Rulbus Device Library for Microsoft Windows Reference Manual

0.2.1 (stable)

Generated by Doxygen 1.4.0

Wed Apr 6 08:59:18 2005

Contents

Chapter 1

Rulbus Device Library for Microsoft Windows Main Page

The Rulbus Device Library is a Microsoft Windows 95/98/NT/2000/XP dynamic-link library (rulbus.dll) with functions to use the most popular Rulbus modules. It supports the ISA Rulbus Interface as well as the EPP Rulbus Interface.

This manual contains the following sections:

- User Manual
- Reference Manual
- Developer Manual

The User Manual tells you more about Rulbus and Rulbus Interfaces and it describes how to use this library.

The Reference Manual describes the functions available for the various Rulbus modules supported. To discover how to use a certain Rulbus module, look up its documentation page by Rulbus number in the table of Contents, Chapter 2, for example *RB8510 12-bit DAC* for the dual 12-bit DAC and read the documentation for the functions available.

To learn about the implementation of the Rulbus Device Library, explore the Developer Manual. It describes the implementation of the Rulbus modules and of the Rulbus interfaces.

RDL License

RDL is written by Martin J. Moene <m.j.moene@eld.physics.LeidenUniv.nl>

Web page: http://www.eld.leidenuniv.nl/~moene/software/rdl/

Chapter 2

Rulbus Device Library for Microsoft Windows Module Documentation

2.1 User Manual

2.1.1 Detailed Description

The Rulbus Device Library is a Microsoft Windows 95/98/NT/2000/XP dynamic-link library (rulbus.dll) with functions to use the most popular Rulbus modules. It supports the ISA Rulbus Interface as well as the EPP Rulbus Interface.

Modules

- Rulbus – RijksUniversiteit Leiden BUS

  computer-indepedent I/O bus – modules – racks – computer–Rulbus interfaces.

- Defining Interface and Modules

  computer–Rulbus interface – module configuration file – setrulbus utility.

- Creating a Program

  structure – compilation – error handling – example.

- Threads and the Rulbus Device Library

  multithreading – example.

- Compilers and the Rulbus Device Library

  Borland C++ – cdecl calling convention – compiling programs.

2.2 Rulbus – RijksUniversiteit Leiden BUS

I/O bus for peripherals The Rulbus is a simple input-output bus for peripherals, like an analog to digital converter. It is designed such that peripheral cards have a simple interface to the bus. Also, it is designed such that many microprocessor types can be connected to the Rulbus via a simple *Rulbus Interface*. Thus a single version of a peripheral card can be used with various types of microprocessors and microcomputer systems. For more information, see [rulbus].

Rulbus Modules or Cards There are circa 100 different Rulbus modules. Some of the most popular are:

- parallel interface module for digital input-output, RB8506 with PIA or VIA
- 12-bit analog to digital converter module, RB8509
- 12-bit digital to analog converter module, RB8510
- delay module, RB8514
- clock module, RB8515

Several properties of a Rulbus module are:

- the printed-circuit board has Euro-card format
- the card has a 50-pin Rulbus connector on its rear side that connects to the Rulbus flatcable
- most cards have input-output connectors on the front side
- the card is enclosed in a metal case

Rulbus Racks Based on an application's measurement and control requirements, various Rulbus modules are collected in one or more 14-inch racks. A rack may contain up to 12 single-width (35 mm) modules.

Figure 2.1: A Rulbus rack (half-width)

Several racks may be chained and then be connected to a computer via a computer–Rulbus interface.

The bus-part of the Rulbus The Rulbus as a *bus* is a collection of several lines that make communication between a computer and the various Rulbus cards possible.

The bus consists of the following lines:

- 8 lines to carry data (one data byte at a time)
- 8 lines to address a data byte
- 5 lines for control (enable, read-write, reset, interrupt and a 4 MHz clock signal)
- many lines for $+5$ Volt, $+15$ Volt and -15 Volt power supply

The Rulbus address space is thus 256 bytes (2^8). Two of these addresses are reserved:

- 0x00 or 0 is reserved to extend the range of available addresses (see further)
- 0xFF or 255 is reserved for the 'bus inactive' state.

There are Rulbus modules that only use one byte of the Rulbus address space, but there are also modules that occupy 32 bytes of the address space of 254 bytes. The address range a module uses, is determined when the module is made: it is programmed in the card's address GAL or PAL (programmable array logic).

If many modules are used together it may be difficult to assign all modules a proper address range and prevent address conflicts.

In section Defining the Modules on page **??** we will see how the Rulbus addressing scheme is extended by giving each rack its own address (secondary address).

In section Defining the Modules we will also see how the address and several other properties of each module to use can be specified in a *Rulbus configuration file*.

Rulbus Interfaces Nowadays, only two Rulbus Interfaces that connect to a PC are of interest:

- the ISA Rulbus Interface
- the EPP Rulbus Interface

The ISA Rulbus Interface is designed for the *Industry Standard Architecture* (ISA) slots of a PC. New PCs may not include ISA slots anymore, so a new Rulbus Interface was developed: the EPP Rulbus Interface.

The EPP Rulbus Interface is connnected to the parallel port of a computer (PC). For the interface to work, the parallel port must be configured to work in *Enhanced Printer Port* mode in the PC's BIOS, hence the EPP [IEEE1284].

The Rulbus Device Library supports both Rulbus Interfaces. At default the library assumes that the EPP Rulbus Interface is being used with the parallel port at address 0x378. If this assumption is not valid, the library must be informed what the proper interface type and address are. To this end the RULBUS environment variable can be defined. See section Defining the Interface.

2.3 Defining Interface and Modules

Defining the Interface The Rulbus Device Library supports the ISA Rulbus Interface as well as the EPP Rulbus Interface. At default the library assumes that the EPP Rulbus Interface is being used with the parallel port at address 0x378. If this assumption is not valid, the library must be informed what the proper interface type and address are. To this end the `RULBUS` environment variable can be defined as follows, here shown as shell commands:

- `set RULBUS=isa,0x200`
- `set RULBUS=epp,0x278`

You can add `;nocheck` to disable checking if the Rulbus Interface is present (EPP only). See also Setrulbus Utility.

Remember to configure the PC's parallel port to work in Enhanced Printer Port mode in the PC's BIOS, if you use the EPP Rulbus Interface.

Defining the Modules In a program, software objects represent the physical Rulbus modules. These software objects are created from a Rulbus configuration file. This happens when the program references `rulbus.dll` for the first time and the DLL is loaded by the operating system.

Here is a small Rulbus device configuration file.

```
# rulbus-small.conf - example rulbus device configuration file.

rack "top"
{
    address = 0

    rb8509_adc12 "adc"
    rb8510_dac12 "dac-ch0" { address = 0xD0 }
    rb8510_dac12 "dac-ch1" { address = 0xD2; bipolar = false; volt_per_bit = 1.25m }
}

rack "bottom"
{
   address = 1

   // still empty
}
```

See section **??** on page **??** for a longer configuration file.

The configuration file defines racks that contain modules (cards). Besides card declarations, a rack declaration usually contains an address specification (See secondary address).

Card Declarations The card declarations specify the type of the card and the name by which it is referenced from the program. All card types can specify the address property and several card types can specify other properties, like bipolar for card type rb8510_dac12. The available properties and their default values are given in the section *Default configuration* in the reference manual for the various card types.

Secondary Address When more than one rack is used, each rack must have a unique address. The address of a rack is defined at its rear side via a 4-bit DIP-switch, so that rack addresses fall in the range 0–15.

Address 15 has a special meaning: the rack is always selected or active. In this case it must be the only rack connected to the computer–Rulbus interface.

If a rack has an address in the range 0–14, the rack must be specifically selected by a program before the cards it contains can be accessed. Note that the Rulbus Device Library takes care of this.

So with rack addressing, the range of available addresses is extended to 15 times 254, or 3810.

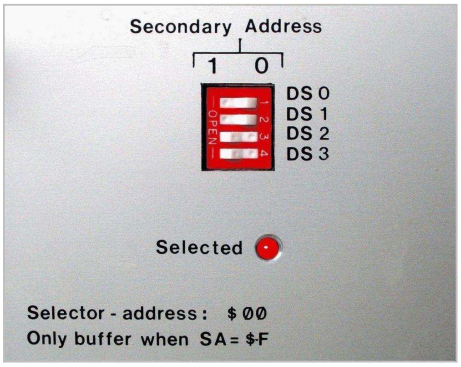


Figure 2.2: Secondary address selector

The figure above shows a rack with secondary address 3: binary 0011, $0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$.

Note that the Rulbus Interface cannot be specified in the Rulbus device configuration file. The rationale behind this is, that the Rulbus Interface is related to the computer, whereas the items in the Rulbus device configuration file are related to the application. When a PC is replaced by another one, the application and the Rulbus device configuration file should move to the new computer, but not the Rulbus Interface specification.

Which Configuration File The path of the configuration file to read is determined as follows:

1. contents of environment variable `RULBUS_CONFIG_FILE`, or
2. file .\rulbus.conf (current directory), or
3. file C:\etc\rulbus.conf

You can set the environment variable `RULBUS_CONFIG_FILE` in a command shell temporarily as follows:

- `set RULBUS_CONFIG_FILE=M:\myexperiment\etc\rulbus.conf`

Setrulbus Utility For Windows 95 types of operating system (Windows 95/98/me) the `RULBUS` and `RULBUS_CONFIG_FILE` environment variables must be recorded in `autoexec.bat` by editing it. For Windows NT types of operating system (Windows NT/2000/XP) they must be recorded in the registry, for example via System Properties, Environment.

To make it easy to select the proper Rulbus Interface and to define the path to the Rulbus configuration file on either operating system type, a small GUI program has been made: `setrulbus.exe`.

