurrence of its trigger event. If a task exceeds this limit an overrun situation is caused, for example, if a periodic task takes longer to complete than its rate. The RTK1603 provides the following methods of reacting to a task overrun:

| Overrun Handling | Reaction on Task Overrun Detection |
|---|---|
| Overrun counting | The system increments the overrun counter in the TCB of the task. |
| Overrun function | The system calls the task-specific overrun function and increments the overrun counter in the TCB of the task. |
| Overrun queuing | 1. The system increments the overrun counter in the TCB of the task and queues the task.<br>If the overrun counter exceeds the `ovc_maxcnt` limit, the system calls the task-specific overrun function.<br>2. The system executes the queued tasks sequentially according to the accumulated count. |

**Overrun function** The overrun function is used to define a reaction to an overrun situation. The function pointer is stored in the TCB and therefore task-specific.

# How to Implement Overrun Handling

**Objective**

If a task overrun situation needs to be handled in a specific way, you must implement task-specific overrun handling.

**RTK1603**

The following C function of the RTK1603 is used for the implementation:

| Function | Used For ... |
|---|---|
| `rtk_create_task()` | Task handling |

For a description of the task handling functions refer to *Task Handling* in the *RapidPro System – Prototyping ECU Based on MPC5554 RTLib Reference*.

**Preconditions**

You must have implemented the task which overrun handling is to be implemented for, that is, the `rtk_create_task()` function for the task already exists in the main routine.

**Method**

**To implement overrun handling**

1. Declare the overrun function (depends on `rtk_create_task()` in step 3):
2. Specify the `ovc_check` parameter in the `rtk_create_task()` function of the task. This parameter specifies the overrun handling. method. Use one of the following predefined symbols:

| Overrun Handling | Predefined Symbol |
|---|---|
| Overrun counting | `ovc_count` |
| Overrun function | `ovc_fcn` |
| Overrun queuing | `ovc_queue` |

3. Implement the overrun function. The function is already referenced by the `ovc_fcn` parameter in:

```
pMyTask = rtk_create_task(fcn, priority, ovc_check,
ovc_fcn, ovc_maxcnt, user_data);
```

4 Specify the number of permitted overruns `ovc_maxcnt`. This parameter is evaluated if the `ovc_check` parameter is specified as `ovc_queue`, otherwise it is ignored. If the counter exceeds the limit, the overrun function is carried out.

**Result** You have implemented overrun handling.

Suppose you want to implement overrun handling for the `DsaTaskA` task as follows:

| Task Parameter | Implementation |
|---|---|
| Overrun handling method | `ovc_fcn` |
| Overrun function | `DsaTaskAOverrunFcn` |
| Number of permitted overruns | `INT_MAX` |

The resulting C code can look like the code fragment shown below, which is an adapted excerpt from the `RTFrameDS1603.c` dSPACE demo file.

```
...
void main(void)
{
...
    s_pTaskATCB = rtk_create_task(DsaTaskA, 12, ovc_fcn, DsaTaskAOverrunFcn, INT_MAX, 0);
...
}
...
void DsaTaskAOverrunFcn(rtk_p_task_control_block pTCB)
{
    /* task overrun code */
    s_TaskAOverrunCnt++;
}
...
```

# Implementing the dSPACE Calibration and Bypassing Service

**Objective**

To access the variables of your application from within CalDesk, you have to implement the dSPACE Calibration and Bypassing Service.

**Where to go from here**

Information in this section

| |
|---|
| ***Basics on the dSPACE Calibration and Bypassing Service*** **on page 137**<br>When implementing the dSPACE Calibration and Bypassing Service, you need some background information. |
| ***Basics on Implementing Memory Pages*** **on page 138**<br>To calibrate the parameters of your application via CalDesk, you have to define and configure memory pages and implement a pointer-based paging mechanism. |

| |
|---|
| ***Requirements for the Variable Descriptions of the RapidPro System*** **on page 141**<br>To access your RapidPro system via CalDesk, you have to provide an A2L file that matches the application running on your RapidPro system. |
| ***How to Implement the dSPACE Calibration and Bypassing Service*** **on page 142**<br>To use the dSPACE Calibration and Bypassing Service, you have to define memory pages, configure the access to the memory pages, and start the dSPACE Calibration and Bypassing Service. |
| ***Example of the dSPACE Calibration and Bypassing Service in the Demo Application*** **on page 147**<br>You can find suggestions for implementing the dSPACE Calibration and Bypassing Service in the demo application. |

# Basics on the dSPACE Calibration and Bypassing Service

**Objective**

When implementing the dSPACE Calibration and Bypassing Service in your application, you need some background information.

**dSPACE Calibration and Bypassing Service**

The dSPACE Calibration and Bypassing Service is used to control communication between an ECU (in our case the RapidPro system used as prototyping ECU) and a calibration tool. It provides access to the ECU application and resources for calibration and measurement. You have to integrate it into your application.

For feature-oriented information on the dSPACE Calibration and Bypassing Service, refer to the *dSPACE Calibration and Bypassing Service Feature Reference*.

**Handling the dSPACE Calibration and Bypassing Service**

The RTLib1603 provides functions for handling the dSPACE Calibration and Bypassing Service.

The `dsecu_service` function, which starts the foreground service of the dSPACE Calibration and Bypassing Service, and the `RTLIB_BACKGROUND_SERVICE` function, which calls the essential functions in the model background loop, for example, the background service of the dSPACE Calibration and Bypassing Service, need to be integrated into your application.

The other functions for handling the dSPACE Calibration and Bypassing Service are encapsulated and do not need to be called explicitly.

**Function overview**

The `dsECUSvc.h` header file contains all function declarations, global variables, and required data types. Refer to *dsecu_service* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

# Basics on Implementing Memory Pages

**Objective**

You can use CalDesk for calibration and measurement (data acquisition). To calibrate the parameters of your application via CalDesk, you have to define and configure memory pages and implement a pointer-based paging mechanism. If you want to use CalDesk only for measurement, memory pages are not necessary.

**Memory pages**

Memory pages are memory segments that are used for parameter calibration. Calibrating the parameters of an application requires 2 memory pages, each containing a complete set of parameters of the application running on an ECU (in our case the RapidPro system used as prototyping ECU). One of the pages is the *working page*, the other is the *reference page*. They are switchable via CalDesk:

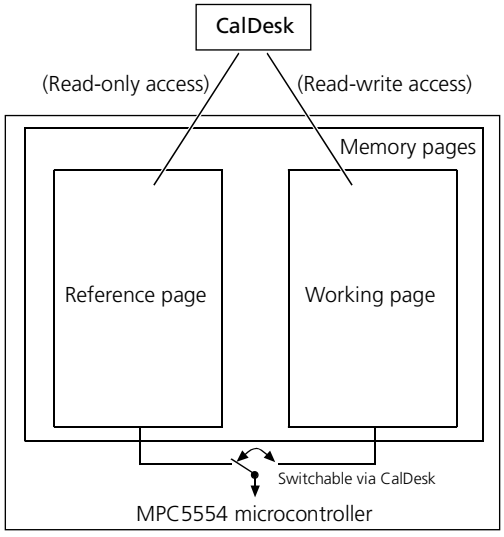
**Working page** The working page contains a complete set of parameters of the application. You can change the values of the parameters with a calibration tool. The working page allows you to perform parameter calibration.

**Reference page** The reference page also contains a complete set of parameters of the application, but you cannot change their values. The page serves as a read-only reference, which allows you to run the RapidPro system with a proven set of parameters.

You create the memory pages by declaring variables for all the parameters of your application. It is advisable to integrate them in suitable struct variables.

If you declare the variables for the memory pages in a separate module, they are all located in the same memory section. You can use the `#pragma option -NCECPREFPAGE` and `#pragma option -NDECPWORKPAGE` preprocessor directives to link the memory areas for the reference and working pages into the ECPREFPAGE and ECPWORKPAGE memory segments. If the structure for the reference page is declared as constant as well, this makes sure that the start address of the reference page stays the same even when your application is modified.

If you do not use the preprocessor directives, the memory pages are located at undefined memory addresses.




You can switch from one page to the other by activating the desired page in CalDesk. For further information, refer to *How to Activate the Working or Reference Data Set* in the *CalDesk Calibration Guide*.

Your application must include a pointer-based paging mechanism that performs the actual page switch on the RapidPro system every time it is requested by CalDesk.

**Pointer-based paging mechanism**

To make sure that only one of the memory pages is "visible" to the microcontroller at a time, you have to implement a pointer-based paging mechanism.

The paging mechanism directs the access to one of the memory pages. Page switching is done by redirecting the pointer to the other page.

To implement the pointer-based paging mechanism, you have to make the memory segments of the reference and working pages known to the system by describing them via memory descriptors, and specify a callback function that performs the actual page switch on the RapidPro system.

**Memory descriptor** The memory pages are specified by their start and end addresses, their access modes, and the mode that specifies the reference page or working page. For further information, refer to *dsconfig_memory_descriptor_t* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**Callback function** You have to implement a function that performs the actual page switch on the RapidPro system. The most effective way to do so is a mechanism that redirects a global access pointer from one memory page to the other. This function is called when pages are switched in CalDesk.

For an example of the pointer-based paging mechanism, refer to the example in *How to Implement the dSPACE Calibration and Bypassing Service* on page 142, or *Example of the dSPACE Calibration and Bypassing Service in the Demo Application* on page 147.

### Memory management with the RTLib1603

The RTLib1603 provides functions for registering and configuring memory pages, implementing the pointer-based paging mechanism, and accessing specified memory blocks of a paged memory area.

### Function overview

The `dsECUmem.h` header file contains all function declarations, global variables, and required data types. For detailed information on functions for registering and configuring memory pages, refer to *Host Service* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

# Requirements for the Variable Descriptions of the RapidPro System

**Objective**

To access your RapidPro system with CalDesk's RapidPro device, you have to provide a suitable A2L file. Make sure that your A2L file matches your application each time your application is modified.

An ASAM-MCD 2MC (A2L) file – formerly known as an ASAP2 file – lists the variables and parameters of an electronic control unit (ECU).

**ASAM-MCD 2MC specification**

ASAM e.V. (Association for Standardisation of Automation- and Measuring Systems e.V.) has specified the ASAM-MCD 2MC file format. You can download this specification and further information on the ASAM-MCD standard from http://www.asam.net.

**A2L file**

An A2L file contains information on the parameters and measurement variables of the application that is implemented on an ECU (in our case: the real-time application running on the RapidPro system used as prototyping ECU).

The A2L file's IF_DATA element for the RapidPro device has to be the same as the one used for the DCI-GME1 (dSPACE Calibration Interface – Generic Memory Emulator). For details on the required IF_DATA element, refer to *AML Definitions for the DCI-GME1* in the *dSPACE Calibration and Bypassing Service Implementation* document.

**A2L file generator**

If you need an A2L file generator (editor), contact dSPACE for further information.

# How to Implement the dSPACE Calibration and Bypassing Service

**Objective**

To use the dSPACE Calibration and Bypassing Service for calibration and measurement, you have to create memory pages, configure the access to the memory pages, and start the dSPACE Calibration and Bypassing Service.

**Functions used in the demo application**

The following instruction deals with the functions which are used for registering and configuring memory pages, and implementing the dSPACE Calibration and Bypassing Service in the demo application.

There are further functions for implementing memory pages, for example, for accessing a specified number of bytes of a paged memory area. For an overview, refer to *Host Service* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Hardware preconditions:
  - For communicating with CalDesk on the host PC, the RapidPro system used as a prototyping ECU must be equipped with a COM-USB-PI 1/1 communication module (DS1607).
- Software preconditions:
  - Your application must contain at least the definitions required for the main routine. Refer to *Creating the Main Routine of Your Application* on page 75.

- For building your application, the following header files must be included in your source code:

| Header File | Used For ... |
|---|---|
| dsECUmem.h | Memory management [1] |
| dsecusvc.h | ECU service code for calibration and bypassing |

1) Included automatically in your application, if you include brtenv.h.

The header files are located in %DSPACE_ROOT%\DS1603\RTLib.

**Method** **To implement the dSPACE Calibration and Bypassing Service**

1. Define a struct type for the memory pages in a separate module. A variable of this type has to hold all the calibratable variables of the application, so it must include all the necessary struct elements beforehand.
2. Use the #pragma option -NCECPREFPAGE preprocessor directive to link the memory area for the reference page into the ECPREFPAGE memory segment, and declare a struct variable of the type defined in step 1. Initialize it with suitable values.
3. Use the #pragma option -NDECPWORKPAGE preprocessor directive to link the memory area for the working page into the ECPWORKPAGE memory segment, and declare a struct variable of the type defined in step 1.

   You have created the reference and working pages.
4. Declare a variable of dsconfig_memory_descriptor_t type, which is used to provide the pointer-based paging mechanism with the information needed for each memory page.
5. Assign the parameters of the reference page, such as start and end addresses, access, and mode to the struct members of this variable.

   To prevent writing on the reference page, the access should be set to DSCONFIG_ACCESS_READ. The mode of the reference page has to be DSCONFIG_STATE_BASE.

6 Make the reference page known to the pointer-based paging mechanism by using `rapidpro_mem_page_segment_information_set` with 0 as the page number and the `dsconfig_memory_descriptor_t` type variable from step 5.

7 Assign the parameters of the working page, such as start and end addresses, access, and mode to the struct members of the `dsconfig_memory_descriptor_t` type variable.
The access of the working page should be set to `DSCONFIG_ACCESS_RW`. The mode of the working page can be set to `DSCONFIG_STATE_UNINITIALIZED` or `DSCONFIG_STATE_INITIALIZED`.

8 Make the working page known to the pointer-based paging mechanism by using `rapidpro_mem_page_segment_information_set` with 1 as the page number and the `dsconfig_memory_descriptor_t` type variable from step 7.

You have made the memory pages known to the pointer-based paging mechanism.

9 Implement a function that switches the memory pages using `rapidpro_mem_access`.

10 Specify this function as a callback function which is called when a page switch is performed in CalDesk using `rapidpro_mem_page_switch_fctn_set`.

You have implemented the pointer-based paging mechanism.

11 Ensure the communication between your RapidPro system and CalDesk calling cyclicly `RTLIB_BACKGROUND_SERVICE`.

12 Start data acquisition (DAQ) using `dsecu_service` at the end of a task.

**Result** When you run your application on your RapidPro system, the memory pages can be accessed and switched by CalDesk.