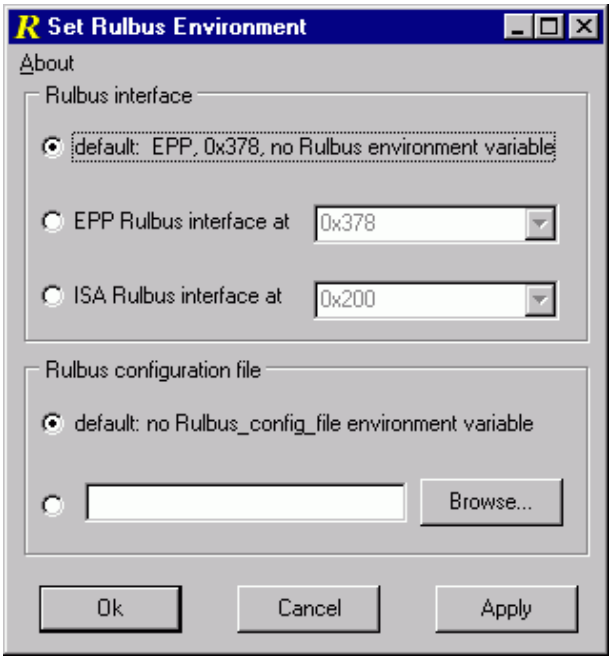


Figure 2.3: Set Rulbus Environment (setrulbus.exe)

Open setrulbus via `Start | Rulbus Device Library | Tools | Set Rulbus Environment`. Program setrulbus is located in the programs subdirectory of the Rulbus Device Library installation directory.

2.4 Creating a Program

Writing a Program The Rulbus Device Library consists of several functions for each Rulbus module. These functions are published in the C/C++ header file `rulbus.h`. You normally need to only call functions for the Rulbus modules you want to use (but also see Handling Errors).

Rulbus modules are used according to the pattern: open – use – close, for example:

```
#include "rulbus.h"

int main()
{
   int32 handle;

   rb8510_dac12_open      ( &handle, "dac-ch0" );    // open device
   rb8510_dac12_setVoltage(  handle,  ...      );    // use it
   ...                                               //    even more
   rb8510_dac12_close     (  handle            );    // close device

   return 0;
}
```

The name used to open the DAC, `"dac-ch0"`, corresponds to a name in the Rulbus device configuration file. See section Defining the Modules on page **??** . Note that the types for parameters that may be modified (pointers) include a size specification in function calls to the library (e.g. `int32 handle`). Other parameters may be specified using the normal types.

Besides the functions related to specific Rulbus modules, there are library-related functions and general Rulbus device functions. Function names appear in the following forms:

- `rdl_`... – library-related functions, like rdl_getLastError()
- `rb`*`yydd`*`_`... – specific Rulbus module related functions, like rb8510_dac12_open()
- `RulbusDevice_`... – general Rulbus device functions, like RulbusDevice_print()

Compiling the Program In the following command, {*include-path*} specifies the directory with `rulbus.h` and `rulbus-types.h` and {*library-path*} specifies the directory with `rulbus-omf.lib`.

To create the program for use with the Rulbus Device Library DLL, compile as follows:

```
C:\>bcc32 -I(include-path) -L{library-path} example.cpp rulbus-omf.lib
```

With Borland C++ 5.6 you can also create a standalone program. See Compiling Programs, section Borland C++ for more information.

Handling Errors All interface functions in the Rulbus Device Library use the same error handling pattern: they return the following status values:

- `0` the function call was successful
- `1` an error occurred

Notable exceptions to this rule are functions RulbusDevice_getByte(), RulbusDevice_putByte() and DllMain().

Properly written programs should always check the function return value to see if the function call was successful and should act accordingly. When an error occurred, a string explaining the error condition may be obtained with function rdl_getLastError().

```
#include <stdio.h>      // for stderr, fprintf()
#include "rulbus.h"     // for rb8510_dac12_open() etc.

static int error();

int main()
{
   int32 handle;

   if ( rb8510_dac12_open( &handle, ... ) )
      return error();

   // ...

   return 0;
}

static int error( )
{
   const int len = 100; char msg[len];

   rdl_getLastError( msg, len );
   fprintf( stderr, "%s\n", msg );

   return 1;
}
```

Complete Example The following example program opens a DAC, generates a staircase voltage and closes the DAC again.

```
/*
 * dac.cpp - generate a staircase voltage.
 *
 * compile: bcc32 dac.cpp rulbus.lib
 */

#include <stdio.h>      // for printf() etc.
#include <stdlib.h>     // for strtol()
#include <windows.h>    // for Sleep()
#include "rulbus.h"     // rulbus interface

static int usage();     // print program usage, return EXIT_FAILURE
static int error();     // print error, return EXIT_FAILURE

/*
 * main - handle commandline arguments and generate staircase voltage on DAC.
 */

int main( int argc, char *argv[] )
{
   /*
    * handle commandline arguments:
    */

   if ( argc < 2 )
      return usage();

   /*
    * the name on the commandline must correspond to the name of a 12-bit
    * DAC in the Rulbus device configuration file, typically rulbus.conf.
```

```
    */

   char *name = argv[1];

   /*
    * open the DAC:
    */

   int32 handle;
   if ( rb8510_dac12_open( &handle, name ) )
      return error();

   /*
    * generate 11 one volt steps, one per second:
    *
    * Note that the last step generates a RulbusRangeError, because the
    * voltage is outside [-10.235 .. +10.24 V].
    */

   for ( int i = 0; i <= 11; i++ )
   {
      fprintf( stdout, "[%d]", i );

      if ( rb8510_dac12_setVoltage( handle, i ) )
         return error();

      Sleep( 1000 );                    // delay one second
   }

   /*
    * close the DAC:
    */

   if ( rb8510_dac12_close( handle ) )
      return error();

   return EXIT_SUCCESS;
}

/*
 * usage - print program usage.
 */

static int usage()
{
   fprintf( stdout, "Usage: dac device-name\n" );
   return EXIT_FAILURE;
}

/*
 * error - retrieve and print Rulbus error.
 */

static int error()
{
   const int len = 100; char msg[len];

   rdl_getLastError( msg, len );
   fprintf( stdout, "%s\n", msg );

   return EXIT_FAILURE;
}
```

2.5 Threads and the Rulbus Device Library

Multithreading Because the Rulbus Device Library is used on a multitasking–multithreading operating system, attention must be given to the effects this may have. For example, it should not be possible that in one thread a Rulbus device just selected its rack, followed by a second device in another thread setting one of its registers, assuming that *its* rack still is selected.

The Rulbus Device Library enforces the following:

- access to all Rulbus devices is serialized so that only one thread can alter a device at a time and only one device can access the Rulbus at a time
- selecting a rack and reading a byte from or writing a byte to the Rulbus is also serialized

Serialized means that an operation is completed before the next operation will be allowed to commence. To this end semaphores are used.

Because of the way DLLs use memory, the following applies:

- Rulbus devices can be shared among different threads of the same process (see client-server thread example)
- Rulbus devices cannot be shared among different processes, although different Rulbus devices can be used from different processes (see client-server process example)

Example The following program example shows the use of three threads, two of which control their own DAC. One thread generates a square wave voltage, the other thread generates a sawtooth voltage. Error handling has been omitted for clarity.

```
/*
 * threads.cpp - use Rulbus Device Library from three threads.
 *
 * compile: bcc32 threads.cpp rulbus.lib
 */

#include <conio.h>                      // for kbhit()
#include <stdio.h>                      // for fprintf(), sscanf()
#include <string.h>                     // for stricmp()
#include <windows.h>                    // for DWORD HANDLE, Sleep()

#include "rulbus.h"                     // for rb8510_dac12_open() etc.

const char *title = "Threads  1.0  use Rulbus DLL from three threads.\n";

enum { E_OK, E_OPT, E_ARG, };           // program return codes

static int           threads   ( );
static HANDLE        mkThread  ( const char *msg, DWORD WINAPI (*ThreadFunc)(LPVOID), int32 *pHandle );
static DWORD  WINAPI SquareWave( LPVOID arg );
static DWORD  WINAPI SawTooth  ( LPVOID arg );

/*
 * main - the program.
 */

int main( int argc, char *argv[] )
{
   fprintf( stdout, "%s\n", title );

   return threads();
}
```

```
/*
 * threads - create a separate thread for each of two DAC channels
 *           and generate a square wave and a sawtooth.
 *
 *      100% |--    --    --
 *           |  |  |  |  |  |
 *       50% |   --    --
 *           | /| /| /| /| /
 *        0% |/_|/_|/_|/_|/___
 */

static volatile int runthread = 1;

static int threads()
{
   rdl_printRulbusInterface();

   fprintf( stdout, "\nGenerating waveforms on DAC channels 0 & 1.\n" );

   /*
    * open two DACs:
    */

   int32 dac0, dac1;

   rb8510_dac12_open( &dac0, "dac-ch0" );
   rb8510_dac12_open( &dac1, "dac-ch1" );

   /*
    * create and resume threads:
    */

   HANDLE thread0 = mkThread( "DAC channel-0", SquareWave, &dac0 );
   HANDLE thread1 = mkThread( "DAC channel-1", SawTooth  , &dac1 );

   fprintf( stderr, "\n\nPress a key to stop...\n\n" );

   ResumeThread( thread0 );
   ResumeThread( thread1 );

   /*
    * wait for key pressed; eat character:
    */

   while (!kbhit() )
     Sleep( 10 );

   (void) getch();

   /*
    *  stop and remove threads and close DACs:
    */

   runthread = 0; Sleep( 10 );

   CloseHandle( thread0 );
   CloseHandle( thread1 );

   rb8510_dac12_close( dac1 );
   rb8510_dac12_close( dac0 );

   return E_OK;
}

/*
 * mkThread - create a thread.
```

```
 */

static HANDLE mkThread( const char *msg, DWORD WINAPI (*ThreadFunc)(LPVOID), int32 *pHandle )
{
   fprintf( stdout, "\ncreating thread for %s", msg );

   DWORD id;

   return CreateThread(
      NULL,             // pointer to thread security attributes
      0,                // initial thread stack size, in bytes
      ThreadFunc,       // pointer to thread function
      pHandle,          // argument for new thread
      CREATE_SUSPENDED, // creation flags
      &id               // pointer to returned thread identifier
   );
}

/*
 * SquareWave - generate a square wave voltage.
 */

static DWORD WINAPI SquareWave( LPVOID arg )
{
   int32 handle = *(int32 *) arg;

   fprintf( stdout, "thread function: squarewave on " ); RulbusDevice_print(handle);

   while ( runthread )
   {
      rb8510_dac12_setValue( handle, 2048 ); Sleep(50);
      rb8510_dac12_setValue( handle, 4095 ); Sleep(50);
   }

   return 0;
}

/*
 * SawTooth - generate a sawtooth voltage.
 */

static DWORD WINAPI SawTooth( LPVOID arg )
{
   int32 handle = *(int32 *) arg;

   fprintf( stdout, "thread function: sawtooth on " ); RulbusDevice_print(handle);

   while ( runthread )
   {
      for( int n = 0; n <= 2047; n = (n + 50) % 2048 )
      {
         rb8510_dac12_setValue( handle, n ); Sleep(1);
      }
   }

   return 0;
}

/*
 * End of file
 */
```

2.6 Compilers and the Rulbus Device Library

This section describes some implementation details, that nonetheless may be important if you want to use the library with another compiler than Borland C++.