The following code pattern shows how 2 memory pages, 1 read-only reference page and 1 working page with read/write access, and the pointer-based paging mechanism can be implemented.

```
/* Demo application for page handling with global access pointer */
#include <brtenv.h>
#include <dsECUsvc.h>
```

```
/* Predefined Values */
#define REFERENCE_PAGE_DATA {0xFF,0x73,0xAD,0x0}

/* Calibration data structure */
typedef struct _calDataStruct_t
{
  UInt32 variableA;
  UInt32 variableB;
  UInt32 variableC;
  UInt32 variableD;
}calDataStruct_t;

/* Calibration working, reference and access elements */
volatile const calDataStruct_t   referencePage  = REFERENCE_PAGE_DATA;
volatile       calDataStruct_t   workingPage;
volatile const calDataStruct_t * calDataAccessP = &referencePage;

/* Result for custom function */
volatile UInt32 globalResult = 0;

/* Callback function performing page switching */
void pageSwitchCallback(UInt8 pageId)
{
  UInt32 int_status;

  /* Disabling interrupts to avoid inconsistent pointer values */
  RTLIB_INT_SAVE_AND_DISABLE(int_status);

  /* Redirecting global access pointer to active page */
  calDataAccessP = &referencePage;
  rapidpro_mem_access(DSCONFIG_ACCESS_READ,
  (void **)&calDataAccessP, sizeof(calDataStruct_t));

  /* Restoring interrupts, again */
  RTLIB_INT_RESTORE(int_status);
  msg_info_printf(MSG_SM_USER,0,"Page switched to pageId %d",pageId);
}

/* Custom function accessing calibration data */
UInt32 customFunction()
{
  return calDataAccessP->variableA;
}
/* main */
void main (void)
{
  /* Initializing DS1603 services */
  init();

  /* Declaring reference page */
  {
    dsconfig_memory_descriptor_t mem_descriptor;
    mem_descriptor.start_address = (UInt32) &referencePage;
    mem_descriptor.end_address =
      mem_descriptor.start_address + sizeof(calDataStruct_t) - 1;
    mem_descriptor.access = DSCONFIG_ACCESS_READ;
    mem_descriptor.mode = DSCONFIG_STATE_BASE;
    rapidpro_mem_page_segment_information_set(0,mem_descriptor);
  }
```

```
  /* Declaring working page */
  {
    dsconfig_memory_descriptor_t mem_descriptor;
    mem_descriptor.start_address = (UInt32) &workingPage;
    mem_descriptor.end_address =
      mem_descriptor.start_address + sizeof(calDataStruct_t) - 1;
    mem_descriptor.access = DSCONFIG_ACCESS_RW;
    mem_descriptor.mode = DSCONFIG_STATE_UNINITIALIZED;
    rapidpro_mem_page_segment_information_set(1,mem_descriptor);
  }

  /* Specifying page switch callback function */
  rapidpro_mem_page_switch_fctn_set(pageSwitchCallback);
  msg_info_printf(MSG_SM_USER,0,"Application started");

  /* Processing main executions */
  while (1)
  {
    /* call RtLib background service
      (consisting of dci_ecu_rapidpro_bckgrnd and dsconfig_background */
    RTLIB_BACKGROUND_SERVICE();

    /* Processing custom function using calibration data */
    {
      static UInt32 prescaler = 0;
      if (++prescaler > 1000)
      {
        prescaler = 0;
        globalResult = customFunction();
        /* transfering calibration data on channel 1 */
        dsecu_service(1);
      }
    }
  } /* while (1) */
}
```

**Further example**

Another example shows how memory pages and the dSPACE Calibration and Bypassing Service are implemented in the demo application. Refer to *Example of the dSPACE Calibration and Bypassing Service in the Demo Application* on page 147.

# Example of the dSPACE Calibration and Bypassing Service in the Demo Application

**Objective**

You can find suggestions for implementing the dSPACE Calibration and Bypassing Service in the demo application. Memory pages and the dSPACE Calibration and Bypassing Service are implemented in various modules of the demo application.

**Structure for memory pages**

The following code pattern is an excerpt from the `ECUCode_CalParams.h` module, which is linked to the main routine. It shows how the structure for implementing the reference and working pages is defined.

```
typedef struct tagEcpPage_tp {
...
   const volatile Float32 CounterGain1ms;
...
   const volatile Bool StopTaskA;
...
} EcpPage_tp; /* Type of structure for data pages. */
```

**Linking memory segments**

The following code pattern is an excerpt from the `ECUCode_CalParams.c` module, which is linked to the main routine. It shows how the memory areas for the reference and working pages are linked into the memory segments.

```
#pragma option -NCECPREFPAGE
...
const volatile EcpPage_tp EcpRefPage = {
...
   2.F /* CounterGain1ms:  */,
...
   0 /* StopTaskA:  */,
...
};

#pragma option -NDECPWORKPAGE
...
volatile EcpPage_tp EcpWorkPage;
```

### Page switching in the ECU code

The following code pattern is an excerpt from the `ECUCode.c` module, which is linked to the main routine. It shows how page switching is implemented in the ECU code and values are read from the active memory page, i.e., the memory page that the access pointer is directed to.

```
...
UInt8 g_EcpCurrentPage = 0;
EcpPage_tp * g_pEcpActivePage;

void RESTART_ECUCode(void)
{
   EcpSwitchPage();
}

void EcpSwitchPage(void)
{
   if (g_EcpCurrentPage == 1) {
      g_pEcpActivePage = (EcpPage_tp *) &(EcpWorkPage);
   }
   else {
      g_pEcpActivePage = (EcpPage_tp *) &(EcpRefPage);
   }
}
void TaskA(void)
{
...
   static Float32 X_Sa15_Memory1 = 0.F;
...
   /* Outport: ECUCode/1ms task/CounterGain1ms_out.
   # combined # Gain: ECUCode/1ms task/CounterGain1ms */
   CounterGain1ms_out =
      X_Sa15_Memory1 * g_pEcpActivePage->CounterGain1ms;
...
}
```

### Pointer-based paging mechanism

The following code patterns are excerpts from the RTFrameDS1603.c module. They show how the pointer-based paging mechanism is configured and how page switching is implemented in the task frame. Values are read from the memory page and the dSPACE Calibration and Bypassing Service is called in a timer task.

```
...
#define TASKA_STEP_FUNCTION()       TaskA()
...
void main(void)
{
...
    // memory descriptor for config of cal service
    dsconfig_memory_descriptor_t  mem_descriptor;
...
    /* set switch function for memory pages */
    rapidpro_mem_page_switch_fctn_set(DsaSwitchECPPage);

    /* build up page config (ref page) */
    mem_descriptor.start_address=
        (rapidpro_ecu_mem_address_t) &EcpRefPage;
    mem_descriptor.end_address  =
        mem_descriptor.start_address + sizeof(EcpPage_tp);
    mem_descriptor.access   = DSCONFIG_ACCESS_READ;
    mem_descriptor.mode     = DSCONFIG_STATE_BASE;
    rapidpro_mem_page_segment_information_set(0,mem_descriptor);

    /* build up page config (work page) */
    mem_descriptor.start_address=
        (rapidpro_ecu_mem_address_t) &EcpWorkPage;
    mem_descriptor.end_address  =
        mem_descriptor.start_address + sizeof(EcpPage_tp);
    mem_descriptor.access   = DSCONFIG_ACCESS_RW;
    mem_descriptor.mode     = DSCONFIG_STATE_UNINITIALIZED;
    rapidpro_mem_page_segment_information_set(1,mem_descriptor);
...
}
```

```
// handling of page switching
void DsaSwitchECPPage(UInt8 page_number)
{
    // set current page
    g_EcpCurrentPage = page_number;
    // call TargetLink generated function to switch
    EcpSwitchPage();
}

// taska
void DsaTaskA(rtk_p_task_control_block pTCB)
{
...
    if( !g_pEcpActivePage->StopTaskA )
    {
        /* task code */
        TASKA_STEP_FUNCTION();
    }
...
    /* DSECU service 1 */
    dsecu_service(1);
}
```

# Implementing I/O Access

**Where to go from here**

Information in this section


| |
|---|
| ***Overview of Supported I/O Features*** **on page 152** |
| ***General Concepts of Implementing I/O Features*** **on page 167** |
| ***How to Implement A/D Conversion*** **on page 171** |
| ***How to Implement Bit I/O on the I/O PLD*** **on page 176** |
| ***How to Implement Digital Inputs on the eTPU*** **on page 179** |
| ***How to Generate 1-Phase PWM Signals*** **on page 183** |
| ***How to Measure Frequencies of PWM Signals*** **on page 188** |

# Overview of Supported I/O Features

**Objective**

The RTLib1603 supports A/D conversion, bit I/O access, and timing I/O features (for PWM signal generation and PWM signal measurement).

**Where to go from here**

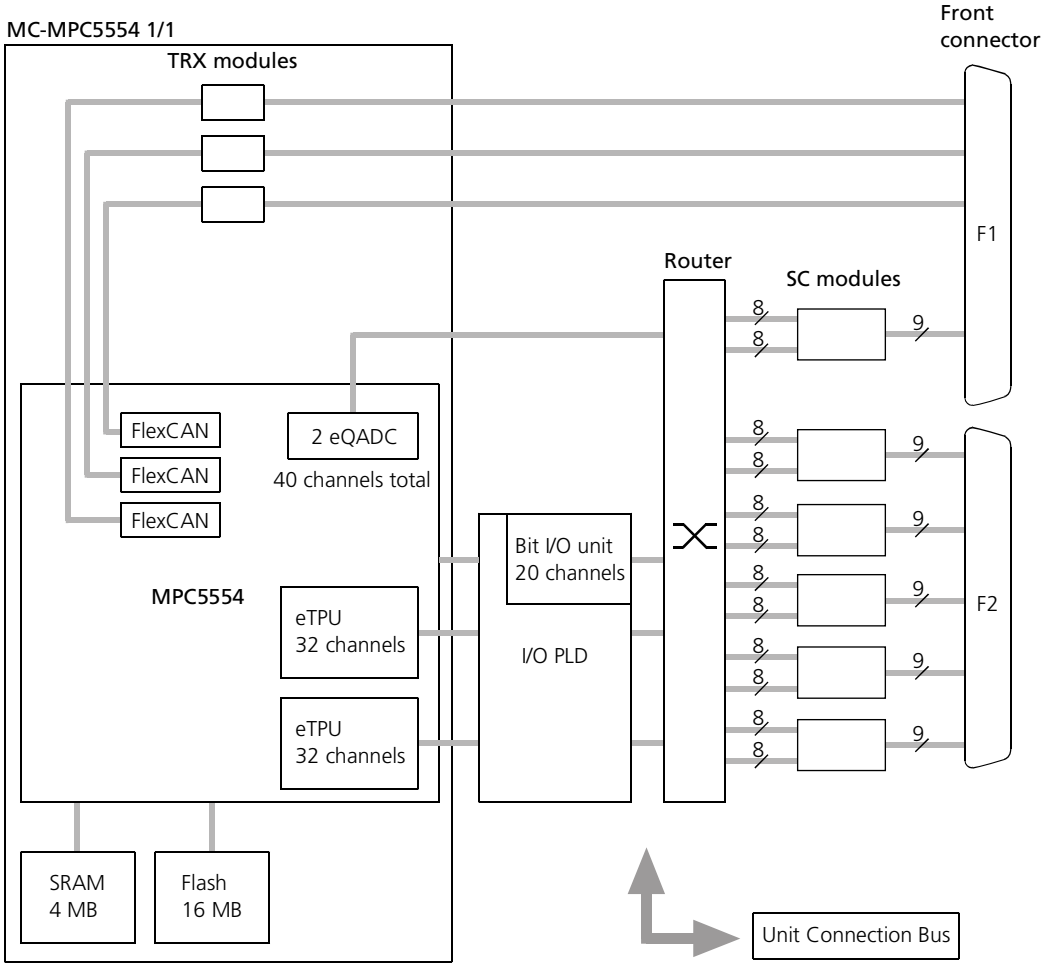
Information in this section


| |
|---|
| ***Features of the RapidPro Control Unit Based on MPC5554*** **on page 153** |
| ***A/D Conversion*** **on page 156** |
| ***Bit I/O*** **on page 160** |
| ***1-Phase PWM Signal Generation*** **on page 162** |
| ***PWM Signal Measurement*** **on page 165** |

# Features of the RapidPro Control Unit Based on MPC5554

**Hardware components**

The illustration below shows the hardware components which support the I/O features provided by the RTLib1603. The illustration also shows the signal routing to the I/O connectors of the RapidPro Control Unit.



The MPC5554 microcontroller from Freescale is installed on the MC-MPC5554 1/1 microcontroller module. Further implementation-relevant hardware components (I/O PLD) are located on the Control Unit.

**Characteristics**

Hardware version 3.1.0 of the MC-MPC5554 1/1 is equipped with the **MPC5554 Rev. A** microcontroller. You can read out the hardware version number via ConfigurationDesk.

The table below lists implementation-relevant characteristics of the RapidPro Control Unit based on MPC5554.

| Component | Characteristic |
|---|---|
| Processor clock | 112 MHz clock rate |
| Memory [1)] | • 4 MB SRAM<br>• 16 MB flash memory |
| eTPU (enhanced time processor unit) | 2 independent eTPUs with:<br>• 32 channels each<br>• 58 channels can be used for PWM signal measurement or as digital input<br>• 64 channels can be used for PWM signal generation.<br>• For further details, see *Basics on Using the Time Processing Unit (eTPU)* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*. |
| eQADC (enhanced queued A/D converters) | 2 independent eQADCs with:<br>• 40 A/D converter channels total<br>• 12-bit resolution<br>• 10-bit accuracy at 400 ksamples/s<br>• 8-bit accuracy at 800 ksamples/s |
| I/O PLD [2)] | • 20 digital channels<br>• Can be configured individually as input or output |

| Component | Characteristic |
|---|---|
| FlexCAN controller | • 3 FlexCAN controller<br>• Supporting CAN 2.0B standard<br>• For each CAN channel (controller) different transceiver modules (TRX module) can be mounted on the MPC5554 1/1 module 3)<br>• For CAN feature details, refer to *Implementing CAN Communication* on page 194. |

1) SRAM and flash memory do not belong to the MPC5554. They are installed on the microcontroller module.

2) The programmable logic device (PLD) does not belong to the MC-MPC5554 1/1 module. It is installed on the carrier board of the Control Unit. The I/O PLD is used to enlarge the functionality of the MPC5554. It is programmed by dSPACE. You can access the PLD's bit I/O unit via RTLib functions.

3) Slots for holding the bus transceiver modules are installed on the MC-MPC5554 1/1 microcontroller module.

**Restrictions of available features**

Because of the modular concept of the RapidPro hardware, you can use only features of the microcontroller that are supported by the signal conditioning modules and/or power stage modules installed on the units of your RapidPro system.

For example, if you want to implement A/D conversion, the RapidPro system must contain modules providing analog channels. If no suitable modules are available, you cannot use the A/D conversion feature of the Control Unit's microcontroller.

# A/D Conversion

**Objective**

The MPC5554 provides two enhanced queued analog-to-digital converters (eQADC) which are used for A/D conversion.

**Characteristics**

Each eQADC can access all 40 channels of the analog inputs. Each converter can handle 3 command queues. The following table shows implementation-relevant characteristics valid for each of the two eQADCs.

| Characteristics | eQADC |
|---|---|
| Number of channels | • 40 channels total<br>• 3 command queues |
| Trigger sources | • Queue 1 and Queue 2 can be started by an external trigger or via software trigger<br>• Queue 3 can be started only via software trigger |
| Interrupt | • Support of end-of-conversion interrupts for each queue<br>• Can be enabled or disabled |
| Resolution | 12-bit |
| Conversion time | • 2.5 µs at 10-bit accurancy<br>• 1.25 µs at 8-bit accurancy |
| Sample time | • 2, 8, 64 or 128 clock cycles<br>• Configurable for each queue item |

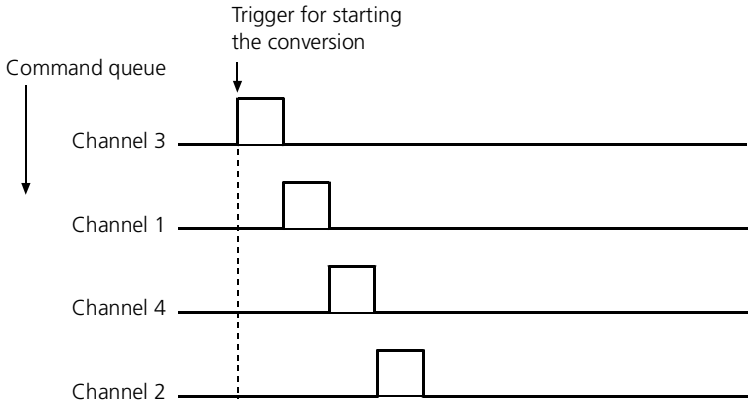
All other technical characteristics, for example, the input voltage range, depend on the signal conditioning module used.

**Conversion process**

The two A/D converters execute a specified number of conversions starting with a single trigger. You can configure the number and the sequence of the A/D channels to be used in the ADC queue structure. You can implement the A/D channels in unsorted order. For example, you can implement the following channel list: 3, 9, 36, 31.

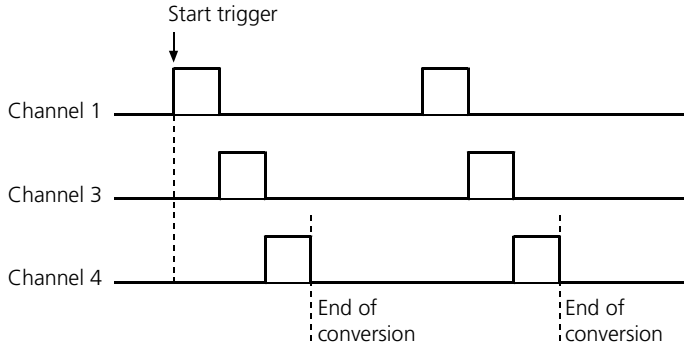
Do not use the same channel more than once. Each channel can be sampled only once in the same ADC queue structure.

The trigger starts A/D conversion according to the implemented channel list. The A/D channels are switched from one channel to the next each time a conversion is completed. There is no additional trigger required to start the conversion on the current channel.



**End of conversion behavior**

The execution of the conversions specified in the command queue starts with a trigger event. The A/D converter stops at the end of conversion, which is reached when the end of the conversion command queue is processed. If you enabled interrupt generation, the end-of-conversion interrupt is set and the microcontroller is triggered to read the conversion result implementing the appropriate RTLib function. The A/D converter waits for the next trigger event to start the next conversion. See also the illustration below.




**Trigger modes**

The start of A/D conversion can be triggered as follows:

- Software trigger mode

  The conversion starts immediately after the corresponding RTLib function was executed.

- External trigger mode

  A/D conversion waits until an external trigger occurs, for example, a falling or rising edge of an external signal.

  After every trigger event, you have to activate the A/D conversion again (using the corresponding RTLib function), before the next external trigger event can be recognized.

**Applicable modules**

To use the A/D conversion feature, the RapidPro system must contain at least one of the following modules:

- SC-AI 4/1
- SC-AI 10/1
- Any other module providing analog signals

For a list of available modules, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.

**Further details**

- You can access the A/D conversion features via RTLib functions. For an overview of the functions available, refer to *A/D Conversion* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.
- For detailed instructions on implementing, refer to *How to Implement A/D Conversion* on page 171.

# Bit I/O

**Objective**

You can use functions provided for the eTPUs (on MPC5554) and for the I/O PLD (on the carrier board of the Control Unit) for bit I/O access.

**Characteristics**

Some implementation-relevant characteristics of the two components differ:

| Characteristics | eTPU | I/O PLD |
|---|---|---|
| Number of channels | • eTPU_A: 26 channels (channels 1 … 24, channels 31, 32) [1)]<br>• eTPU_B: 32 channels | 20 channels |
| I/O access | Bit-wise | Bit-wise |
| Input/output | **Input only** | Input and output |
| Power-up default | All channels are disabled. | All channels are configured as input. |
| Interrupt generation | Interrupt generation on falling, rising, or both edges of the input signal. | No |

1) Channels 25 … 30 are output channels only.

All other characteristics of bit I/O access, like the threshold voltage range, depend on the signal conditioning modules used.

**Applicable modules**

To use the bit I/O feature, your RapidPro system must contain at least one of the following modules:

- SC-DI 8/1 for handling digital input signals
- SC-DO 8/1 for handling digital output signals
- Any other module providing digital signals

For a list of available modules, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.

**Further details**

- You can access the bit I/O features via RTLib functions. For an overview of the functions available, refer to:
  - *Bit I/O on the I/O PLD* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*
  - *Bit I/O on the eTPU* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*

- For detailed instructions on implementing bit I/O, refer to:
  - *How to Implement Bit I/O on the I/O PLD* on page 176
  - *How to Implement Digital Inputs on the eTPU* on page 179

# 1-Phase PWM Signal Generation

**Objective**

The MPC5554 provides two enhanced time processor units (eTPU) which can be used for PWM signal generation.

**Characteristics**

The implementation-relevant characteristics of an eTPU used for PWM signal generation are listed below:

| Characteristics | eTPU |
|---|---|
| Number of channels | 32 channels (on each eTPU) |
| Timer | 2 timers |
| Prescaler values [1)] | • Timer 1: 2 … 512 (256 steps available)<br>• Timer 2: 8 … 512 (64 steps available) |
| Frequency ranges [1)] | • Min.: 0.03 Hz … 1.71 kHz<br>• Max.: 6.68 Hz … 437.50 kHz |
| Pulse width ranges [1)] | • Min.: 2.29 µs … 149.80 µs<br>• Max.: 585.14 µs … 38347.92 ms |
| Interrupt generation | Interrupt rate within the range 1 … 127 periods |

1) For basics on prescaling and detailed values, refer to *Prescaler Settings* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

The frequency and the duty cycle can also be adjusted during run time using the corresponding RTLib functions.

**PWM basics**

A PWM signal is characterized by its period, duty cycle, and polarity.

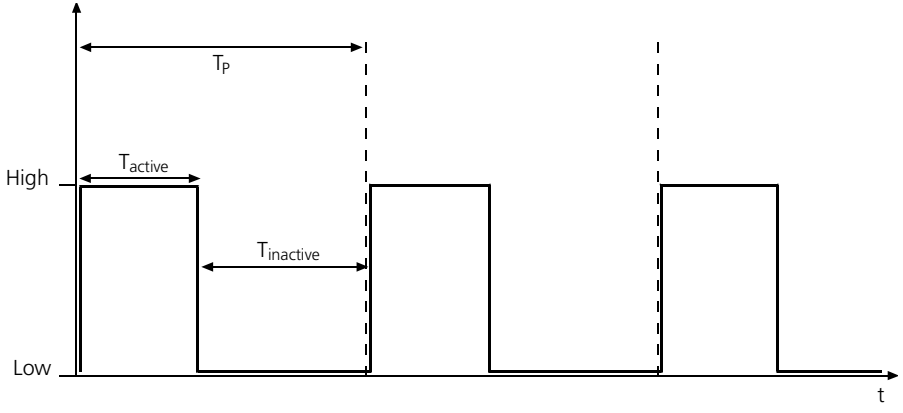
**PWM period**

For PWM signals, you can specify the PWM period $T_P$ (= $T_{active}$+ $T_{inactive}$) in a range that corresponds to the specified timer and prescaler values. Each of the PWM output channels can have different values for the PWM period.

The ranges are theoretical values. In practice, the values depend on the SC and PS modules used. For further information, refer to the module data sheet in the *RapidPro System – Installation and Configuration Reference*.

**Duty cycle**

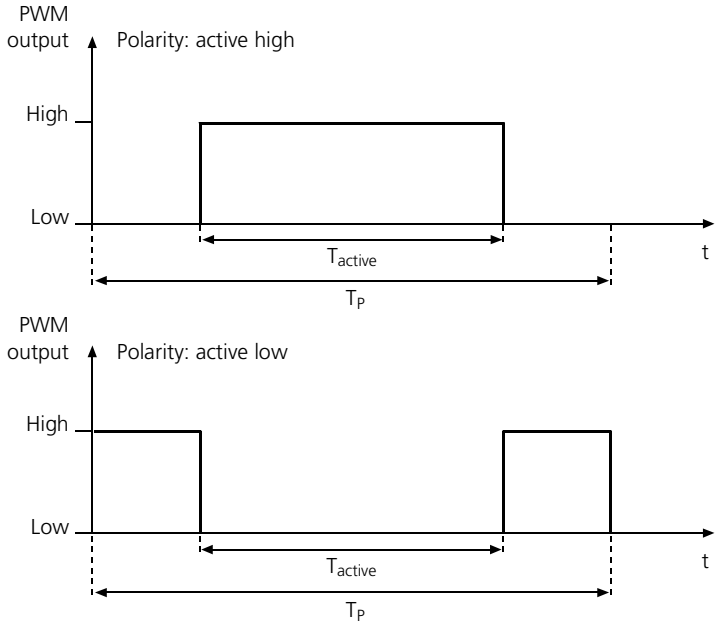
You can specify the duty cycle, which is defined as d = $T_{active}$/ $T_P$. The available duty cycle range is 0 … 1 (0 … 100 %).




**Polarity**

The polarity is specified by a function parameter and the configuration of the SC and PS modules used. It can happen that the module configuration using ConfigurationDesk inverts the specified signal polarity again.

The following illustration shows how the duty cycle ($T_{active}/T_p$ ratio) and polarity are defined.




**Applicable modules**

To use the PWM signal generation feature, your RapidPro system must contain at least one of the following modules:

- SC-DO 8/1
- A PS module, if your RapidPro system contains a Power Unit
- Any other module providing digital output signals

For further information, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.

**Further details**

- You can access the PWM signal generation features via RTLib functions. For an overview of the functions available, refer to *1-Phase PWM Signal Generation* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.
- For detailed instructions on implementing, refer to *How to Generate 1-Phase PWM Signals* on page 183.

# PWM Signal Measurement

**Objective**

The MPC5554 provides two enhanced time processor units (eTPU) which can be used to measure frequencies of PWM signals.

**Characteristics**

The implementation-relevant characteristics of a eTPU used for PWM signal measurement are listed below:

| Characteristics | eTPU |
|---|---|
| Channel number | • eTPU_A: 26 channels (channels 1 … 24, channels 31, 32) [1)]<br>• eTPU_B: 32 channels |
| Timer | 2 timers |
| Prescaler values [2)] | • Timer 1: 2 … 512 (256 steps available)<br>• Timer 2: 8 … 512 (64 steps available) |
| Frequency ranges [2)] | • Min.: 0.03 Hz … 1.71 kHz<br>• Max.: 6.68 Hz … 437.50 kHz |
| Interrupt generation | Yes |
| Signal polarity | A high active PWM signal or a low active PWM signal can be specified. |

1) Channels 25 … 30 are output channels only an can therefore not be used for PWM signal measurement.

2) For basics on prescaling and detailed values, refer to *Prescaler Settings* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

**Measurement features**

With the PWM2D function, you can measure the frequency of a connected PWM signal and calculate the corresponding duty cycle.

**Behavior at frequencies outside the range**

If the frequency of the measured signal is below the lower limit of the specified frequency range, a value of 0 Hz is returned. The duty cycle then follows the input signal:

- If you measure a high active PWM signal, the duty cycle is 1 for a high input signal.
- If you measure a low active PWM signal, the duty cycle is 1 for a low input signal.

Whether you measure high active or low active PWM signals depends on the configuration of the SC modules used.

If the frequency of the measured signal is above the upper limit of the specified frequency range, the value of the detected frequency is unpredictable.

**Applicable modules**

To use the PWM signal measurement feature, your RapidPro system must contain at least one of the following modules:

- SC-DI 8/1
- Any other module providing digital input channels

For further information, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.

**Further details**

- You can access the PWM signal measurement features via RTLib functions. For an overview of the functions available, refer to *PWM Signal Measurement (PWM2D)* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.
- For detailed instructions on implementing, refer to *How to Measure Frequencies of PWM Signals* on page 188.

# General Concepts of Implementing I/O Features

**Objective**

The driver organization of the supported I/O features is modular in design. The driver structures used for different features are divided into different layers. To use an I/O feature, you have to create, set, and apply its layers.