Compiler The Rulbus Device Library has been developed with Borland C++ 5.6.

Some reasons for this are:

- it is a 32-bit Windows native compiler that supports the creation of DLLs well
- it supports standard C++ exception handling

There are at least two important aspects to this choice:

1. the library uses C++ exception handling internally

   GNU C++ does not support the C++ exception handling as specified in the C++ standard, so that it cannot be used to compile the library.

2. the `rulbus.lib` import library is in OMF format

   Borland C++ produces (import) libraries in OMF format, whereas GNU C and Visual C++ use (import) libraries in COFF format. There is no simple tool to convert an OMF import library to COFF, but an import library in COFF format, `rulbus-coff.lib`, is provided to use the library with a compiler other than Borland C++.

Calling convention The Rulbus Device Library uses the `cdecl` calling convention. It is specified via the `RDL_API` preprocessor symbol in the function prototypes (rulbus.h) and definitions.

The calling convention is the arrangement of arguments for a procedure or function call. Different programming languages may require arguments to be pushed onto a stack or entered in registers in left-to-right or right-to-left order, and either the caller or the callee can be responsible for removing the arguments. The calling convention also determines if a variable number of arguments is allowed [foldoc].

Implicit linking a DLL requires an import library that is compatible with the compiler that is used to create a program. It is especially difficult to create an import library for Microsoft Visual C++ when the `stdcall` calling convention is used for the functions that are exported by the DLL created with Borland C++. Harold Howe's *Creating DLLs in BCB that can be used from Visual C++* [BCBDEVDLL] shows how it can be done for both the `cdecl` and the `stdcall` calling convention.

Compiling programs

The following sections show how programs can be compiled using Borland C++, Microsoft Visual C++, GNU C and LabWindows/CVI.

Here, {*include-path*} specifies the directory with `rulbus.h` and `rulbus-types.h` and {*library-path*} specifies the directory with `rulbus-omf.lib` or with `rulbus-coff.lib`.

Borland C++ This compiler uses OMF library format.

```
bcc32 -I{include-path} -L{library-path} {program}.cpp rulbus-omf.lib
```

With Borland C++ version 5.6 you also can create a program that can be used without the DLL (standalone program). To this end the program is statically linked with the Rulbus Device Library, the Rulbus Device Class Library and the Windows Utilities Library.

```
bcc32 -tWM -I(include-path) -L{library-path} {program}.cpp \
   rulbus-omf-static.lib rulbusdcl-omf-static.lib winutils-omf-static.lib
```

Option `-tWM` specifies that the program must be linked with the multi-threaded C runtime library.

When you create a standalone program, you must inititialize the Rulbus Device Library yourself with rdl_initialize().

Visual C++ This compiler uses COFF library format.

```
cl -I{include-path} {program}.cpp /link -Libpath:{library-path} rulbus-coff.lib
```

GNU C This compiler uses COFF library format.

```
g++ -I{include-path} -L{library-path} -o{program} {program}.cpp -lrulbus-coff
```

LabWindows/CVI This compiler uses OMF/COFF ?? library format.

Todo

find out how to use rulbus.dll with LabWindows/CVI

2.7 Reference Manual

2.7.1 Detailed Description

The Rulbus Device Library contains two groups of functions:

- general library functions, Rulbus DLL Interface
- Rulbus module related functions, for example RB8510 12-bit DAC

When you want to use a specific Rulbus module, look up its documentation page by Rulbus number in the table of Contents, Chapter 2, for example *RB8510 12-bit DAC* for the dual 12-bit DAC and read the documentation for the functions available.

Modules

- Rulbus DLL Interface
  Rulbus DLL interface.
- Generic Rulbus Device
  generic rulbus device.
- RB8506 Parallel Interface
  dual parallel interface (PIA/VIA).
- RB8506 SIFU
  sense interrupt flag unit (BF).
- RB8509 12-bit ADC
  8-channel 12-bit ADC.
- RB8510 12-bit DAC
  dual 12-bit DAC.
- RB8513 Timebase
  programmable timebase.
- RB8514 Time Delay
  programmable time delay.
- RB8515 Clock for Time Delay
  clock for delay module RB8514.
- RB8905 12-bit ADC
  high speed 12-bit ADC (BF).
- RB9005 Instrumentation Amplifier
  programmable instrumentation amplifier for RB8905 12-bit ADC (BF)
- RB9603 Monochromator Controller

monochromator controller (BF).

- PIA Motorola Peripheral Interface Adapter MC6821

  Motorola Peripheral Interface Adapter MC6821 (Pia).

- VIA Rockwell Versatile Interface Adapater R6522

  Rockwell Versatile Interface Adapater R6522 (Via).

2.8 Rulbus DLL Interface

2.8.1 Detailed Description

The general interface of the Rulbus Device Library contains three groups of functions:

- *report* – functions to report on the Rulbus hardware interface used and on the currently open Rulbus devices and to report on the last error that occurred in the library
- *input/output* – functions to enable reading from and writing to the Rulbus directly, via a generic Rulbus device
- *initialization* – the DLL initialization functions DllMain(), rdl_initialize() and rdl_finalize(). The operating system calls DllMain() when it loads or unloads the DLL and DllMain in turns initializes and cleans-up the library for you with rdl_initialize() and rdl_finalize().

For typical use of the Rulbus modules only function rdl_getLastError() is of interest. See dac.cpp. for an example of its use.

For access to the Rulbus for unsupported operations on a module, or for unsupported modules, see section Generic Rulbus Device.

Defines

- #define EXPORT __declspec(dllimport)

  import (export) functions from DLL

- #define RDL_API __cdecl

  calling convention

Typedefs

- typedef char int8

  8-bit signed int

- typedef unsigned char uInt8

  8-bit unsigned int

- typedef **int8** char8

  8-bit signed character

- typedef **uInt8** uChar8

  8-bit unsigned character

- typedef **uInt8** uChar

  8-bit unsigned character

- typedef short int int16

  16-bit signed int

- typedef unsigned short int uInt16
  16-bit unsigned int
- typedef long int32
  32-bit signed int
- typedef unsigned long uInt32
  32-bit unsigned int
- typedef float float32
  32-bit float
- typedef double float64
  64-bit float
- typedef **int32** bool32
  boolean (32-bit)
- typedef char * Cstr
  C-string.
- typedef const char * CCstr
  const C-string

Functions

- EXPORT **int32** RDL_API rdl_initialize ()
  initialize Rulbus Device Library.
- EXPORT **int32** RDL_API rdl_finalize ()
  finalize Rulbus Device Library.
- EXPORT **int32** RDL_API rdl_printRulbusInterface ()
  report Rulbus Interface being used.
- EXPORT **int32** RDL_API rdl_printRulbusDeviceList ()
  report all registered Rulbus devices.
- EXPORT **int32** RDL_API rdl_getLastError (Cstr msg, **int32** maxlen)
  format last error into message buffer.
- EXPORT **int32** RDL_API RulbusDevice_open (**int32** *pHandle, CCstr name)
  open a generic Rulbus device.
- EXPORT **int32** RDL_API RulbusDevice_close (**int32** handle)
  close a generic Rulbus device.
- EXPORT **int32** RDL_API RulbusDevice_print (**int32** handle)

report on specified Rulbus device.

- EXPORT **int32** RDL_API RulbusDevice_putByte (**int32** handle, **int32** offset, **int32** byte)
  write a byte to a generic Rulbus device.
- EXPORT **int32** RDL_API RulbusDevice_getByte (**int32** handle, **int32** offset)
  read a byte from a generic Rulbus device.
- EXPORT **int32** RDL_API RulbusDevice_getRack (**int32** handle, **int32** *pRack)
  get Rulbus device's rack.
- EXPORT **int32** RDL_API RulbusDevice_getAddress (**int32** handle, **int32** *pAddress)
  get Rulbus device's address.
- BOOL EXPORT WINAPI DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
  entry point into dynamic-link library.

2.8.2 Function Documentation

2.8.2.1 BOOL EXPORT WINAPI DllMain (HINSTANCE *hinstDLL*, DWORD *fdwReason*, LPVOID *lpvReserved*)

DllMain() is called by the operating system (or a call to function LoadLibrary()), for example when a process (program) wants to use the DLL. In this case `fdwReason` is DLL_PROCESS_ATTACH and DllMain() calls rdl_initialize() to prepare the library for use.

When the DLL is no longer needed, the operating system calls DllMain() with a `fdwReason` of DLL_PROCESS_DETACH and DllMain() in turn calls rdl_finalize() to clean-up the library.

DllMain() returns TRUE on success, FALSE on error (DLL_PROCESS_ATTACH).

2.8.2.2 EXPORT int32 RDL_API rdl_finalize ()

If the Rulbus Device Library is used as a dynamic-link library (DLL), the library is cleaned-up with this function by DllMain() when the operating system unloads the DLL; you should not call rdl_finalize() yourself.

If however you link the Rulbus Device Library directly to your application, you must call rdl_initialize() before using any other function from the library and when done with the library, you should call rdl_finalize() to clean-up the library.

See also rdl_initialize().

2.8.2.3 EXPORT int32 RDL_API rdl_getLastError (Cstr *msg*, int32 *maxlen*)

rdl_getLastError() copies at most `maxlen` characters (including the terminating \0) of the last error message into the buffer specified by `msg`.

Only one message is retained for all threads using the Rulbus Device Library. If the error message for the thread calling rdl_getLastError() is not available (anymore), the following message is returned:

```
[Rulbus:  error message not available for this thread]
```

Parameters:

msg the message buffer

maxlen the size of the message buffer `msg`

Returns:

0, 1 (Ok, error)
rdl_getLastError() returns 0 if the message could be copied, it returns 1 in the following situations:

- `msg` is NULL
- `maxlen` is zero or less
- the message was not available for this thread

Examples:

dac.cpp, error.cpp, run-daccs.cmd, and run-daccs2.cmd.