**Device driver structures**

The layers used for the I/O features are designed to separate device driver structures:

- A/D conversion is separated into three device driver structures:
  - The ADC layer providing functions for the physical ADC device, represented by the `DsS1603Adc` data structure
  - The ADC queue layer providing functions for queue management, represented by the `DsS1603AdcQueue` data structure. An A/D queue holds the conversion commands
  - The ADC queue item layer providing functions for the conversion of one specific ADC channel, represented by the `DsS1603AdcQueueItem` data structure
- The Time Processer Unit (eTPU) is divided into
  - The eTPU unit layer, represented by the `DsS1603Tpu` data structure
  - The eTPU channel layer

  The eTPU channel layer has three instances, according to the channel functionality:
  - PWM in channel, represented by the `DsS1603TpuPwmIn` data structure
  - PWM out channel, represented by the `DsS1603TpuPwmOut` data structure
  - Digital in channel, represented by the `DsS1603TpuDigIn` data structure

The bit I/O handling on the eTPU provides functions only for digital inputs.

- The bit I/O handling on the I/O PLD is provided by one device driver structure represented by the `DsS1603ExtBitIo` data structure.

**Create and apply**

To configure a device for an I/O feature, you have to create layer-specific data structures using the `create` functions in hierarchical order. The allocated data structures are used to parameterize the device drivers. Each allocated device structure has to be configured after the device-specific `create` function and before the device-specific `apply` function. To finalize the initialization of each I/O feature, you have to use the layer-specific `apply` function in reverse order.

All the information in a device structure can only be accessed by the device-specific `set` or `get` functions. Do not access any data in the device structure directly.

If you use the described I/O features, you have to create and apply them in reverse order. If you use CAN, you have to create and apply the required device driver structures in a sequence. For further information, refer to *Implementing CAN Communication* on page 194.

**Starting I/O devices**

Long-term functionality like bit I/O can be started with the `start` and `stop` functions. After starting the devices, you can use the `read` and `get` functions for input devices and the `set` and `write` functions for output devices. If you want to deactivate the specific device, you can use the `stop` functions. To trigger a functionality inside your task, like the queue of the ADC, you can use the `trigger` functions instead. For an overview of the functions to be used for I/O handling, refer to *Standard I/O* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

The following example shows how to use the `create` and `apply` functions to create, set, and apply the device drivers for A/D conversion:

```
…
// Create the device drivers by using the create functions:
   Ds1603Adc_create(&pAdc,DS1603_ADC_UNIT_1);
      Ds1603AdcQueue_create(&pQueue,pAdc,DS1603_ADC_QUEUE_1);
         Ds1603AdcQueueItem_create(&pItem_Ch1,pQueue,1);
         Ds1603AdcQueueItem_create(&pItem_Ch2,pQueue,2);
         Ds1603AdcQueueItem_create(&pItem_Ch10,pQueue,10);
         Ds1603AdcQueueItem_create(&pItem_Ch11,pQueue,11);
// Configure A/D conversion, the AD queue and queue item:
…
// Finalize the initialization by using the apply functions in
// reverse order:
         Ds1603AdcQueueItem_apply(pItem_Ch1);
         Ds1603AdcQueueItem_apply(pItem_Ch2);
         Ds1603AdcQueueItem_apply(pItem_Ch10);
         Ds1603AdcQueueItem_apply(pItem_Ch11);
      Ds1603AdcQueue_apply(pQueue);
   Ds1603Adc_apply(pAdc);
…
```

The following example shows you how to use the different functionalities of the eTPU:

```
// Create the eTPU device driver:
   Ds1603Tpu_create(&pTPU,1);
// Create the one eTPU channel as digital in, one as PWM in and
// one as PWM out:
      Ds1603TpuDigIn_create(&pDigIn,pTPU,1);
      Ds1603TpuPwmIn_create(&pPwmIn,pTPU,2);
      Ds1603TpuPwmOut_create(&pPWmOut,pTPU,3);
// Configure the channels of the eTPU:
…
// Finalize the initialization of the eTPU channels:
      Ds1603TpuPwmOut_apply(pPWmOut);
      Ds1603TpuPwmIn_apply(pPWmIn);
      Ds1603TpuDigIn_apply(pDigIn);
// Finalize the initialization of the eTPU device driver:
   Ds1603Tpu_apply(pTPU);
```

**Implementing I/O handling in your application**

It is recommended to initialize, configure, and set the device structures in the main routine of your application. All the functions that are used during the run time of your application can be used outside the main routine, for example, in a task routine. If you use a bit I/O feature with outputs of your RapidPro system, you have to enable the outputs and check the TopologyID before you can use the I/O feature. For further information, refer to *How to Create the Main Routine from Scratch* on page 78.

**I/O mapping**

The mapping to external pins on the I/O connectors of the RapidPro hardware depends on the types of the installed SC and PS modules and the slots on the unit they are inserted in. For example, to use a channel of the eTPU as digital output, you have to check if the output of the MPC5554 is mapped to an output pin of the system via a signal conditioning (SC) or power stage (PS) module. ConfigurationDesk lets you export the required pinout information to a file. For further information, refer to *Signal Mapping to I/O Pins* on page 24.

**Signal scaling**

To avoid measurement faults, ensure that the scaling of an input signal (provided by the configuration of the installed SC modules) matches the scaling which you expect.

# How to Implement A/D Conversion

**Objective**

A/D conversion is an element of most applications in rapid control prototyping, because sensors, for example for pressure or temperature, provide analog signals which have to be processed as digital values. To implement this feature in your application, you have to use the method described below.

**Basics**

- For an overview of the supported I/O features, refer to *Overview of Supported I/O Features* on page 152.
- For further information on the general concepts of implementing the I/O features, refer to *General Concepts of Implementing I/O Features* on page 167.

**Configuring A/D conversion**

To implement A/D conversion on the RapidPro System, you have to declare the following data structures provided by the RTLib:

- `DsS1603Adc`

  The structure represents an A/D converter.

- `DsS1603AdcQueue`

  The structure represents an A/D converter queue.

- `DsS1603AdcQueueItem`

  The structure represents an A/D queue item.

Each A/D converter has three converter queues. Each A/D queue can contain several items. Each queue item handles the conversion of one channel. You can sample the same channels in different queues.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Hardware preconditions:
  - To implement A/D conversion, your RapidPro system must contain SC or PS modules which provide analog signals, for example, SC-AI 4/1. Additionally, these analog signals must be routed to one of the 40 analog input channels of the MPC5554. To use the external trigger functionality, your RapidPro system must contain SC/PS modules for digital input signals of your hardware trigger. For a list of available modules, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.
- Software preconditions:
  - Your application must contain at least the definitions required for the main routine. For further information, refer to *Creating the Main Routine of Your Application* on page 75.
  - For building your application, the following header files must be included in your source code:

| Header File | Used For ... |
|---|---|
| `Ds1603Adc.h` | A/D converter handling |
| `Ds1603AdcQueue.h` | A/D converter queue handling |
| `Ds1603AdcQueueItem.h` | A/D converter queue item handling |

The header files reside in `%DSPACE_ROOT%\DS1603\RTLib`.

If you include `brtenv.h`, these header files are included automatically in your application.

Steps 1 - 7 must be implemented in the main routine directly after initialization of the RapidPro system.

**Method** **To implement A/D conversion**

1. Declare the variables to be used for the data structures of the A/D converter, the A/D converter queue, and the items of the queue.

   If you want to use more than one converter, queue, or item, you can specify arrays of the required size or use one variable for each converter.

2. Create the ADC data structure and specify the A/D converter.

   Use the `Ds1603Adc_create` function in your source code as required.

   The following table shows the default settings and the functions which can be used to change them:

| Parameter | Default Setting | Change Function |
|---|---|---|
| Converter clock | 12 MHz (8 bit accurancy) | `Ds1603Adc_setClock` |

3. Create the ADC queue data structure and specify the A/D converter queue.

   Use the `Ds1603AdcQueue_create` function in your source code as required.

   The following table shows the default settings and the functions which can be used to change them:

| Parameter | Default Setting | Change Function |
|---|---|---|
| Trigger mode | `DS1603_ADC_TRIGGER_SOFTWARE` | `Ds1603AdcQueue_setTriggerMode` |
| Interrupt mode | `DS1603_ADC_INTERRUPT_DISABLE` | `Ds1603AdcQueue_setInterruptMode` |

4. Create and specify the A/D channels to be handled by the queue.

   Use the `Ds1603AdcQueueItem_create` function in your source code as required.

   The following table shows the default settings and the functions which can be used to change them:

| Parameter | Default Setting | Change Function |
|---|---|---|
| Sample time | `DS1603_ADC_SAMPLE_TIME_2CLK` | `Ds1603AdcQueueItem_setSampleTime` |

5 Finalize the configuration of the ADC queue item.

   Use the `Ds1603AdcQueueItem_apply` function.

6 Finalize the configuration of the ADC queue.

   Use the `Ds1603AdcQueue_apply` function.

7 Finalize the configuration of the A/D converter.

   Use the `Ds1603Adc_apply` function.

8 Start the queue by typing the `Ds1603AdcQueue_trigger` function in a task routine.

9 Process the conversion result by implementing a combination of the `Ds1603AdcQueue_isReady`, the `Ds1603AdcQueue_getValue`, and the `Ds1603AdcQueueItem_readValue` functions in your code.

**Result** When you execute your application, the specified queue item is converted with the specified sample time and by the specified converter.

There are further functions for configuring and handling A/D conversion. For an overview of the functions available, refer to *A/D Conversion* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

The following source code excerpt shows how to implement A/D conversion for one queue with two queue items:

```
// Include brtenv.h:
#include <brtenv.h>

// Declare the ADC unit:
static DsS1603Adc *s_pAdc1;

// Declare the queue and two queue items:
static DsS1603AdcQueue* s_pAdc1Queue1;
static DsS1603AdcQueueItem* s_pAdc1Queue1Items[2];
init();
…
// Create the data structure for Adc 1:
Ds1603Adc_create(&s_pAdc1, DS1603_ADC_UNIT_1);
```

```
// Create the data structure for the first queue for ADC 1:
Ds1603AdcQueue_create(&s_pAdc1Queue1,
s_pAdc1, DS1603_ADC_QUEUE_NO_1);

// Create a conversion item for channel 35 for the first queue:
Ds1603AdcQueueItem_create(&s_pAdc1Queue1Items[0],
s_pAdc1Queue1, 35);

// Create the conversion item for channel 3 for the same queue:
Ds1603AdcQueueItem_create(&s_pAdc1Queue1Items[1],
s_pAdc1Queue1, 3);

// Set the sample time to a value different from default:
Ds1603AdcQueueItem_setSampleTime(s_pAdc1Queue1Items[0],
DS1603_ADC_SAMPLE_TIME_4CLK);

// Apply the settings to the hardware:
Ds1603AdcQueueItem_apply(s_pAdc1Queue1Items[0]);
Ds1603AdcQueueItem_apply(s_pAdc1Queue1Items[1]);
Ds1603AdcQueue_apply(s_pAdc1Queue1);
Ds1603Adc_apply(s_pAdc1);
…
// Start the conversion
Ds1603AdcQueue_trigger(s_pAdc1Queue1);
…
…
//
while( Ds1603AdcQueue_isReady(s_pAdc1Queue1)==DS_FALSE)
{
// Wait for ready
};
// Read value to internal buffer
Ds1603AdcQueue_read(s_pAdc1Queue1);
// Receive conversion values
val1=Ds1603AdcQueueItem_getValue(s_pAdc1Queue1Items[0]);
val2=Ds1603AdcQueueItem_getValue(s_pAdc1Queue1Items[1]);
…
```

# How to Implement Bit I/O on the I/O PLD

**Objective**

The RTLib1603 provides functions to read and write digital signals which are handled on the I/O PLD of the RapidPro Control Unit. The I/O PLD provides 20 channels, which can be individually configured as digital input or as digital output. To implement the bit I/O features in your application, you can use the following method.

**Basics**

- For an overview of the supported I/O features, refer to *Overview of Supported I/O Features* on page 152.
- For further information on the general concepts of implementing the I/O features, refer to *General Concepts of Implementing I/O Features* on page 167.

**Configuring bit I/O on the I/O PLD**

To implement bit I/O in the I/O PLD of the RapidPro system, you have to initialize the following data structures provided by the RTLib:

- `DsS1603ExtBitIo`

  This structure represents an external bit I/O unit, which is used either as digital input or as digital output.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Hardware preconditions:
  - To implement bit I/O, your RapidPro system must contain SC or PS modules which provide digital input (for example, SC-DI 8/1) and digital output (for example, SC-DO 8/1) signals. Additionally, these digital inputs and outputs must be routed to the I/O PLD of the RapidPro system. For a list of available modules, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.
- Software preconditions:
  - Your application must contain at least the definitions required for the main routine. For further information, refer to *Creating the Main Routine of Your Application* on page 75.

- For building your application, the following header files must be included in your source code:

| Header File | Used For ... |
|---|---|
| Ds1603ExtBitIO.h | Bit I/O on the I/O PLD |

The header files reside in %DSPACE_ROOT%\DS1603\RTLib.

If you include brtenv.h, these header files are included automatically in your application.

- Before you can use bit I/O in the out direction, you have to enable the outputs of your RapidPro system by using the ds1603enable_output function. For further information, refer to *Implementing I/O handling in your application* on page 170.

Steps 1 - 5 must be implemented in the main routine directly after the RapidPro system initialization.

**Method** **To implement bit I/O on the I/O PLD**

1. Declare the variables to be used for the data structures on the I/O PLD.

   If you want to use more than one channel, you can specify arrays of the required size or use one variable for each channel.
2. Create the bit I/O data structure and specify the channel.

   Use the Ds1603ExtBitIo_create function in your source code as required.
3. Specify the settings for bit I/O on the I/O PLD.

   Use the Ds1603ExtBitIo_setBit for the logical level of the channel and Ds1603ExtBitIo_setDirection for the direction of the bit I/O.
4. Finalize the configuration of the bit I/O channel on the I/O PLD.

   Use the Ds1603ExtBitIo_apply function.
5. Start the bit I/O by using the Ds1603ExtBitIo_start function.

6 Process the bit I/O result by implementing a combination of the `Ds1603ExtBitIo_read`, the `Ds1603ExtBitIo_getBit`, the `Ds1603ExtBitIo_setBit`, and the `Ds1603ExtBitIo_write` functions in your code.

**Result** When you execute your application, you have implemented bit I/O on the I/O PLD.

There are further functions for configuring and handling bit I/O. For an overview of the functions available, refer to *Bit I/O on the I/O PLD* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

The following source code excerpt shows how to implement bit I/O on the I/O PLD:

```
#include <brtenv.h>
int main()
{
// Declare data structures for bit I/O on the I/O PLD:
static DsS1603ExtBitIo* s_pBitIO1;
init();
…
// Create DIG_IO_CHANNEL_01:
Ds1603ExtBitIo_create(&s_pBitIO1, 1);

// Specify the settings for channel 1:
Ds1603ExtBitIo_setBit(s_pBitIO1, 0);
Ds1603ExtBitIo_setDirection(s_pBitIO1, DS1603_EXT_BITIO_OUTPUT);

// Apply the settings to the hardware:
Ds1603ExtBitIo_apply(s_pBitIO1);
…
// Start the bit I/O on the I/O PLD:
Ds1603ExtBitIo_start(s_pBitIO1);
…
…
// Process the Bit I/O result:
Ds1603ExtBitIo_setBit(IO1,1)
Ds1603ExtBitIo_write(IO1);
…
```

# How to Implement Digital Inputs on the eTPU

**Objective**

The RTLib1603 provides functions to read digital signals, which are handled on the two enhanced time processor units (eTPUs) of the MPC5554. To implement the digital input feature in your application, you can use the method described below.

**Basics**

- For an overview of the supported I/O features, refer to *Overview of Supported I/O Features* on page 152.
- For further information on the general concepts of implementing the I/O features, refer to *General Concepts of Implementing I/O Features* on page 167.

**Configuring digital inputs on the eTPU**

To implement digital inputs on the eTPU, you have to declare the following data structures provided by the RTLib:

- `DsS1603Tpu`

  The structure represents an enhanced time processor unit (eTPU).

- `DsS1603TpuDigIn`

  The structure represents an eTPU channel configured as digital input on the MPC5554 microcontroller (DS1603).

The MPC5554 provides two eTPUs with a total of 58 channels to be used as digital inputs.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Hardware preconditions:
  - To implement digital inputs, your RapidPro system must contain SC or PS modules which provide digital input signals (for example, SC-DI 8/1). Additionally, these digital input signals must be routed to three eTPU channels of the MPC5554. For a list of available modules, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.

■ Software preconditions:

- Your application must contain at least the definitions required for the main routine. For further information, refer to *Creating the Main Routine of Your Application* on page 75.
- For building your application, the following header files must be included in your source code:

| Header File | Used For ... |
|---|---|
| `Ds1603Tpu.h` | eTPU initialization |
| `Ds1603TpuDigIn.h` | Bit I/O on the eTPU |

The header files reside in `%DSPACE_ROOT%\DS1603\RTLib`.

If you include `brtenv.h`, these header files are included automatically in your application.

Steps 1 - 6 must be implemented in the main routine directly after the RapidPro system initialization.

**Method** **To implement digital inputs on the eTPU**

1. Declare the variables to be used for the data structures on the eTPU.
2. Specify the eTPU.

   Use the `Ds1603Tpu_create` function in your source code as required.
3. Specify the eTPU channel and configure it as digital input.

   Use the `Ds1603TpuDigIn_create` function in your source code.

   The following table shows the default settings and the functions which can be used to change them:

| Parameter | Default Setting | Change Function |
|---|---|---|
| Interrupt mode | `DS1603_TPU_DIGIN_INT_NONE` | `Ds1603TpuDigIn_setInterruptMode` |

4 Finalize the configuration of the digital input channel on the eTPU.

   Use the `Ds1603TpuDigIn_apply` function.

5 Finalize the configuration of the eTPU.

   Use the `Ds1603Tpu_apply` function.

6 Start the digital input by using the `Ds1603TpuDigIn_start` function.

7 Process the digital input result by implementing a combination of the `Ds1603TpuDigIn_read` and the `Ds1603TpuDigIn_getBit` functions in your code.

**Result** When you execute your application, you have implemented a digital input on the eTPU.

There are further functions for configuring and handling bit I/O on the eTPU, for example, for reading the logical level applied to the I/O pin of the selected channel. For an overview of the functions available, refer to *Bit I/O on the eTPU* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

The following source code excerpt shows how to implement digital inputs on the eTPU:

```
#include <brtenv.h>
int main()
{
// Declare an eTPU data structure for eTPU A:
static DsS1603Tpu *s_pTpu1;
UInt32 Bit1;

// Declare an eTPU DigIn data structure for eTPU A:
static DsS1603TpuDigIn *s_pTpuADigIn;
init();
…
// Create an eTPU device for eTPU A:
Ds1603Tpu_create(&s_pTpu1, DS1603_TPU_UNIT_1);

// Create a digital input on channel 1 of the eTPU A:
Ds1603TpuDigIn_create(&s_pTpuADigIn, s_pTpu1, 1);
```

```
// Apply the settings to the eTPU channel:
Ds1603TpuDigIn_apply(s_pTpuADigIn);

// Apply the settings to the eTPU device:
Ds1603Tpu_apply(s_pTpu1);
…
// Start the digital input on the eTPU:
Ds1603TpuDigIn_start(s_pTpuADigIn);
…
…
// Process the digital input result
Ds1603TpuDigIn_read(s_pTpuADigIn);
Bit1 = Ds1603TpuDigIn_getBit(s_pTpuADigIn);
…
```

# How to Generate 1-Phase PWM Signals

**Objective**

1-phase PWM signal generation is fundamental to many control applications. A PWM signal is characterized by its period, duty cycle, and polarity. To implement this feature in your application, you can use the method described below.

**Basics**

- For an overview of the supported I/O features, refer to *Overview of Supported I/O Features* on page 152.
- For further information on the general concepts of implementing the I/O features, refer to *General Concepts of Implementing I/O Features* on page 167.

If you generate a PWM signal at the upper limit of the specified frequency range, it is not guaranteed that the duty cycle can be changed in small steps, for example, steps of 1%. To ensure precise duty cycle updating, you can select another frequency range.

**Configuring PWM signal generation**

To implement 1-phase PWM signal generation on the RapidPro system, you have to initialize the following data structures provided by the RTLib:

- `DsS1603Tpu`

  The structure represents an enhanced time processor unit (eTPU).

- `DsS1603TpuPwmOut`

  To represent an eTPU channel configured for 1-phase PWM signal generation.

The MPC5554 provides two eTPUs, each with 32 channels which can be used for PWM signal generation.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Hardware preconditions:
  - To implement PWM signal generation, your RapidPro system must contain SC modules, or PS modules, or any other modules which provide digital output channels, for example, SC-DO 8/1. For an overview of available modules, *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.
- Software preconditions:
  - Your application must contain at least the definitions required for the main routine. For further information, refer to *Creating the Main Routine of Your Application* on page 75.
  - For building your application, the following header files must be included in your source code:

| Header File | Used For ... |
|---|---|
| `Ds1603Tpu.h` | eTPU initialization |
| `Ds1603TpuPwmOut.h` | PWM signal generation |

The header files reside in `%DSPACE_ROOT%\DS1603\RTLib`.

If you include `brtenv.h`, these header files are included automatically in your application.

  - Before you can generate PWM signals on the eTPU, you have to enable the outputs of your RapidPro system by using the `ds1603_output_enable` function. For further information, refer to *Implementing I/O handling in your application* on page 170.

Steps 1 - 6 must be implemented in the main routine directly after the RapidPro system initialization.

**Method** **To generate 1-Phase PWM signals**

1. Declare the variables to be used for the data structures of the eTPU and the eTPU channel.

   If you want to use more than one eTPU, or more than one channel, you can specify arrays of the required size.

2. Specify the eTPU.

   Use the `Ds1603Tpu_create` function in your source code as required.

3. Specify the prescaler of the time base.

   Use the `Ds1603Tpu_setTimeBasePrescaler` function in your source code as required.

4. Specify the eTPU channel.

   Use the `Ds1603TpuPwmOut_create` function in your source code as required.

   The following table shows the default settings and the functions which can be used to change them:

| Parameter | Default Setting | Change Function |
|---|---|---|
| Edge polarity | `DS1603TPU_HIGH_ACTIVE` | `Ds1603TpuPwmOut_setPolarity` |
| Time base | `DS1603_TPU_TIME_BASE_1` | `Ds1603TpuPwmOut_setTimeBase` |
| Frequency | 1000 Hz | `Ds1603TpuPwmOut_setFrequency` |
| Duty cycle[1)] | 0x08000 (50%) | `Ds1603TpuPwmOut_setDuty` |
| Interrupt mode | `DS_DISABLE` | `Ds1603TpuPwmOut_setInterruptMode` |

1) The duty cycle is scaled in the range 0x0000 (0%) … 0x10000 (100%).

2. Finalize the configuration of the eTPU channel.

   Use the `Ds1603TpuPwmOut_apply` function.

3. Finalize the configuration of the eTPU.

   Use the `Ds1603Tpu_apply` function.

4. Start the PWM signal generation by typing the `Ds1603TpuPwmOut_start` function.

5. Update the PWM values by implementing a combination of the `Ds1603TpuPwmOut_setFrequency`, `Ds1603TpuPwmOut_setDuty`, and the `Ds1603TpuPwmOut_write` functions in your code.

**Result** When you execute your application, you have implemented 1-phase PWM signal generation on the eTPU.

There are several functions for configuring and handling PWM signal generation on the eTPU, for example, for writing the specified frequency and duty cycle to the eTPU. For an overview of the functions available, refer to *1-Phase PWM Signal Generation* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

The following source code excerpt shows how to implement 1-phase PWM signal generation on the eTPU:

```
#include <brtenv.h>
int main()
{
// Declare an eTPU data structure:
static DsS1603Tpu *s_pTpu2;

// Declare an eTPU PwmOut data structure:
static Ds1603TpuPwmOut *s_pTpuB13PwmOut;
init();
…
// Create an eTPU device for using eTPU B:
Ds1603Tpu_create(&s_pTpu2, DS1603_TPU_UNIT_2);

// Configure the time base prescaler 1 of the eTPU B:
Ds1603_setTimeBasePrescaler(s_pTpu2,
   DS1603_TPU_TIME_BASE_1, 16);

// create an PWM output channel using channel 13 of eTPU B:
Ds1603TpuPwmOut_create(&s_pTpuBPwmOut13, s_pTpu2, 13);

// set the polarity of the PWM signal:
Ds1603TpuPwmOut_setPolarity(s_pTpuBPwmOut13,
   DS1603_HIGH_ACTIVE);

// apply the settings to the eTPU channel:
Ds1603TpuPwmOut_apply(s_pTpuBPwmOut13);
```

```
// apply the settings to the eTPU device:
Ds1603Tpu_apply(s_pTpu2);
…
// start the PWM signal generation on channel 13 of eTPU B:
Ds1603TpuPwmOut_start(s_pTpuBPwmOut13);
…
…
// Reduce frequency from default 1000 Hz to 500 Hz:
Ds1603TpuPwmOut_setFrequency(s_pTpuBPwmOut13,500);

// Reduce duty cycle to 25%:
Ds1603TpuPwmOut_setDuty(s_pTpuBPwmOut13,0x04000);

// Update the PWM values:
Ds1603TpuPwmOut_write(s_pTpuBPwmOut13);
…
```

# How to Measure Frequencies of PWM Signals

**Objective**

With the PWM measurement, you can measure the frequency of a connected PWM signal and calculate the corresponding duty cycle. The MPC5554 supports the measurement of duty cycles and frequencies of 58 PWM signals on both eTPUs. To implement this feature into your application, you can use the below described method.

**Basics**

- For an overview of the supported I/O features, refer to *Overview of Supported I/O Features* on page 152.
- For further information on the general concepts of implementing the I/O features, refer to *General Concepts of Implementing I/O Features* on page 167.

**Configuring PWM signal measurement**

To implement PWM signal measurement on the RapidPro system, you have to implement the following data structures provided by the RTLib:

- `DsS1603Tpu`

  The structure represents an enhanced time processor unit (eTPU).

- `DsS1603TpuPwmIn`

  The structure represents an eTPU channel configured for PWM signal measurement.

The MPC5554 provides two eTPUs with a total of 58 channels to be used for PWM signal measurement.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Hardware preconditions:
  - To implement PWM signal generation, your RapidPro system must contain SC modules which provide digital input channels, for example, SC-DI 8/1. Additionally these signals must be routed to one of the 58 TPU channels which can be used for PWM signal generation. For an overview of available modules, refer to *Hardware Overview* in the *RapidPro System – Installation and Configuration Reference*.
- Software preconditions:
  - Your application must contain at least the definitions required for the main routine. For further information, refer to *Creating the Main Routine of Your Application* on page 75.
  - For building your application, the following header files must be included in your source code:

| Header File | Used For ... |
|---|---|
| `Ds1603Tpu.h` | eTPU initialization |
| `Ds1603TpuPwmIn.h` | PWM signal measurement |

The header files reside in `%DSPACE_ROOT%\DS1603\RTLib`.

If you include `brtenv.h`, these header files are included automatically in your application.

Steps 1 - 6 must be implemented in the main routine directly after the RapidPro system initialization.

**Method**

**To measure frequencies of PWM signals**

**1** Declare the variables to be used for the data structures of the eTPU and the eTPU channel.

If you want to use more than one eTPU, or more than one channel, you can specify arrays of the required size.

2 Specify the eTPU.

Use the `Ds1603Tpu_create` function in your source code as required.

3 Specify the prescaler of the time base.

Use the `Ds1603Tpu_setTimeBasePrescaler` function in your source code as required.

4 Specify the eTPU channel which should used for PWM measurement.

Use the `Ds1603TpuPwmIn_create` function in your source code as required.

The following table shows the default settings and the functions which can be used to change them:

| Parameter | Default Setting | Change Function |
|---|---|---|
| Edge polarity | `DS1603TPU_HIGH_ACTIVE` | `Ds1603TpuPwmIn_setPolarity` |
| Time base | `DS1603_TPU_TIME_BASE_1` | `Ds1603TpuPwmIn_setTimeBase` |
| Interrupt mode | `DS_DISABLE` | `Ds1603TpuPwmIn_setInterruptMode` |

5 Finalize the configuration of the eTPU channel.

Use the `Ds1603TpuPwmIn_apply` function.