2.8.2.4 EXPORT int32 RDL_API rdl_initialize ()

rdl_initialize() determines the Rulbus Interface to use and initializes the list to hold open Rulbus devices. (See Rulbus DLL Implementation, Create Rulbus interface .)

If the Rulbus Device Library is used as a dynamic-link library (DLL), the library is initialized with this function by DllMain() when the operating system loads the DLL; you should not call rdl_initialize() yourself.

If however you link the Rulbus Device Library directly to your application, you must call rdl_initialize() before using any other function from the library. When done with the library, you should call rdl_finalize() to clean-up the library.

See also rdl_finalize().

2.8.2.5 EXPORT int32 RDL_API rdl_printRulbusDeviceList ()

For each opened Rulbus device, rdl_printRulbusDeviceList() prints information for that device to standard output.

2.8.2.6 EXPORT int32 RDL_API rdl_printRulbusInterface ()

rdl_printRulbusInterface() prints one of the following messages to standard output:

- EPP Rulbus Interface at [*0x378*], using *CanIO* port I/O, or
- ISA Rulbus Interface at [*0x200*], using *CanIO* port I/O

Examples:

run-daccs.cmd, and run-daccs2.cmd.

2.8.2.7 EXPORT int32 RDL_API RulbusDevice_close (int32 *handle*)

RulbusDevice_close() closes a generic Rulbus device that was opened with RulbusDevice_open(), or with one of the rb*yydd*_..._open() functions. RulbusDevice_close() returns 0 on success, 1 on error.

2.8.2.8 EXPORT int32 RDL_API RulbusDevice_getByte (int32 *handle*, int32 *offset*)

RulbusDevice_getByte() returns the value read from the Rulbus at the address `offset` bytes from the generic devices' base address as specified with RulbusDevice_open(). On success RulbusDevice_getByte() returns the byte read, otherwise it returns -1.

Returns:
0..255, -1 (byte read, error)

2.8.2.9 EXPORT int32 RDL_API RulbusDevice_open (int32 * *pHandle*, CCstr *name*)

RulbusDevice_open() opens a generic Rulbus device with the specified `name` and passes back a `handle` to it on success.

The `handle` can be used in functions RulbusDevice_putByte(), RulbusDevice_getByte() and RulbusDevice_close() to write to the Rulbus, read from it and close the generic Rulbus device again.

RulbusDevice_open() returns 0 on success, 1 on error.

Examples:
pattern.cpp.

2.8.2.10 EXPORT int32 RDL_API RulbusDevice_print (int32 *handle*)

RulbusDevice_print() prints information for the device specified by `handle` to standard output. `handle` may be obtained with RulbusDevice_open() or one of the rb*yydd*_..._open() functions.

Examples:
run-daccs.cmd, and run-daccs2.cmd.

2.8.2.11 EXPORT int32 RDL_API RulbusDevice_putByte (int32 *handle*, int32 *offset*, int32 *byte*)

RulbusDevice_putByte() writes `byte` to the Rulbus at the address `offset` bytes from the generic devices' base address as specified with RulbusDevice_open(). On success RulbusDevice_putByte() returns the byte written, otherwise it returns -1.

Returns:
0..255, -1 (byte written, error)

2.9 Generic Rulbus Device

Purpose Provide an escape to access the Rulbus for non-supported operations or modules.

Description Sometimes you may want to use an operation on a Rulbus module that is not provided by the modules' interface in this library. Or you may want to use a Rulbus module that is not supported by this library at all.

In these cases the required operations may be constructed with the functions for the Generic Rulbus Device to open a device, read and write a byte and close a device (see Rulbus DLL Interface).

Configuration Not much is known of operations or modules that are not supported by this library.

Default configuration

```
rb_generic "mydevice"
{
   address = 0          # must be zero
}
```

When the Rulbus configuration file is read, the device is unaffected.

Usage For a generic Rulbus device you can use the functions as described in Rulbus DLL Interface

Here is a small snippet of code that shows how to access the Rulbus with the generic Rulbus interface functions.

```
int32 handle;
int   mybase  = 0x12;   // mydevice Rulbus base address
int   myreg0  = 0;      // e.g. register offset for LSB
int   myreg1  = 1;      // e.g. register offset for MSB
int   myvalue = 0x1234; // value to write to mydevice

RulbusDevice_open   ( &handle, "mydevice" );
RulbusDevice_putByte(  handle, mybase + myreg0, myvalue % 256 );  // LSB
RulbusDevice_putByte(  handle, mybase + myreg1, myvalue / 256 );  // MSB
RulbusDevice_close  (  handle );
```

Here is a small program to demonstrate how the generic Rulbus interface functions may be used to continuously write a test pattern to a Rulbus address.

```
/*
 * pattern - write a data pattern to the Rulbus.
 */

#include "rulbus.h"             // header
#include <stdio.h>              // for fprintf()
#include <stdlib.h>             // for EXIT_SUCCESS
#include <conio.h>              // for kbhit()

int error() { return EXIT_FAILURE; }

int main()
{
   int32 pattern = 0x5E;        // test pattern
   int32 rack    = 0;           // rulbus rack number
   int32 addr    = 0;           // rulbus base address
```

```
   int32 offset  = 0x12;        // rulbus address offset
   int32 handle  = 0;           // handle to generic rulbus device
                                // open the generic rulbus device
   if ( RulbusDevice_open( &handle, "Rulbus-test-device" ) )
      return error();

   RulbusDevice_getRack   ( handle, &rack );
   RulbusDevice_getAddress( handle, &addr );

   fprintf( stdout, "Writing [%d:0x%02X] <- 0x%02X\n", rack, addr + offset, pattern );
   fprintf( stderr, "\nPress a key to stop..." );

   while( !kbhit() )            // write pattern until a key is pressed
      if ( 0 > RulbusDevice_putByte( handle, offset, pattern ) );
         return error();

   (void) getch();              // eat character

   if ( RulbusDevice_close( handle ) )
      return error();           // close the generic rulbus device

   return EXIT_SUCCESS;
}
```

2.10 RB8506 Parallel Interface

2.10.1 Detailed Description

Purpose Provide digital inputs and outputs.

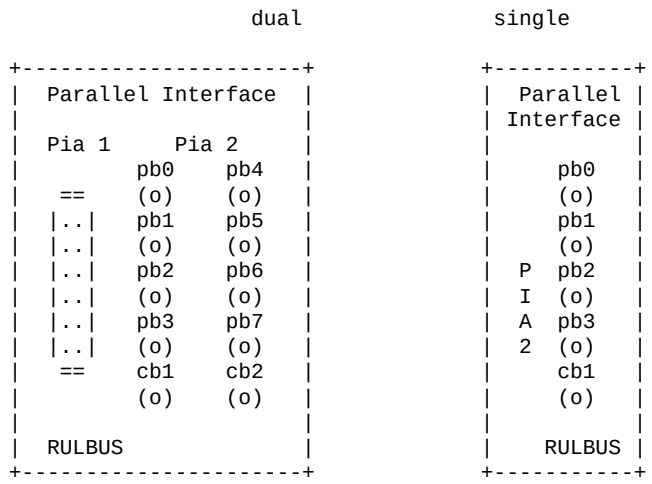
Description Module RB8506 is a (dual) parallel interface. Internally the module may contain Motorola Peripheral Interface Adapter MC6821 (Pia) ICs or Rockwell Versatile Interface Adapater R6522 (Via) ICs or a combination of both.

The following variations of this module are known to exist:

- dual with two Pia's
- dual with a Pia and a Via
- dual with two Via's
- single Pia (small front)

You should create a separate object for each Pia/Via in a Parallel Interface module.

The Parallel Interface front-panels look as follows.



PIA and VIA offer the following interface pins.

port	pin	direction	description
A	PA0..PA7	i/o	level inputs and outputs
A	CA1	input	active edge transition sets interrupt flag
A	CA2	i/o	complex operation
B	PB0..PB7	i/o	level inputs and outputs
B	CB1	input	active edge transition sets interrupt flag
B	CB2	i/o	complex operation

See Pia and Via for a more detailed description of the IC's capabilities.

The 50-pin connector for Pia 1 makes all its pins of Port A and Port B available. Of Pia 2, the BNC-connectors provide access to only Port B pins.

See pport.cpp for an example program. It contains several bit-manipulation functions.

Configuration The RB8506 parallel interface module has no properties that are configured at production time.

Default configuration

```
rb8506_pia "name1"
{
   address = 0x90
}

rb8506_via "name2"
{
   address = 0x90
}
```

When the Rulbus configuration file is read, the parallel interface ports are left unchanged.

Usage There are three groups of functions:

- open and close functions

  type names are "pia" and "via"

- port-related functions:

  port names are "CA", "CB", "PA" and "PB"

- line-releated functions:

  line names are "CA1", "CA2" "PA0".."PA7" and "CB1", "CB2" "PB0".."PB7"

The port-related functions allow you to read or write a complete port at one time, or to get or set its data direction.

However, if you want to act on a single line, the line-related functions are more convenient to use.

The following tables show the port names and the bit-positions for the various peripheral lines as used in the port-related functions and the line names and data-direction specifications for the line-related functions.

```
       bit
port   7  6  5  4  3  2  1  0
------------------------------
"PA"  PA7 .  .  .  .  .  .  PA0
"CA"                   CA2 CA1
"PB"  PB7 .  .  .  .  .  .  PB0
"CB"                   CB2 CB1

 peripheral lines bit-positions
     for port-related functions
       (i/o and data-direction)
```

```
       direction   active edge
line   in    out    neg   pos
------------------------------
"PAn"  'i'   'o'
"CA1"                'n'   'p'
"CA2"  'i'   'o'
"PBn"  'i'   'o'
"CB1"                'n'   'p'
"CB2"  'i'   'o'

peripheral lines data-direction
for line-related functions
(n: 0..7)
```

When setting data direction, a **0** in the bitmask makes a line *input*, whereas a **1** makes it an *output*. For CA1 and CB1, a **0** makes the input act on the negative edge, a **1** makes it sensitive to the positive, upgoing edge.

When setting output, a **0** in the bitmask makes the line level *low*, whereas a **1** makes it *high*.

When reading data-direction from port CA or CB, the CA1 and CA2 (CB1 and CB2) data-direction settings are combined as follows:

```
int dir = ( getDirCA2() != 'i' ) << 1 ) | getEdgeCA1() != 'n';
```

When reading port CA, or CB the CA1 and CA2 (CB1 and CB2) lines are combined as follows:

```
int data = ( getLineCA2() << 1 ) | getIrqCA1();
```

So to set PB0..PB1 and CB2 to output and to set PB2..PB3 to input, you may write:

```
rb8506_pport_setPortDir( handle, "PB", 0x03 );
rb8506_pport_setPortDir( handle, "CB", 0x02 );
```

Note that this ignores the previous data-direction setting: it also sets PB4..PB7 to input.

The following code also sets PB0..PB1 and CB2 to output and PB2..PB3 to input, but leaves the other lines unaffected.

```
rb8506_pport_setLineDir( handle, "PB0", 'o' );
rb8506_pport_setLineDir( handle, "PB1", 'o' );
rb8506_pport_setLineDir( handle, "CB2", 'o' );
rb8506_pport_setLineDir( handle, "PB2", 'i' );
rb8506_pport_setLineDir( handle, "PB3", 'i' );
```

To do the equivalent with the port-related functions you could write:

```
int32 dir;
rb8506_pport_getPortDir( handle, "PB", &dir );
rb8506_pport_setPortDir( handle, "PB", (dir & ~0x0F ) | 0x03 );
rb8506_pport_getPortDir( handle, "CB", &dir );
rb8506_pport_setPortDir( handle, "CB",  dir | 0x02 );
```

With `dir & ~0x0F` we first clear the bits for lines we want to define, PB0..PB3 (0x0F is 00001111 binary).

Note that for CB2 we just as well could write *rb8506_pport_setPortDir(handle, "CB", 0x02)* as before, since it is the only output-capable line of port CB (setting CB1's data-direction is silently ignored).

Bitmasks To address specific pins in port data, you need to use a bitmask. For example, a bitmask of `0x21` specifies pin 0 and pin 5. The mask can also be made with:

```
1 << 5 | 1 << 0
```

Having read direction or data, you use the bitmask to select the pin(s) you are interested in, for example:

```
char dir_pin5 = ( direction_read & (1 << 5) ) ? 'o' : 'i';
```

See pport.cpp for an example program. It contains several bit-manipulation functions.

Open and close

- EXPORT **int32** RDL_API rb8506_pport_open (**int32** *pHandle, CCstr name)

  open a PIA or a VIA.

- EXPORT **int32** RDL_API rb8506_pport_close (**int32** handle)

  close a PIA.

Data direction

- EXPORT **int32** RDL_API rb8506_pport_getPortDir (**int32** handle, CCstr port, **int32** *pDir)
  get port data direction.
- EXPORT **int32** RDL_API rb8506_pport_setPortDir (**int32** handle, CCstr port, **int32** dir)
  set port data direction.
- EXPORT **int32** RDL_API rb8506_pport_getLineDir (**int32** handle, CCstr line, char8 *pDir)
  get line direction.
- EXPORT **int32** RDL_API rb8506_pport_setLineDir (**int32** handle, CCstr line, char8 dir)
  set line direction.

Data input and output

- EXPORT **int32** RDL_API rb8506_pport_getPortData (**int32** handle, CCstr port, **int32** *pData)
  get port data.
- EXPORT **int32** RDL_API rb8506_pport_setPortData (**int32** handle, CCstr port, **int32** data)
  set port data.
- EXPORT **int32** RDL_API rb8506_pport_getLineLevel (**int32** handle, CCstr line, **int32** *pLevel)
  get line level.
- EXPORT **int32** RDL_API rb8506_pport_setLineLevel (**int32** handle, CCstr line, **int32** level)
  set line level.

2.11 RB8506 SIFU

2.11.1 Detailed Description

Purpose Provide digital inputs and outputs.

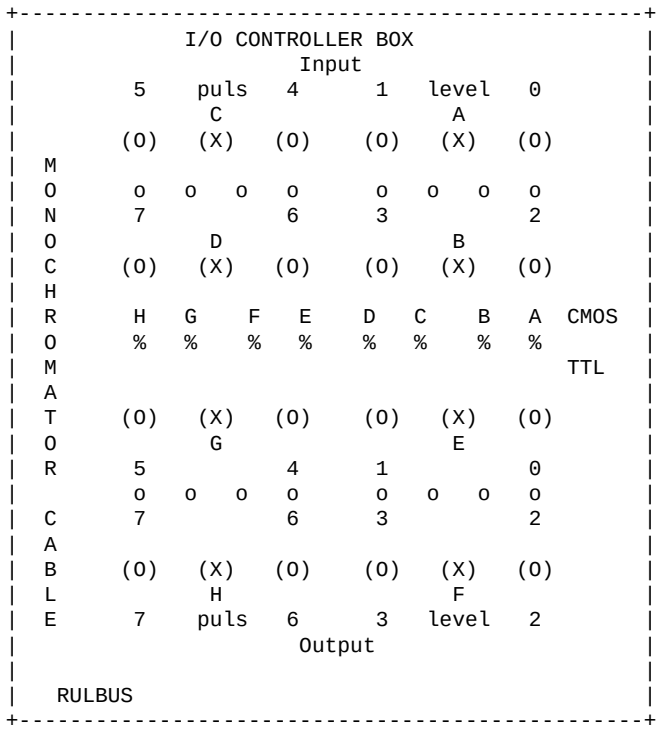
Description Module RB8506, SIFU variant, is a digital input-output interface. It has four level-sensitive inputs, four negative-edge-sensitive inputs, four level outputs and four outputs that can be pulsed as a group.

There is also a provision to check if any of both the negative-edge-sensitive inputs and the level-sensitive inputs have changed state since it was last checked.

Inputs and outputs are available in TTL and in CMOS: TTL or CMOS can be selected with a switch on the front, two inputs or two outputs at the time. A LED indicates if TLL (red) or CMOS (green) is selected.

The SIFU module consists of a PIA parallel interface (see RB8506 Parallel Interface) and a Sense Interrupt Flag Unit (SIFU) board.

The module's front-panel looks as follows.



(O) BNC connector ů (X) red-green LED ů o mini plug ů % switch

The table below indicates which lines are pulse or level inputs-ouputs.

line	kind of input-output
0	]
1	] level
2	]
3	]
4	]

```
5  |  ] negative pulse
6  |  ]           note 1
7  |  ]
```

Note 1: contrary to what the hardware documentation states, I think that the pulse inputs are sensitive to a high-low transition.

Configuration The RB8506 SIFU module has no properties that are configured at production time.

Default configuration

```
rb8506_sifu "name"
{
   address = 0x94
}
```

When the Rulbus configuration file is read, the SIFU is initialized as follows:

- the input port is read to clear any history
- the ouput port is set to zero

Usage There are functions to read all inputs at a time or to read each input line individually. Further there are functions to check if any line has changed since the last time it was checked. Ouputs may be read or written all at once or each output may be read or written individually. Four pulse outputs can be pulsed at one time.

```
int32 handle;
int32 flag;

rb8506_sifu_open( &handle, "sifu" );

rb8506_sifu_isChangedInputLine( handle, &flag );

if ( flag )
{
   int32 data;

   rb8506_sifu_getInputPortData ( handle, &data );
   rb8506_sifu_setOutputPortData( handle,  data );
}

rb8506_sifu_close( handle );
```

Open and close

- EXPORT **int32** RDL_API rb8506_sifu_open (**int32** *pHandle, CCstr name)

  open a sifu.

- EXPORT **int32** RDL_API rb8506_sifu_close (**int32** handle)

  close a sifu.

Data input

- EXPORT **int32** RDL_API rb8506_sifu_getInputPortData (**int32** handle, **int32** *pData)

  get the input port data; also clears input flip-flops.

- EXPORT **int32** RDL_API rb8506_sifu_isChangedInputLine (**int32** handle, **int32** *pFlag)

  true if the level of one or more input lines have changed; clears input-has-changed flag; see also rb8506_sifu_tstInputLineLevel().

- EXPORT **int32** RDL_API rb8506_sifu_tstInputLineLevel (**int32** handle, **int32** line, **int32** *pLevel)

  get the level of the specified input line; clears input-has-changed flag, retains state of input flip-flops; see also rb8506_sifu_isChangedInputLine().

- EXPORT **int32** RDL_API rb8506_sifu_getInputLineLevel (**int32** handle, **int32** line, **int32** *pLevel)

  get the level of the specified input line; clears input flip-flops.

Data ouput

- EXPORT **int32** RDL_API rb8506_sifu_getOutputPortData (**int32** handle, **int32** *pData)

  get the output port data.

- EXPORT **int32** RDL_API rb8506_sifu_setOutputPortData (**int32** handle, **int32** data)

  set the output port data.

- EXPORT **int32** RDL_API rb8506_sifu_getOutputLineLevel (**int32** handle, **int32** line, **int32** *pLevel)

  get the line level of the specified output line.

- EXPORT **int32** RDL_API rb8506_sifu_setOutputLineLevel (**int32** handle, **int32** line, **int32** level)

  set or clear specified output line.

Pulse control

- EXPORT **int32** RDL_API rb8506_sifu_isEnabledPulseOutputs (**int32** handle, **int32** *pFlag)

  true if the pulse outputs are enabled.

- EXPORT **int32** RDL_API rb8506_sifu_enablePulseOutputs (**int32** handle, **int32** flag)

  enable or disable pulse outputs.

2.12 RB8509 12-bit ADC

2.12.1 Detailed Description

Purpose Measure a voltage.

Description Module RB8509 is a 8-channel 12-bit analog to digital converter (ADC).

The module has eight analog inputs and a trigger input to start a data acquisition. A data acquisition can also be started by a software trigger.