6 Finalize the configuration of the eTPU.

Use the `Ds1603Tpu_apply` function.

7 Start the PWM signal generation by typing the `Ds1603TpuPwmIn_start` function.

8 Process the generation results by implementing a combination of the `Ds1603TpuPwmIn_readgetDuty`, `Ds1603TpuPwmIn_getFrequency`, and the `Ds1603TpuPwmIn_getDuty` functions in your code.

**Result** When you execute your application, you have implemented PWM signal measurement on the eTPU.

There are further functions for configuring and handling PWM signal measurement on the eTPU, for example, for reading the measured PWM frequency values to an internal buffer. For an overview of the functions available, refer to *PWM Signal Measurement (PWM2D)* in the *RapidPro System – Prototyping ECU MPC5554 RTLib Reference*.

The following source code excerpt shows how to implement PWM signal measurement on the eTPU:

```
#include <brtenv.h>
int main()
{
// Declare an eTPU data structure:
static DsS1603Tpu *s_pTpu1;

// Declare an eTPU PwmIn data structure:
static Ds1603TpuPwmIn *s_pTpuAPwmIn3;
UInt32 frequency;
UInt32 duty;
init();
…
// create an eTPu device for eTPU A:
Ds1603Tpu_create(s_pTpu1, Ds1603_TPU_UNIT_1);

// Configure the time base prescaler 1 of the eTPU A:
Ds1603Tpu_setTimeBasePrescaler(s_pTpu,
   DS1603_TPU_TIME_BASE_1,16);

// create a PWM input channel on channel 3 on eTPU A:
Ds1603TpuPwmIn_create(&s_pTpuAPwmIn3, s_Tpu1, 3);

// set the polarity of the channel:
Ds1603TpuPwmIn_setPolarity(s_pTpuAPwmIn3,
   DS1603_TPU_HIGH_ACTIVE);

// apply the settings to the eTPU channel:
Ds1603TpuPwmIn_apply(s_pTpuAPwmIn3);

// apply the settings to the eTPU device:
Ds1603Tpu_apply(s_pTpu1);
…
// start the PWM signal measurement on channel 3 on eTPU A:
Ds1603TpuPwmIn_start(s_pTpuAPwmIn3);
```

```
…
…
// Process the generation results:
Ds1603TpuPwmIn_read(s_pTpuAPwmIn3);
frequency = Ds1603TpuPwmIn_getFrequency(s_pTpuAPwmIn3);
duty = Ds1603TpuPwmIn_getDuty(s_pTpuAPwmIn3);
…
```

# Implementing Bus Protocols

**Objective**

To set up communication between your RapidPro system and a Controller Area Network (CAN), you can use the functions for CAN channel, message, and task handling provided by the Real-Time Library for the MPC5554 microcontroller module (DS1603).

**Where to go from here**

Information in this section

| |
|---|
| ***Implementing CAN Communication*** **on page 194**<br>When implementing CAN communication, you have to know some background information. |
| ***How to Implement CAN Communication*** **on page 200**<br>To use CAN communication, you have to create and configure CAN channels and messages. |
| ***Example of CAN Communication in the Demo Application*** **on page 205**<br>You can find suggestions for implementing CAN communication in the demo application. |

# Implementing CAN Communication

**Objective**

When implementing communication between your RapidPro system and a Controller Area Network (CAN), you have to know some specifics.

**CAN controller and transceivers**

The MPC5554 microcontroller provides 3 FlexCAN controllers which support the CAN 2.0B specification. Each FlexCAN controller represents a CAN channel which can be accessed via a bus transceiver. The MC-MPC5554 1/1 (DS1603) provides three slots for installing bus transceiver modules (TRX).

The following table gives an overview of the available TRX modules:

| Name | Identification | Description |
|---|---|---|
| TRX-CAN-LS 1/1 | DS1615 | 1 channel bus transceiver module for fault-tolerant low-speed CAN (ISO 11992) |
| TRX-CAN-HS 1/1 | DS1619 | 1 channel bus transceiver module for high-speed CAN (ISO 11898) |

The type of TRX module(s) in your RapidPro system determines whether you use a low- or high-speed CAN bus. The slot where a TRX module is mounted determines the channel number. You have to use this channel number in your application. For information on accessing the hardware details, refer to *Getting Hardware Details for Implementation* on page 37.

**WARNING! Corrupt CAN communication!**

Due to design limitations, the maximum baud rate must be limited under specific conditions. Refer to *Limitations for CAN Communication* on page 226.

**Supported CAN features**

The RapidPro hardware and the RTLib1603 support the following CAN features:

- Initializing CAN channels
- Setting the baud rate
- Setting up an unlimited number of CAN receive (Rx) and transmit (Tx) messages. The RTLib1603 guarantees that messages with higher priority (lower ID) are transmitted first.
- Message counter
- CAN monitoring function with FIFO buffer
- Bitmasks for filtering the CAN receive (Rx) monitor
- Events on receiving or sending specific CAN messages
- Event on the transition of a CAN channel from bus on to bus off state

**Unsupported CAN features**

The FlexCAN controllers on the MPC5554 microcontroller provide some CAN features that are not supported by the RapidPro hardware or the RTLib1603:

- The RTLib1603 does not support CAN remote messages.
- The RTLib1603 does not support CAN remote request messages.
- The RTLib1603 supports only the standard transceiver mode. You cannot switch to listen-only, sleep, wake, or standby modes.
- The RTLib1603 does not support wake-up and error transceiver signals.

**CAN handling via RTLib1603/RTKernel**

Handling CAN communication via RTLib1603/RTKernel can be divided into 3 parts:

- CAN channel handling
- CAN message handling
- CAN task handling

**CAN channel handling**

CAN channel handling is used to initialize and handle CAN channels. It comprises:

- Initializing CAN channels
- Setting the baud rate
- Setting and reading the status of the CAN controller

The RTLib1603 provides functions for handling CAN channels. The `DsCanCh.h` header file contains all function declarations, global variables, and required data types. For detailed information on functions for CAN channel handling, refer to *CAN Channel Handling* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**CAN message handling**

CAN message handling is used to initialize and handle CAN messages. It comprises:

- Creating CAN transmit (Tx) and receive (Rx) messages
- Monitoring the CAN bus
- Receiving and transmitting CAN messages, including reading and writing the CAN data
- Changing parameters of CAN messages such as data length code and message identifier
- Enabling and disabling CAN messages

The RTLib1603 provides functions for handling CAN messages. The `DsCanMsg.h` header file contains all function declarations, global variables, and required data types. For detailed information on functions for CAN message handling, refer to *CAN Message Handling* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**CAN task handling**

CAN task handling supports functions for binding user tasks to:

- The transmission of a specific CAN message
- The reception of a specific CAN message
- The occurrence of a CAN controller event, for example, the bus off event

If you want to implement CAN communication in connection with task handling, you have to bind RTKernel tasks to CAN events.

The `rtkCAN.h` header file contains all function declarations, global variables, and required data types. For detailed information on functions for CAN task handling, refer to *CAN Task Handling* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**Configuration of CAN communication**

The RTLib1603 provides configuration functions for handling CAN channels and messages which you can use to create and initialize a CAN object (channel or message).

To implement CAN communication on the RapidPro system, you have to initialize the following data structures provided by the RTLib1603:

- DsSCanCh

  The structure represents a channel on the CAN bus and corresponds to a CAN controller. You can combine various channels of the `DsSCanCh` type to specify a CAN bus.

  When configuring a CAN channel, you can make the following settings:

| Parameters | Default |
|---|---|
| Channel number | - |
| Baud rate | 500 kBit/s |
| Bus state | DSCAN_CH_BUS_ON |

- DsSCanMsg

  The structure represents a CAN message.

  When configuring a CAN message, you can make the following settings:

| Parameters | Default |
|---|---|
| Message identifier | - |
| Message format | - |
| Data length code[1)] | - |
| Message data[1)] | - |
| 1) CAN Tx messages only | |

- DsSCanMsgItem

  At run time, the structure can be used to store and read data of messages that cannot be accessed directly.

**Creating and configuring**

In contrast to other I/O driver structures, which are created and configured recursively, CAN objects (channels or messages) have to be created and configured sequentially. You have to carry out the following steps for each CAN object:

1. You have to create a data structure using the appropriate `create` function. The allocated data structure is used to parameterize the CAN object.
2. If necessary, you can make settings using the appropriate `set` functions.
3. To finalize the initialization of the CAN object, you have to use the appropriate `apply` function.

Before you can specify messages, you have to create a CAN channel structure.

**CAN communication at run time**

The RTLib1603 provides run-time functions for handling CAN channels and messages and some configuration functions, which can also be used at run time. You can use these to change the properties of a channel or message at run time.

You can change the following channel settings at run time:

| Parameters | Default |
|---|---|
| Baud rate | 500 kBit/s |
| Bus state | DSCAN_CH_BUS_ON |

You can change the following message settings at run time:

| Parameters | Default |
|---|---|
| Data length code[1)] | - |
| Message data[1)] | - |
| 1) CAN Tx messages only | |

**Changing settings**

When changing the settings of a CAN object at run time, you have to carry out the following steps for each CAN object:

1. Change the settings of the CAN object using the appropriate `set` functions.
2. You have to finalize the change using the appropriate `apply` function. With CAN Tx messages, you can also use `DsCanMsg_transmit`, which comprises the `apply` function.

# How to Implement CAN Communication

**Objective**

To implement CAN communication, you have to add RTLib functions for CAN handling to your application. You can do so by creating the necessary data structures, making suitable settings, and using adequate functions.

**Functions used in the demo application**

The following instruction deals with the functions which are used for CAN channel and message handling in the demo application.

There are further functions for configuring and handling CAN communication, for example, for connecting callback functions to specified events. For an overview, refer to *CAN* in the *RapidPro System - Prototyping ECU MPC5554 RTLib Reference*.

**Preconditions**

You must fulfill the following preconditions before you can carry out the instructions:

- Hardware preconditions:
  - To implement CAN communication, your RapidPro system must contain CAN transceiver modules. For further information, refer to *Getting Hardware Details for Implementation* on page 37.
- Software preconditions:
  - Your application must contain at least the definitions required for the main routine. Refer to *Implementing CAN Communication* on page 194.
  - If you want to implement CAN communication in connection with task handling, CAN services have to be bound to interrupts. This is done by using functions from the `rtkCAN.h` header file.

- For building your application, the following header files must be included in your source code:

| Header File | Used For ... |
|---|---|
| `DsCanCh.h` | CAN channel handling |
| `DsCanMsg.h` | CAN message handling |

The header files are located in `%DSPACE_ROOT%\DS1603\RTLib`.

If you include `brtenv.h`, these header files are included automatically in your application.

You are recommended to implement steps 1 - 9, which show the configuration of the CAN communication, in the main routine directly after the initialization of the RapidPro system.

**Method** **To implement CAN communication**

**1** Declare the variables to be used for the CAN channels and messages.

**WARNING! Corrupt CAN communication!**

Due to design limitations, the maximum baud rate must be limited under specific conditions. Refer to *Limitations for CAN Communication* on page 226.

**2** Create a channel using `DsCanCh_create` and parameterize the channel number that corresponds to the number of the CAN controller. By default, the CAN controller uses a baud rate of 500 kBit/s. If you want to change the baud rate, you can specify it via the `DsCanCh_setBaudrate` function.

**3** Finalize the configuration of the CAN channel by using the `DsCanCh_apply` function.

The specified settings are then stored in the channel data structure. You have created and configured a CAN channel.

4. Create a CAN receive (Rx) message using `DsCanMsg_createRx`, specify the CAN channel, and parameterize the message identifier and format.
5. Finalize the configuration of the CAN Rx message by using the `DsCanMsg_apply` function.

   The specified settings are then stored in the message data structure. You have created and configured a CAN receive message.
6. Create a CAN transmit (Tx) message using `DsCanMsg_createTx`, specify the CAN channel, and parameterize the message identifier and the message format.
7. Set the data length code using `DsCanMsg_setDlc`.
8. Specify the message data of the CAN Tx message using `DsCanMsg_setData`.
9. Finalize the configuration of the CAN Tx message by using the `DsCanMsg_apply` function.

   The specified settings are then stored in the message data structure. You have created and configured a CAN transmit message.

   The configured CAN messages can now be used in your run-time code.
10. In the run-time code, specify the message data of the CAN Tx message using `DsCanMsg_setData`.
11. In the run-time code, transmit a CAN Tx message using `DsCanMsg_transmit`.
12. In the run-time code, copy the CAN data of a CAN Rx message to a structure of DsSCanMsgItem type using `DsCanMsg_readRxItem`.

**Result** When you execute your application, your RapidPro system is ready to send and receive the specified CAN messages on the specified CAN channels.

The following code pattern shows how CAN channels are created and configured.

**Variable definitions:**

```
DsSCanCh * pMyCanChannel1 = 0;
DsSCanCh * pMyCanChannel2 = 0;
DsSCanMsg * pMyRxMessage = 0;
DsSCanMsg * pMyTxMessage = 0;
UInt8 MyTxMessageData[8] = {1, 6, 0, 3, 0, 0, 0, 0};
DsSCanMsgItem MyRxMessageItem;
```

**Channel configuration:**

```
DsCanCh_create(&pMyCanChannel1,1);
DsCanCh_setBaudrate(pMyCanChannel1,125000);
DsCanCh_apply(pMyCanChannel1);

DsCanCh_create(&pMyCanChannel2,1);
DsCanCh_setBaudrate(pMyCanChannel2,125000);
DsCanCh_apply(pMyCanChannel2);
```

**Message configuration:**

```
DsCanMsg_createRx(&pMyRxMessage,
   pMyCanChannel1,
   0x12,
   DSCAN_MSG_FORMAT_STD);
DsCanMsg_apply(pMyRxMessage);

DsCanMsg_createTx(&pMyTxMessage,
   pMyCanChannel2,
   0x12,
   DSCAN_MSG_FORMAT_STD);
DsCanMsg_setDlc(pMyTxMessage, 8);
DsCanMsg_setData(pMyTxMessage, MyTxMessageData);
DsCanMsg_apply(pMyTxMessage);
```

**Run time:**

```
DsCanMsg_setData(pMyTxMessage, MyTxMessageData);
DsCanMsg_transmit(pMyTxMessage);

DsCanMsg_readRxItem(pMyRxMessage, &MyRxMessageItem);
```

**Further example**

Additionally, there is an example that shows how CAN communication is implemented in the demo application. Refer to *Example of CAN Communication in the Demo Application* on page 205.