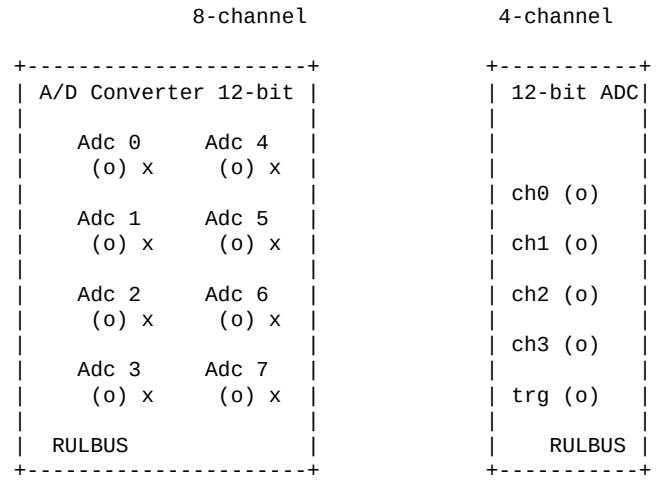
Module RB8509 contains the following parts:

- a channel selector to select one of the eight inputs [ch0..ch7]
- a gain selector to select one of the four gains [1, 2, 4, 8]
- a 12-bit data Analog to Digital Converter

The following variations of this module are known to exist:

- 8-channel input, no external trigger input
- 4-channel input

The module's front-panels look as follows.



Configuration At production time, the ADC is configured for unipolar or bipolar operation and its input voltage range is defined as 10 or 20V.

The following table shows the volt-per-bit values for the various input voltage ranges.

Voltage range	volt-per-bit	bipolar
0 .. 10.2375 V	2.5 mV	0
-5.12 .. 5.1175 V	2.5 mV	1
-10.24 .. 10.235 V	5 mV	1

Default configuration

```
rb8509_adc12 "name"
{
   address = 0xC0
   bipolar = true
   volt_per_bit = 5e-3
}
```

When the Rulbus configuration file is read, ADCs are configured as follows:

- external trigger disabled
- gain 1 selected
- channel 0 selected

Usage Among others, there are functions to select channel and gain and to acquire and obtain the input voltage of the selected input.

```
int32     handle;
const int channel = 0;
float32   voltage = 0;

rb8509_adc12_open( &handle, "adc" );

rb8509_adc12_setChannel( handle, channel  );
rb8509_adc12_autoscale ( handle           );
rb8509_adc12_getVoltage( handle, &voltage );

rb8509_adc12_close( handle )
```

Open and close

- EXPORT **int32** RDL_API rb8509_adc12_open (**int32** ∗pHandle, CCstr name)

  open an ADC.

- EXPORT **int32** RDL_API rb8509_adc12_close (**int32** handle)

  close an ADC

Channel selection

- EXPORT **int32** RDL_API rb8509_adc12_getChannel (**int32** handle, **int32** ∗pChannel)

  get currently selected channel [0..7].

- EXPORT **int32** RDL_API rb8509_adc12_setChannel (**int32** handle, **int32** channel)

  select channel [0..7].

Gain selection

- EXPORT **int32** RDL_API rb8509_adc12_getGain (**int32** handle, **int32** ∗pGain)

  get currently selected gain [1,2,4,8].

- EXPORT **int32** RDL_API rb8509_adc12_setGain (**int32** handle, **int32** gain)

  select gain [1,2,4,8].

- EXPORT **int32** RDL_API rb8509_adc12_autoscale (**int32** handle)

  determine and set amplifier gain to fit current signal.

Data acquisition

- EXPORT **int32** RDL_API rb8509_adc12_convert (**int32** handle)

  start an analog to digital conversion.

- EXPORT **int32** RDL_API rb8509_adc12_isReady (**int32** handle, **int32** ∗pFlag)

  true if result is available; flag is reset by reading result.

- EXPORT **int32** RDL_API rb8509_adc12_getVoltage (**int32** handle, **float32** ∗pVoltage)

  issue software trigger and return input-voltage measured, may return 1 on timeout; OR wait for external trigger and return input-voltage measured.

- EXPORT **int32** RDL_API rb8509_adc12_getValue (**int32** handle, **int32** ∗pValue)

  issue software trigger and return conversion value, may return 1 on timeout; OR wait for external trigger and return conversion value.

Triggering

- EXPORT **int32** RDL_API rb8509_adc12_getExtTriggerLevel (**int32** handle, **int32** ∗pFlag)

  the current external trigger level.

- EXPORT **int32** RDL_API rb8509_adc12_isEnabledExtTrigger (**int32** handle, **int32** ∗pFlag)

  true if external trigger-input is enabled.

- EXPORT **int32** RDL_API rb8509_adc12_enableExtTrigger (**int32** handle, **int32** flag)

  enable or disable external trigger-input.

Configuration

- EXPORT **int32** RDL_API rb8509_adc12_isBipolar (**int32** handle, **int32** ∗pFlag)

  true if ADC has bipolar configuration.

- EXPORT **int32** RDL_API rb8509_adc12_getVoltperbit (**int32** handle, **float32** ∗pVpb)

  the ADC's input sensitivity configuration.

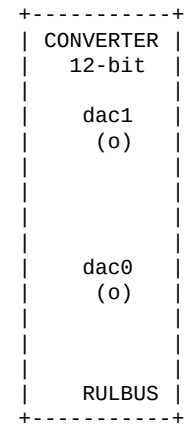
2.13 RB8510 12-bit DAC

2.13.1 Detailed Description

Purpose Generate a unipolar or bipolar voltage.

Description Module RB8510 is a dual 12-bit Digital to Analog Converter. It contains two 12-bit Digital to Analog Converters (DACs), each connecting to a BNC connector on the front panel. There are no inputs.

The module's front-panel looks as follows.



Configuration At production time, each DAC is configured for unipolar or bipolar operation and its output voltage range is defined as 5, 10 or 20V.

The following table shows the volt-per-bit values for the various output voltage ranges.

Voltage range	volt-per-bit	bipolar
0 .. 5.11875 V	1.25 mV	0
0 .. 10.2375 V	2.5 mV	0
0 .. 20.475 V	5 mV	0
-5.12 .. 5.1175 V	2.5 mV	1
-10.24 .. 10.235 V	5 mV	1

Default configuration

```
rb8510_dac12 "name"
{
   address = 0xD0
   bipolar = true
   volt_per_bit = 5e-3
}
```

When the Rulbus configuration file is read, DAC-outputs are set to 0 Volt.

Usage There are functions to set and get the output voltage and functions to set and get the DAC-register code.

```
int32 handle;

rb8510_dac12_open( &handle, "dac-ch0" );
rb8510_dac12_setVoltage( 1.23 );
rb8510_dac12_close( handle );
```

For each channel of the DAC module, a separate object must be created.

See dac.cpp for a complete example.

Open and close

- EXPORT **int32** RDL_API rb8510_dac12_open (**int32** *pHandle, CCstr name)
  open a DAC.
- EXPORT **int32** RDL_API rb8510_dac12_close (**int32** handle)
  close a DAC.

Voltage output

- EXPORT **int32** RDL_API rb8510_dac12_getVoltage (**int32** handle, **float32** *pVoltage)
  get current DAC voltage.
- EXPORT **int32** RDL_API rb8510_dac12_setVoltage (**int32** handle, **float32** voltage)
  set DAC to new voltage.

Value output

- EXPORT **int32** RDL_API rb8510_dac12_getValue (**int32** handle, **int32** *pValue)
  get current DAC register value.
- EXPORT **int32** RDL_API rb8510_dac12_setValue (**int32** handle, **int32** value)
  set DAC register to new value.

Configuration

- EXPORT **int32** RDL_API rb8510_dac12_isBipolar (**int32** handle, **int32** *pFlag)
  true if DAC has bipolar configuration.
- EXPORT **int32** RDL_API rb8510_dac12_getVoltperbit (**int32** handle, **float32** *pVpb)
  set DAC register to new code.

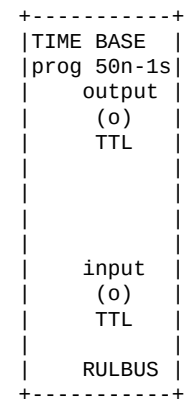
2.14 RB8513 Timebase

2.14.1 Detailed Description

Purpose Generate short pulses with a programmable interval time.

Description Module RB8513 is a clock signal generator with a programmable interval time. It has a TTL trigger input and a TTL clock output.

The module's front-panel looks as follows.



The module can generate a clock signal with the following interval times.

```
 Programmable interval times
--------------------------------
  50 ns
 100 ns
 200 ns
 500 ns
   1 us -  99   us, step   1 us
 100 us -   9.9 ms, step 100 us
  10 ms - 990   ms, step  10 ms
```

Configuration The RB8513 timebase module has no properties that are configured at production time.

Default configuration

```
rb8513_timebase "name"
{
   address = 0xB0
}
```

When the Rulbus configuration file is read, the timebase interval time is set to 1 us.

Usage There are functions to set and get the interval time.

```
int32 handle;

rb8513_timebase_open( &handle, "timebase" );
rb8513_timebase_setIntervalTime( 1e-3 );
rb8513_timebase_close( handle )
```

Open and close

- EXPORT **int32** RDL_API rb8513_timebase_open (**int32** ∗pHandle, CCstr name)

  open a timebase.

- EXPORT **int32** RDL_API rb8513_timebase_close (**int32** handle)

  close a timebase.

Control

- EXPORT **int32** RDL_API rb8513_timebase_stop (**int32** handle)

  stop timebase.

Interval Time

- EXPORT **int32** RDL_API rb8513_timebase_getIntervalTime (**int32** handle, **float32** ∗pTime)

  get the current interval time in s.

- EXPORT **int32** RDL_API rb8513_timebase_setIntervalTime (**int32** handle, **float32** time)

  set set interval time to 50 ns, 100 ns, 200 ns, 500 ns, 1..99 us, 0.1..9.9 ms, 10..990 ms.

2.15 RB8514 Time Delay

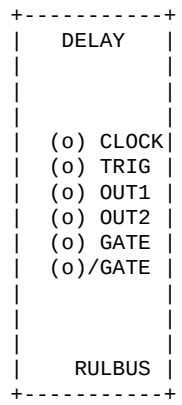
2.15.1 Detailed Description

Purpose Generate a programmable time delay.

Description Module RB8514 is a programmable delay time generator.

It has a clock input, a trigger input, two delay outputs, a gate and a gate-not output, and a Rulbus interrupt output.

The module's front-panel looks as follows.



The delay is made with a 24-bit down counter that counts a programmable number of clock-pulses, when started by a trigger-pulse on the trigger input. See setClockFrequency(), setIntrinsicDelayTime(), setDelayTime(), setDelayCount(), isBusy().