# Example of CAN Communication in the Demo Application

**CAN implementation**

In the demo application, a large part of CAN communication is implemented in a separate module, `DsaRapidProIO.c`, which is linked to the main routine.

The following example shows excerpts of how CAN communication is implemented in the demo application.

```
// Variable definitions
static DsSCanCh * s_pCANCh1;
DsSCanMsg * g_pCanCh1Msg0x10 = 0;
DsSCanMsg * g_pCanCh2Msg0x10 = 0;
UInt8 g_TxMsgData[8] = {0, 0, 0, 0, 0, 0, 0, 0};

void DsaCANInit()
{
// Channel configuration
   DsCanCh_create(&s_pCANCh1, 1);
   DsCanCh_setBaudrate(s_pCANCh1, 250000);
   DsCanCh_apply(s_pCANCh1);

// Message configuration
   DsCanMsg_createRx(&g_pCanCh1Msg0x10,
      s_pCANCh1,
      0x10,
      DSCAN_MSG_FORMAT_STD);
   DsCanMsg_apply(g_pCanCh1Msg0x10);

   DsCanMsg_createTx(&g_pCanCh2Msg0x10,
      s_pCANCh2,
      0x10,
      DSCAN_MSG_FORMAT_STD);
   DsCanMsg_setDlc(g_pCanCh2Msg0x10, 8);
   DsCanMsg_setData(g_pCanCh2Msg0x10, g_TxMsgData);
   DsCanMsg_apply(g_pCanCh2Msg0x10);
}
```

```
// Run time
void DsaCanTriggerTxCh2()
{
   DsCanMsg_setData(g_pCanCh2Msg0x10, g_TxMsgData);
   DsCanMsg_transmit(g_pCanCh2Msg0x10);
}

Int8 DsaGetCanIn10msSine1()
{
   DsSCanMsgItem MsgItem;
   DsCanMsg_readRxItem(g_pCanCh1Msg0x10, &MsgItem);
   CanIn_10ms_Sine1 = (Int8) MsgItem.Data[2];
   return (CanIn_10ms_Sine1);
}
```

# Building and Executing Real-Time Applications

**Objective**

For executing your applications on the RapidPro hardware, you need to generate and download them to the RapidPro system.

**Where to go from here**

Information in this section

| |
|---|
| ***Basics on Building and Downloading Real-Time Applications*** **on page 208**<br>To work with your application, you must start a build and download process by calling `down1603.exe`. |
| ***Basics on Executing Real-Time Applications*** **on page 212**<br>After you have built and downloaded your application, you can execute it. |
| ***How to Build and Download Real-Time Applications Using Your Own Makefile*** **on page 213**<br>You can build your application with your own makefile and then download it to the RapidPro hardware. |

# Basics on Building and Downloading Real-Time Applications

**Objective**

Building and downloading real-time applications is a process which you must start manually. To simplify the process, dSPACE provides a utility with which you can compile, link, and download your applications to your hardware.

**Build and download process**

The build and download process for real-time applications is managed by the `down1603.exe` utility installed on `%DSPACE_ROOT%\Exe`. The build and download process starts when you call `down1603` in the Command Window of your host PC. You can specify either the source files of your application or your own makefile as arguments:

- Using source files as arguments

  The following file types are supported by `down1603`:

  `.c, .ss, .s, .asm`.

  The first source file determines the naming of the compiled application. If you specify more than one source file, they must be separated by blanks, for example:

  ```
  down1603 myapplication.c mysubroutine.c
  ```

- Using your own makefile with included source files as arguments

  A makefile contains instructions for building, compiling, and generating a specific application. Typically it holds, for example, macros or rules for building a file. A makefile is particularly useful if you work with many source files. Instead of entering all the source file names in the Command Window every time you want to build the application, you can specify them once in the makefile. New source files can easily be added to the makefile. You can name your makefile according to your needs. Its name determines the naming of the compiled application.

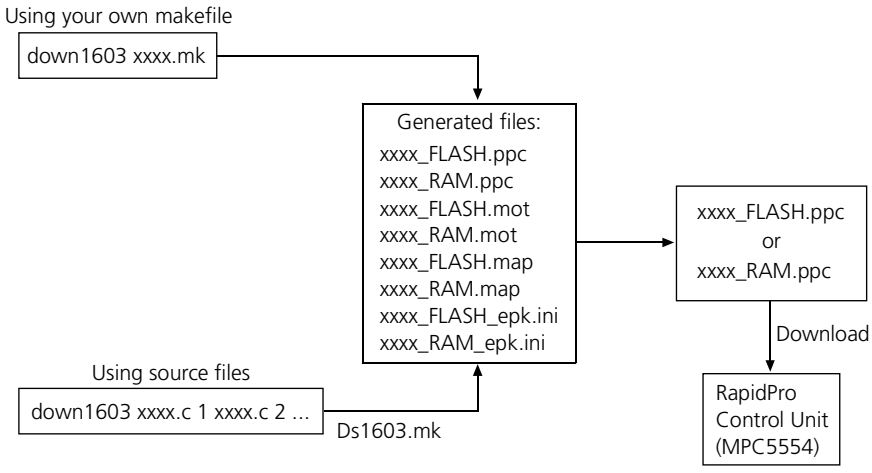
If you use individual source files as arguments, the build process is supported by the standard makefile `Ds1603.mk`.This is provided by dSPACE on `%DSPACE_ROOT%\DS1603\RTLib`. If you work with your own makefile, the standard makefile is not necessary for file generation. The files that are generated during the build process are always a RAM application and a flash application. This is because either a flash or a RAM application can run on the MPC5554 microcontroller. Options are used to define whether the flash or the RAM application is downloaded to the RapidPro system after file generation. By default, the flash application is downloaded.

**WARNING! Risk of injury and material damage!**

Uncontrolled movements and/ or material damage can occur since the application starts running immediately after download.

- Before downloading, think through the consequences calling `down1603` can have.
- Ensure that no one is in the potential danger zone of the device (test bench, etc.) when the application starts running.

The following illustration is an overview of the build and download process:




**Generated files**

The following file types are generated during the build process:

- Loadable program files for the MPC5554 microcontroller (`.ppc`)
- ECU Image files, which contain the code of the real-time application and, if available, the calibration parameters within the application (`.mot`)
- Files which contain the address ranges of the application and map symbolic names to physical addresses generated by the linker (`.map`)
- Files which contain an EPK identifier for consistency checks. The EPK identifier is an optional attribute in A2L files and ECU Image files. If it is defined, CalDesk automatically evaluates it to check if the A2L file and the ECU Image file are consistent with the real-time application on the RapidPro system. If they are inconsistent, a warning is displayed. (`epk.ini`).

**Options for downloading**

You can influence the build and download process by specifying options at the command line. The following table shows a selection of options:

| Option | Meaning |
|---|---|
| `/d` | Disables the download; only compiling and linking. |
| `/dsf` | Does not start the flash application after download. |
| `/ram` | Downloads the RAM application to the MPC5554 automatically after building. If this option is not specified, the flash application is loaded after the build process. |
| `/z` | Disables the building of the application and downloads only the existing object file. |
| `/?` | Displays information. |

For more options, refer to *Down1603.exe* in the *RapidPro System - Prototyping ECU Based on MPC5554 RTLib Reference*.

# Basics on Executing Real-Time Applications

**Objective**

Having built and downloaded your application, you can execute it.

**Running applications**

Depending on what options are used for the download process, either a flash or a RAM application is downloaded to the RapidPro hardware. Both applications start running on the MPC5554 microcontroller immediately after download. However, this does not apply to the flash application if the option `/dsf` is set, as this prevents the immediate starting of flash applications after download.

**Stopping applications**

Real-time applications can be stopped

- By switching off the RapidPro system.

  If the power of the RapidPro system is turned off, the contents of the RAM memory are lost. To work with the RAM application again, you must therefore download it again. A flash application, however, is not lost if you switch off power. It starts running again immediately after the RapidPro system is turned on.

- By switching to idle mode via ConfigurationDesk.

  A flash application can be restarted by switching back from idle mode to execution mode. For instructions, refer to *How to Switch from Execution Mode to Idle Mode in the ConfigurationDesk Configuration Guide*.

A RAM application, however, cannot be restarted. It must always be downloaded again before you can work with it again.

Be aware that you cannot stop or restart applications running on the RapidPro system via CalDesk.

# How to Build and Download Real-Time Applications Using Your Own Makefile

**Objective**

You can build a real-time application with your own makefile and then download the generated code to the RapidPro hardware.

**Described method**

You can, of course, also work with individual source files which you specify in the Command Window to build and download your real-time application. But since it is very convenient to specify source files in a makefile, this method is described here.

**Makefile template**

dSPACE provides a makefile which you can use as a template for your own makefiles. It is called `Templ1603.mk` and stored on `%DSPACE_ROOT%\DS1603\RTLib`.

**Default download setting**

If no additional option is set for the download process, the flash application is downloaded to the RapidPro hardware by default. For details on setting options, refer to *Basics on Building and Downloading Real-Time Applications* on page 208.

**Preconditions**

- C- or assembler-coded source files to be included in the makefile must exist.
- The TopologyID of your RapidPro system must be identical to the TopologyID specified in the initialization section of your application.
- ConfigurationDesk must be in offline mode or closed. For details, refer to *Accessing the RapidPro System* on page 218.
- The RapidPro device in CalDesk must be in a disconnected state, or CalDesk must be closed. For details, refer to *Accessing the RapidPro System* on page 218.

**Method**

**To build and download a real-time application using your own makefile**

1. Copy `Templ1603.mk` which is stored on `%DSPACE_ROOT%\DS1603\RTLib` to your working folder.
2. Rename the template file according to your requirements, for example `myMakefile.mk`.
3. In your makefile you must specify the names of your C- or assembler-coded source files for the `SRC_FILES` parameter. If there is more than one source file, you must separate them by blanks, for example:

   ```
   SRC_FILES      = myapplication.c mysubroutine.c
   ```

If required, you can also specify other parameters, for example, `OBJ_FILES`, where you can enter the names of already compiled components.

4. Open the Command Window from your working folder.

**WARNING! Risk of injury and material damage!**

Uncontrolled movements and/ or material damage can occur since the application starts running immediately after download.

- Before downloading, think through the consequences calling `down1603` can have.
- Ensure that no one is in the potential danger zone of the device (test bench, etc.) when the application starts running.

5. Type `down1603 myMakefile.mk`

**Result**

Your source files are compiled and linked and the generated files are stored in your working folder. The flash application is automatically downloaded to your RapidPro system and starts running immediately after download. It is named according to the specified makefile.

- To download the RAM application to your hardware and start it, use the `/ram` option. In the Command Window, type the following: `down1603 myMakefile.mk /ram`.

- If you have no hardware connected and you just want to compile and link your source files, you can set `/d` as an option to prevent downloading.
- For more options, refer to *Down1603.exe* in the *RapidPro System - Prototyping ECU Based on MPC5554 RTLib Reference*.

If the TopologyID in your source file is not identical to the TopologyID of your RapidPro system, your application starts briefly, then immediately stops running, and an error message is output during the TopologyID check. Make sure that the TopologyIDs always match.

# Using ConfigurationDesk and CalDesk Simultaneously

**Objective**

If you use ConfigurationDesk to configure or monitor your RapidPro hardware and simultaneously use CalDesk to control prototyping tasks, you should familiarize yourself with some specific characteristics of accessing the RapidPro system, and of handling projects, experiments, and applications.

**Where to go from here**

Information in this section

| |
|---|
| ***Accessing the RapidPro System*** **on page 218**<br>The RapidPro system cannot be accessed simultaneously with CalDesk and ConfigurationDesk. |
| ***Handling CalDesk and ConfigurationDesk Projects*** **on page 221**<br>Gives you information on the specifics of using the same projects in CalDesk and ConfigurationDesk. |

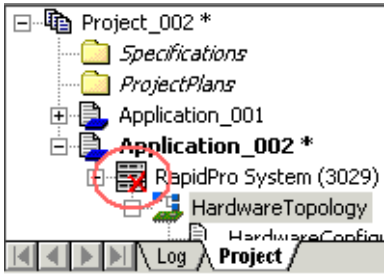
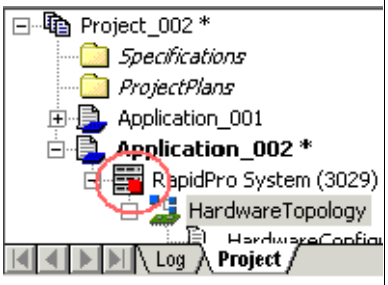
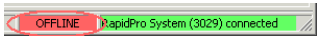
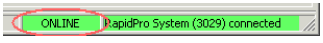
# Accessing the RapidPro System

**Objective**

If the RapidPro system is accessed by ConfigurationDesk, it is locked for CalDesk, and vice versa. The conditions that lead to locking the RapidPro system are different for ConfigurationDesk and CalDesk.

**Locking the access to the RapidPro system**

**ConfigurationDesk** The RapidPro system is locked when you switch from offline mode to online mode. The Project Manager and the status bar indicate whether the RapidPro system is locked or unlocked by ConfigurationDesk:

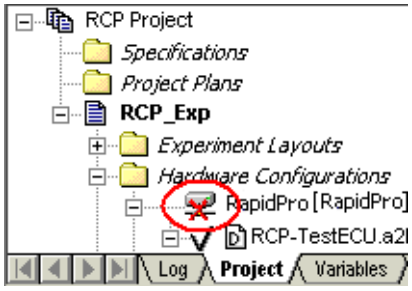
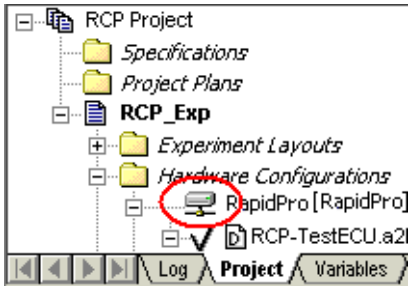
| | RapidPro System Unlocked | RapidPro System Locked |
|---|---|---|
| Project Manager |  |  |
| Icons |  <br>Offline mode |  <br>Online Mode (Submodes: Idle mode, Execution mode, Start-up mode, Error mode) |
| Status bar |  |  |

To unlock the RapidPro hardware, you must switch to offline mode.

For more information on configuring RapidPro hardware online and offline, refer to *Choosing the Mode* in the *ConfigurationDesk Configuration Guide*.

**CalDesk** The RapidPro hardware is locked when the status of the RapidPro device changes from "disconnected" to "connected".

In the Project Manager, you can identify whether the RapidPro hardware is locked or unlocked:

| | RapidPro System Unlocked | RapidPro System Locked |
|---|---|---|
| Project Manager |  |  |
| Icons |  <br>Device state: disconnected |  <br>Device states: connected, online calibration started, measuring |

If you open a project/experiment in CalDesk, CalDesk automatically connects to each device that is configured correctly and not locked by another tool. An unlocked RapidPro system is therefore often automatically locked by CalDesk and you have to unlock it manually.

To unlock the RapidPro device in CalDesk, you can disconnect it via its context menu in the Project Manager. If the RapidPro device is online (device state: "online calibration started" or "measuring"), stopping online calibration also disconnects it.

For more information, refer to *Basics of Device States* in the *CalDesk Calibration Guide*.

**Effects on other tools** If the RapidPro system is accessed by ConfigurationDesk or CalDesk, all write accesses to the RapidPro system are locked for other tools. For example, the following actions lead to an error message:

- Downloading handcoded real-time applications
- Updating the RapidPro firmware

The following table shows the tools that are involved in the locking process and how they are restricted.

| Tool | Restriction if the RapidPro System is Accessed by Another Tool |
|---|---|
| ConfigurationDesk | You cannot switch to online mode. |
| CalDesk | CalDesk cannot connect to the RapidPro device, you cannot start online calibration. |
| Down1603.exe[1)] | Write access is locked. |
| RapidProUpdate.exe[2)] | Write access is locked. |

1) An executable file for compiling, linking, and downloading handcoded real-time applications.

2) An executable file to check and update the RapidPro firmware.

**Diagnostic messages**

ConfigurationDesk's diagnostic messages are visible in CalDesk's log viewer. To reset diagnostic messages, you must disconnect the RapidPro device in CalDesk and operate ConfigurationDesk in online mode. For more information, refer to *Diagnostics Handling* in the *ConfigurationDesk Configuration Guide*.

# Handling CalDesk and ConfigurationDesk Projects

**Objective**

You can work with the same project in CalDesk and in ConfigurationDesk. This involves some specifics which you should note.

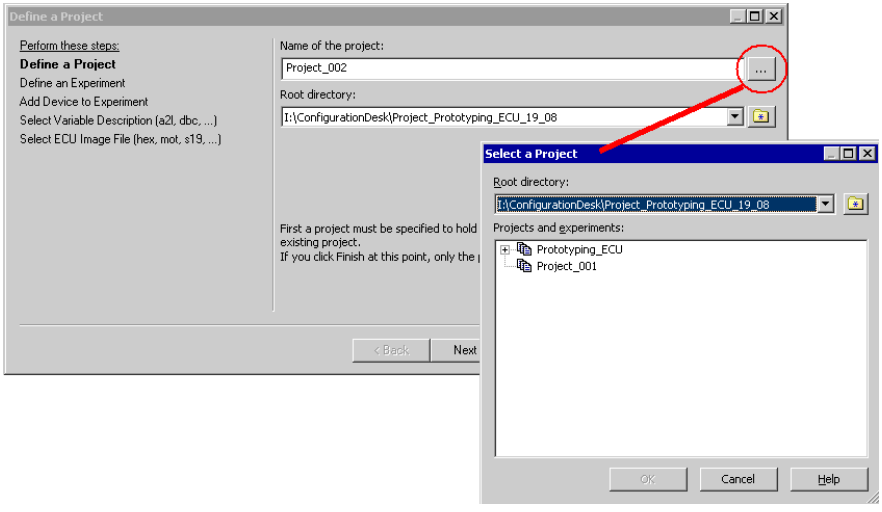
**Opening a project of the other tool**

You can open a CalDesk project in ConfigurationDesk and a ConfigurationDesk project in CalDesk.

In such cases the specific project content of the other tool is shown with grayed items (read-only):

- CalDesk experiments are read-only in ConfigurationDesk
- ConfigurationDesk applications are read-only in CalDesk

If you have created a project in ConfigurationDesk, you cannot open it in CalDesk via the "Open Project + Experiment" command. Instead, use "New Project + Experiment" and click the Browse button in the dialog to select the ConfigurationDesk project.



### Opening a project in both tools

You can also open the same project simultaneously in CalDesk and ConfigurationDesk. In this case you can edit existing CalDesk experiments with CalDesk and existing ConfigurationDesk applications with ConfigurationDesk (see the table below for exceptions).

If you want to add a new experiment or a new application to a simultaneously used project, you must open the project with the appropriate tool first:

- To add new experiments, open the project with CalDesk first.
- To add new applications, open the project with ConfigurationDesk first.

The following table gives an overview of the effects of different project loading sequences.

| Loading Sequence | Effects on Working with CalDesk | Effects on Working with ConfigurationDesk |
|---|---|---|
| The project is first opened in CalDesk | • The whole project can be edited, except for ConfigurationDesk applications.<br>• ConfigurationDesk applications are read-only, and changes made with ConfigurationDesk during the session are not displayed in CalDesk. | • Only existing ConfigurationDesk applications can be edited. However, some actions are locked, unless the project is opened in ConfigurationDesk first. For example, renaming or removing applications, hardware topologies, or hardware configurations is not possible.<br>• New applications cannot be added.<br>• CalDesk experiments are read-only, and changes made with CalDesk during the session are not displayed in ConfigurationDesk. |

| Loading Sequence | Effects on Working with CalDesk | Effects on Working with ConfigurationDesk |
|---|---|---|
| The project is first opened in ConfigurationDesk | • Only existing CalDesk experiments can be edited. However, some actions are locked, unless the project is opened in CalDesk first. For example, renaming or removing of experiments is not possible.<br>• New CalDesk experiments cannot be added.<br>• ConfigurationDesk applications are read-only, and changes made with ConfigurationDesk during the session are not displayed in CalDesk. | • The whole project can be edited, except for CalDesk experiments.<br>• CalDesk experiments are read-only, and changes made with CalDesk during the session are not displayed in ConfigurationDesk. |

**Global devices**

In CalDesk, you can make devices available to other experiments of a project. These devices are called global devices.

If you want to add, remove, or edit a global device to/in a simultaneously used project, you must open the project in CalDesk first. For more information, refer to *How to Add a Device to an Experiment* in the *CalDesk Calibration Guide*.

# Limitations and Troubleshooting

**Objective**

When using the RapidPro system as a prototyping ECU, you must be aware of some limitations and problems you might experience.

**Where to go from here**

Information in this section


| |
|---|
| ***Limitations for CAN Communication*** **on page 226** |
| ***Limitation for Using Floating-Point Constants*** **on page 227** |
| ***Troubleshooting*** **on page 228** |

# Limitations for CAN Communication

**Objective**

Due to design limitations, CAN communication with the RapidPro system using the MPC5554 Rev. A must be limited to guarantee faultless communication.

To avoid corrupt CAN communication, dSPACE has defined constraints and limitations relating to the maximum baud rate and to the maximum number of bytes used in the data field of the message.

**Constraints**

The following constraints must be met:

- A maximum of 100 RX messages per MPC5554 CAN channel are registered on the RapidPro system.
- Interrupts are disabled by the user in the user code for a maximum of 10 µs only.

**Limitations**

In compliance with the above constraints, the following limitations apply:

- Limitations using all **three** FlexCAN controllers of the MPC5554:
  - If you want to use baud rates equal to or higher than 850 kBit/s, the data field of a CAN standard message (CAN 2.0A) must have a min. length of 2 byte.

    It makes no difference whether the messages on the CAN bus are used by the RapidPro system or by other bus members.

  - Baud rates less than 850 kBit/s are not critical.
  - If you use only CAN extended messages (CAN 2.0B), CAN communication is not limited.
- Limitations using **one** or **two** FlexCAN controllers of the MPC5554: CAN communication is not limited.

# Limitation for Using Floating-Point Constants

**Low performance using double precision float values**

The Microtec PowerPC C compiler interprets floating-point constants by default as double precision float values. The MPC5554 microcontroller works efficiently only with single precision float values, because double precision float values are calculated in software emulation mode. You should therefore use the f suffix for constants to mark them for the compiler as float data types with single precision.

```
double period = 0.02f
```

# Troubleshooting

**Objective**

If a problem comes up, this chapter provides a collection of possible malfunctioning scenarios and how to solve the problem.

**Download of your application failed**

The download of your flash application is hung up with the following error message:

```
ERROR: downloading of <application_name> failed.
```

➤ Open a Command Window and type `Ldrpap /wf`.

This clears the application's flash memory.

If this information does not help you solve the problem, you should check the support section of our Web site http://www.dspace.com/goto?support. This might help you solve the problem on your own. The support's FAQ section especially might be of help.

If you cannot solve the problem, contact dSPACE Support via dSPACE Support Wizard. It is available:

- On your dSPACE CD/DVD at `\Diag\Tools\dSPACESupportWizard.exe`
- Via **Start – Programs – dSPACE Tools** (after installation of the dSPACE software)
- At http://www.dspace.com/goto?supportwizard

  You can always find the latest version of dSPACE Support Wizard here.

## A

A/D conversion
  applicable modules 158
  characteristics 156
  configuration 171
  conversion process 156
  end of conversion behavior 158
  implementation instructions 173
  trigger modes 158
A2L file generator 19
A2L files
  format specification 141
application structure 76
  calibration 77
  controller model 77
  demo application 77
  I/O access 77
  real-time frame 76
ASAM-MCD 2MC files
  format specification 141

## B

bit I/O
  applicable modules 160
  characteristics 160
bit I/O on eTPU
  configuring 179
  implementation instructions 180
bit I/O on I/O PLD
  configuration 176
  implementation instructions 177
building and executing real-time applications 207
  down1603 208
  makefile 208
  options for downloading 211

## C

CalDesk 19
CAN communication 194
  channel handling 195
  configuration 196
  controller and transceivers 194
  example 205
  implementing instructions 201
  limitations 226
  message handling 196
  run time functions 198
  supported CAN features 195
  task handling 196
  unsupported CAN features 195
CAN controllers 194
changing hardware components 29
COM-USB-PI 1/1 21
ConfigurationDesk 18
connection to the host PC 18
creating
  main routine 78

## D

demo application
  application structure 77
  binaries 42
  prepared binaries 44
  preparing calibration and measurement 64
  ready-to-use demo 44
  source files 42
down1603 208
download utility 18
DsaAdcInit 54
DsaAdcStartQueue 54
DsaCANInit 57
DsaCanTriggerTxCh 58
DsaDigIOInit 55
DsaGetADC 54
DsaGetBitIn 55
DsaGetCanIn 58
DsaGetPwmIn 57
DsaGetVecPwmIn 57
DsaRapidProIO.c 52, 54
DsaRapidProIO.h 52, 54
DsaSetBitOut 55
DsaSetCanOut 58
DsaSetPwmOut 56
DsaSetVecPwmOut 56
DsaTPUIOInit 55, 56
dSPACE Calibration and Bypassing Service 137
  implementing 142
dSPACE HelpDesk 8
duty cycle 163

## E

ECUCode.c 52
ECUCode.h 52
ECUCode_CalParams.c 52
ECUCode_CalParams.h 52
error messages 82
exception types 96
execution order
  tasks 108

## F

field of application 14
flushing message buffer 88

## G

general implementing concepts
  device driver structures 167
  starting I/O devices 168
getting hardware details
  I/O mapping information 39
  installed CAN transceivers 40
  TopologyID 38

## H

HelpDesk 8

## I

I/O mapping information 27
I/O PLD 154
implementation software
  introduction 17
  RTK1603 17
  RTLib1603 17
implementing
  CAN communication 201
  dSPACE Calibration and Bypassing Service 142
  exception handling 95
  memory pages 142
  messages 81
  paging mechanism 142
  time stamping 93
information messages 82
installed CAN transceivers 40
installing of CAN transceivers 28

## M

main routine
  creating 78
  real-time frame 76
makefile 208
MC-MPC5554 1/1 21
memory pages 138
  example 147
  implementing 142
  reference page 138
  working page 138
message handling 82
  basics 82
Microtec PowerPC C Compiler 18

MPC5554 14, 153
A/D converter 154
characteristics 154
FlexCAN controller 155
memory 154
processor clock rate 154
restrictions of available features 155
time processor unit 154

## O

options for downloading 211

## P

page switching
example 148
paging mechanism
callback function 140
implementing 142
memory descriptor 140
pointer-based 139
PDF files 9
pointer-based paging mechanism 139
example 149
polarity 163
printed documents 9
priority
tasks 106
PS modules 22
PWM period 163
PWM signal generation
applicable modules 164
characteristics 162
configuration 183
duty cycle 163
implementation instructions 185
polarity 163
PWM period 163
PWM signal measurement
applicable modules 166
characteristics 165
configuration 188
implementation instructions 189

## R

RapidPro Control Unit
features 154
RapidPro hardware
hardware details 37
RapidPro units 21
real-time frame (main routine) 76
reference page 138
refining exception handling 97
routing code 26
RoutingID 26
RTFrameDS1603.c 53
RTFrameDS1603.h 53
RTK1603
introduction 17
RTLib1603
introduction 17

## S

SC modules 22
signal routing 26
signal scaling 170
software versions required 19
stack 22

## T

tasks
aperiodic tasks 104
background tasks 105
execution order 108
idle state 108
implementing aperiodic tasks 121
implementing inherited tasks 126
implementing multiple periodic tasks 115
implementing one periodic task 110
implementing overrun handling 133
inherited tasks 104
interrupt service 105
overrun function 132
overrun handling 131
periodic tasks 103
priority 106
ready state 108
running state 108
states 108
task data structure 106
TCB 106
trigger 105
time stamping 92
implementing 93
timer 92
TopologyID 27
typical workflow for MPC5554 application 32

## U

unit connection bus 22

## V

variable descriptions
A2L files 141

## W

warning messages 82
working pages 138