The trigger input can be made positive or negative edge sensitive. Besides triggering the delay with the trigger-input, it is also possible to trigger the delay with a software command. See setSignalDirection() and trigger().

Each of the two delay outputs (1 and 2) can be programmed to generate a short pulse (15 ns) at the start of the delay, at the end of it, both or to generate no pulse at all. See enableSignal().

The level of the start- and end-pulses of output 1 and 2 can be made positive or negative. The restriction is that both outputs have the same polarity for the same pulse. See setSignalDirection().

An additional restriction with negative pulses is, that only the start- *or* the end-pulse may be enabled on an output.

The gate and gate-not outputs generate a pulse from the start of the delay time upto the end of it. The gate-output is active high, the gate-not output is active low.

Configuration The RB8514 delay module has no attributes that are configured at production time.

Default configuration

```
rb8515_delay "name"
{
   address = 0xC4
}
```

When the Rulbus configuration file is read, delay modules are initialized as follows:

- clock frequency 1 Hz
- delay time 1 s
- all signals positive
- start and stop pulses enabled

Usage Among others, there are functions to specify the delay module's clock frequency and signal polarities and signal enable-states and to specify the module's delay time.

```
int32 handle;

rb8514_delay_open( &handle, "delay" );

rb8514_delay_setClockFrequency( handle, 10e6 );
rb8514_delay_enableSignal     ( handle, "s1",  1 );
rb8514_delay_enableSignal     ( handle, "e1",  0 );
rb8514_delay_enableSignal     ( handle, "s2",  0 );
rb8514_delay_enableSignal     ( handle, "e2",  1 );
rb8514_delay_setDelayTime     ( handle, 1e-3 );

rb8514_delay_close( handle )
```

Open and close

- EXPORT **int32** RDL_API rb8514_delay_open (**int32** ∗pHandle, CCstr name)

  open a delay module.

- EXPORT **int32** RDL_API rb8514_delay_close (**int32** handle)

  close a delay module .

Clock frequency

- EXPORT **int32** RDL_API rb8514_delay_getClockFrequency (**int32** handle, **float32** ∗pFrequency)

  get the current clock frequency.

- EXPORT **int32** RDL_API rb8514_delay_setClockFrequency (**int32** handle, **float32** frequency)

  set clock frequency.

Delay

- EXPORT **int32** RDL_API rb8514_delay_getIntrinsicDelayTime (**int32** handle, **float32** ∗pTime)

  get the current intrinsic delay time (the time subtracted from requested delay time).

- EXPORT **int32** RDL_API rb8514_delay_setIntrinsicDelayTime (**int32** handle, **float32** time)

  set intrinsic delay time (will be subtracted from requested delay time).

- EXPORT **int32** RDL_API rb8514_delay_getDelayTime (**int32** handle, **float32** ∗pTime)

get the current delay time.

- EXPORT **int32** RDL_API rb8514_delay_setDelayTime (**int32** handle, **float32** time)

  set new delay time.

- EXPORT **int32** RDL_API rb8514_delay_getDelayCount (**int32** handle, **int32** *pCount)

  get the current delay count.

- EXPORT **int32** RDL_API rb8514_delay_setDelayCount (**int32** handle, **int32** count)

  set new delay count.

Signals

- EXPORT **int32** RDL_API rb8514_delay_getSignalDirection (**int32** handle, CCstr signal, **int32** *pDir)

  get the current signal direction: signal in [t s1 s2 e1 e2], result: [p n].

- EXPORT **int32** RDL_API rb8514_delay_setSignalDirection (**int32** handle, CCstr signal, **int32** dir)

  set new signal direction: signal in [t s1 s2 e1 e2], dir in [p n i] for positive, negative and invert.

- EXPORT **int32** RDL_API rb8514_delay_isEnabledSignal (**int32** handle, CCstr signal, bool32 *pEnabled)

  get the current signal enable-state: signal in [i s1 s2 e1 e2].

- EXPORT **int32** RDL_API rb8514_delay_enableSignal (**int32** handle, CCstr signal, bool32 enable)

  enable signal: signal in [i s1 s2 e1 e2], i for interrupt.

Timing

- EXPORT **int32** RDL_API rb8514_delay_trigger (**int32** handle)

  software trigger: start delay.

- EXPORT **int32** RDL_API rb8514_delay_isBusy (**int32** handle, bool32 *pBusy)

  true if timing a delay.

2.16 RB8515 Clock for Time Delay

2.16.1 Detailed Description

Purpose Generate a clock signal for the time delay module RB8514.

Description Module RB8515 is a clock signal generator for time delay module RB8514.

It provides fixed frequency outputs and clock outputs with programmable frequency output with programmable frequency via SMB connectors on the front panel. The module can also generate a Rulbus interrupt with a programmable frequency, but it is not made available by this driver. There are no inputs.

The following front-panel variations of this module exist:

- seven identical outputs with programmable frequency
- four different fixed-frequency output and three identical outputs with programmable frequency

The module's front-panels look as follows.

```
fixed & programmable     programmable only
  +-----------+            +-----------+
  |   CLOCK   |            |   CLOCK   |
  |           |            |           |
  |           |            |           |
  |   (o) 100M|            |   (o)     |
  |   (o) 10M |            |   (o)     |
  |   (o) 1M  |            |   (o)     |
  |   (o) 100k|            |   (o)     |
  |SELECTABLE |            |   (o)     |
  |   (o)     |            |   (o)     |
  |   (o)     |            |   (o)     |
  |   (o)     |            |           |
  |           |            |           |
  |           |            |           |
  |    RULBUS |            |    RULBUS |
  +-----------+            +-----------+
```

The following frequencies are available.

```
Output        | Description
--------------+------------------------------------------------------------
Ffixed        | fixed frequencies:
              | 100 kHz, 1 MHz, 10 MHz and 100 MHz
              |
Fprogrammable | programmable frequencies:
              | 0 Hz (disabled), 100 Hz, 1 kHz, 10 kHz, 100 kHz, 1 MHz, 10 MHz and 100 MHz.
```

Configuration The RB8515 clock module has no properties that are configured at production time.

Default configuration

```
rb8515_clock "name"
{
   address = 0xC8
}
```

When the Rulbus configuration file is read, the programmable clock outputs are set to 1 kHz.

Usage There are functions to set and get the clock frequency.

```
int32 handle;

rb8515_clock_open( &handle, "clock" );
rb8515_clock_setClockFrequency( 10e6 );
rb8515_clock_close( handle )
```

Open and close

- EXPORT **int32** RDL_API rb8515_clock_open (**int32** *pHandle, CCstr name)

  open a clock.

- EXPORT **int32** RDL_API rb8515_clock_close (**int32** handle)

  close a clock.

Clock frequency

- EXPORT **int32** RDL_API rb8515_clock_getClockFrequency (**int32** handle, **float32** *pFrequency)

  get current clock frequency.

- EXPORT **int32** RDL_API rb8515_clock_setClockFrequency (**int32** handle, **float32** frequency)

  set clock frequency.

2.17 RB8905 12-bit ADC

2.17.1 Detailed Description

Purpose Measure a voltage trace.

Description Module RB8905 is a 1 Mega-sample per second 12-bit analog to digital converter with a 32 kByte on-board data buffer.

The module has a single analog input and a clock input to do a series of data acquisitions. The number of samples to store on-board is programmable.

Module RB8905 contains the following parts:

- a track and hold
- a 12-bit ADC
- an address counter
- a 32 kByte on-board data buffer

The ADC can be programmed for bipolar or unipolar use. The input voltage span of the ADC is configured at production time for 10 V or 20 V.

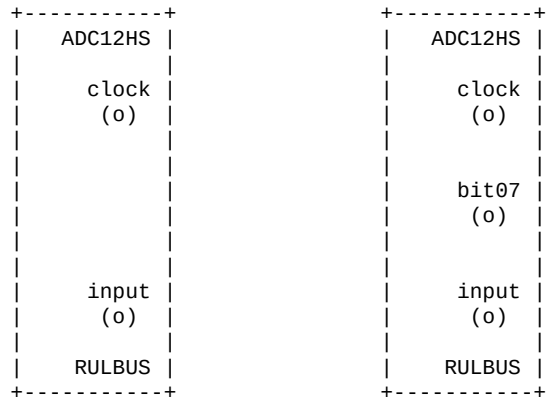
The analog to digital converter IC is an ADC601 manufactured by Burr-Brown. This 12-bit ADC normally converts samples at a rate of 1 Ms/s (900 ns), but it also can convert at a rate of 2 Ms/s (400 ns). In the latter case, only eight bits are determined and this mode of operation is called short-cycle mode (fast mode, 8-bit result).

The number of samples to store on-board can be specified in one of two ways (normal conversion mode assumed):

1. 1–128 samples: specified by JMP3 on the module's board
2. 256–16384 samples: specified programmatically

The number of samples to store on-board can be specified with function rb8905_adc12_setBuffer-Capacity(). Specifiying zero for the number of samples, or capacity, selects the JMP3 setting. For 256 and more samples, the number of samples is a power of two: 256, 512, 1024 etc.

The module's front-panels look as follows.



Note that bit07 on the right front-panel provides the signal that indicates the end of a series of data acquisitions, or that indicates that all samples have been read, when reading.

Configuration At production time, the ADC is configured for an input voltage range of 10 V or 20 V.

The following table shows the volt-per-bit values for the various input voltage spans.

```
 Voltage span  | volt-per-bit   bipolar
---------------+-------------------------
    0 .. 10 V  |  2.442002 mV   no
   -5 ..  5 V  |  2.442002 mV   yes
  -10 .. 10 V  |  4.884004 mV   yes
```

Note that when the ADC's fast conversion mode (8-bit result) is used, the input voltage span is reduced with a factor of 16 (2^4). See rb8905_adc12_getMinInputVoltage() and rb8905_adc12_getMaxInputVoltage().

Default configuration

```
rb8905_adc12 "name"
{
   address = 0xBC
   volt_per_bit = 2.442002 m
}
```

When the Rulbus configuration file is read, an ADC is configured as follows:

- normal conversion mode
- bipolar mode
- 256 samples
- Rulbus interrupt disabled

Usage There are functions to set the conversion mode, to set the input voltage polarity, to set the number of samples to acquire and to read the input as ADC- value or as voltage at the ADC's input.

```
#define N 512

int32 handle;
int32 nread;
float32 trace[ N ];

rb8905_adc12_open( &handle, "adc" );

rb8905_adc12_setBufferCapacity( handle, N );
rb8905_adc12_arm();

while ( !rb8905_adc12_isReady( handle ) )
   wait();

rb8905_adc12_readVoltage( handle, trace, N, &nread );

rb8905_adc12_close( handle );
```

Open and close

- EXPORT **int32** RDL_API rb8905_adc12_open (**int32** ∗pHandle, CCstr name)

  open an ADC.

- EXPORT **int32** RDL_API rb8905_adc12_close (**int32** handle)

  close an ADC.

Data acquisition

- EXPORT **int32** RDL_API rb8905_adc12_arm (**int32** handle)

  arm ADC for data-acquisition; a sample is taken each clock-pulse.

- EXPORT **int32** RDL_API rb8905_adc12_isReady (**int32** handle, **int32** ∗pFlag)

  true if all samples have been measured or if all samples have been read-out.

- EXPORT **int32** RDL_API rb8905_adc12_readVoltage (**int32** handle, **float32** ∗pVoltage, **int32** nelem, **int32** ∗nread)

  read nelem elements as voltage from the on-board data buffer, pass number of elements actually read in nread; **note** *that the voltage datatype is* **float32**.

- EXPORT **int32** RDL_API rb8905_adc12_readValue (**int32** handle, **int16** ∗pValue, **int32** nelem, **int32** ∗nread)

  read nelem elements as value from the on-board data buffer, pass number of elements actually read in nread; the 8-bit/12-bit unipolar samples are presented as (unsigned) magnitude codes, bipolar values are presented as offset binary codes; **note** *that the value datatype is* **int16**.

On-board buffer

- EXPORT **int32** RDL_API rb8905_adc12_getBufferCapacity (**int32** handle, **int32** ∗pCapacity)

  get the number of samples to acquire in the on-board buffer.

- EXPORT **int32** RDL_API rb8905_adc12_getJmp3BufferCapacity (**int32** handle, **int32** ∗pCapacity)

  get the JMP3 setting for the number of samples to store on-board.

- EXPORT **int32** RDL_API rb8905_adc12_setBufferCapacity (**int32** handle, **int32** capacity)

  set the number of samples to acquire in the buffer; use JMP3 setting when zero is specified.

Configuration

- EXPORT **int32** RDL_API rb8905_adc12_isFastMode (**int32** handle, **int32** ∗pFlag)

  true if fast (short-cycle) mode is selected (8-bit data).

- EXPORT **int32** RDL_API rb8905_adc12_setFastMode (**int32** handle, **int32** isfast)

  set converter to fast or short-cycle mode (8-bit data, isfast is true), or set converter to normal mode (12-bit data, isfast is false).

- EXPORT **int32** RDL_API rb8905_adc12_isUnipolarMode (**int32** handle, **int32** ∗pFlag)
  true if unipolar mode is selected.
- EXPORT **int32** RDL_API rb8905_adc12_setUnipolarMode (**int32** handle, **int32** isunipolar)
  set converter to unipolar mode (isunipolar is true), or set converter to bipolar mode (isunipolar is false).
- EXPORT **int32** RDL_API rb8905_adc12_getVoltperbit (**int32** handle, **float32** ∗pVpb)
  get the ADC input sensitivity configuration.
- EXPORT **int32** RDL_API rb8905_adc12_getVoltageSpan (**int32** handle, **float32** ∗pSpan)
  get the ADC input voltage span.
- EXPORT **int32** RDL_API rb8905_adc12_getOffsetVoltage (**int32** handle, **float32** ∗pVoltage)
  get the ADC offset voltage.
- EXPORT **int32** RDL_API rb8905_adc12_getMinInputVoltage (**int32** handle, **float32** ∗pVoltage)
  get the lowest acceptable ADC input voltage; takes mode settings into account.
- EXPORT **int32** RDL_API rb8905_adc12_getMaxInputVoltage (**int32** handle, **float32** ∗pVoltage)
  get the highest acceptable ADC input voltage; takes mode settings into account.

2.18 RB9005 Instrumentation Amplifier

2.18.1 Detailed Description

Purpose Provide programmable amplifiers and filters to form a data-acquisition system with the high speed 12-bit ADC RB8905.

Description Module RB9005 is a 4-channel programmable amplifier-filter module.

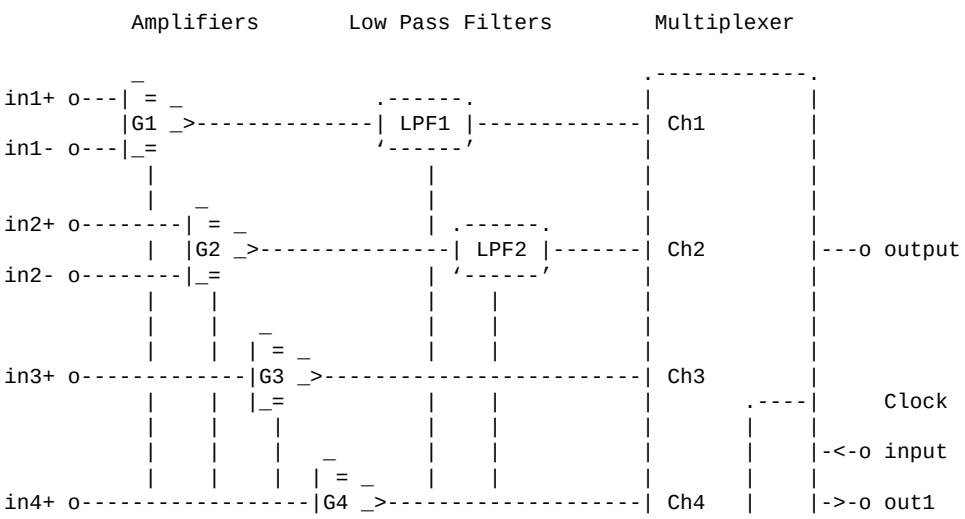
The module's front-panel looks as follows.

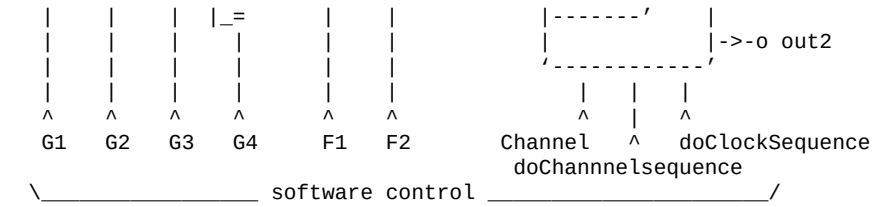
```
+----------------------+
|     Programable      |
|  Differential Amp.   |
|   +input1   input3   |
|     (o)       (o)    |
|                      |
|   -input1   input4   |
|     (o)       (o)    |
|                      |
|   +input2    ouput   |
|     (o)       (o)    |
|              clock   |
|   -input2   in o1 o2 |
|     (o)    (o)(o)(o) |
|                      |
|  RULBUS              |
+----------------------+
```

It has two differential programmable amplifiers followed by a programmable filter, two unbalanced input (single-ended) programmable amplifiers and an ouput multiplexer to present one, two, three or all four inputs in sequence.

```
1-1-1-1   ů   2-1-2-1   ů   3-2-1-3   ů   4-3-2-1
```

Further the module has a clock input and two clock outputs that can be programmed to duplicate the clock input or to follow an alternating sequence so that out1 presents a clock signal for the odd channels and out2 presents a clock signal for the even channels in the selected channel sequence. (Apparently this is not very useful for the 3-2-1-3 sequence where channel 3 appears at odd and even positions in the sequence.)





The following table shows the module's gain and low-pass filter characteristics.

Channel	Gain	F-LowPass [kHz]	programmable
1, 2	10, 20, 40, 80	>= 1000	(no filter)
	100, 200, 400, 800	100	yes
	1000, 2000, 4000, 8000	10	yes
	10000, 20000, 40000, 80000	1	yes
		0.1	yes
3, 4	1	3000	no
	10	:	no
	100	:	no
	1000	312	no

Configuration The RB9005 amplifier module has no properties that are configured at production time.

Default configuration

```
rb9005_amplifier "name"
{
   address = 0xB3
}
```

When the Rulbus configuration file is read, an amplifier is configured as follows:

- channel 1 selected
- gain channel 1 and 2 set to 10
- gain channel 3 and 4 set to 1
- low pass frequency channel 1 and 2 set to >= 1 MHz (no filter)
- channel sequence disabled
- clock sequence disabled

Usage There are functions to select and obtain the channel, the gain and the low pass filter frequencies (channel1 and 2), and there are functions to enable or disable the channel and clock sequence modes.

```
int32 handle;

rb9005_amplifier_open( &handle, "amplifier" );
rb9005_amplifier_setChannel( handle, 1 );
rb9005_amplifier_setGain( handle, 1, 10 );
rb9005_amplifier_setLowPassFrequency( handle, 1, 1e4 );
rb9005_amplifier_close( handle )
```

Open and close

- EXPORT **int32** RDL_API rb9005_amplifier_open (**int32** *pHandle, CCstr name)

  open an amplifier.

- EXPORT **int32** RDL_API rb9005_amplifier_close (**int32** handle)

  close an amplifier.

Channel, gain and filters

- EXPORT **int32** RDL_API rb9005_amplifier_getChannel (**int32** handle, **int32** *pChannel)

  get the current channel (1..4).

- EXPORT **int32** RDL_API rb9005_amplifier_setChannel (**int32** handle, **int32** channel)

  select the specified channel, or channel sequence (1..4).

- EXPORT **int32** RDL_API rb9005_amplifier_getGain (**int32** handle, **int32** channel, **int32** *pGain)

  get the channel's current gain.

- EXPORT **int32** RDL_API rb9005_amplifier_setGain (**int32** handle, **int32** channel, **int32** gain)

  set gain for specified channel.

- EXPORT **int32** RDL_API rb9005_amplifier_getLowPassFrequency (**int32** handle, **int32** channel, **float32** *pFreq)

  the channel's current low pass frequency (channel 1 & 2 only).

- EXPORT **int32** RDL_API rb9005_amplifier_setLowPassFrequency (**int32** handle, **int32** channel, **float32** freq)

  set low pass frequency (channel 1 & 2 only).

Channel and clock sequencing

- EXPORT **int32** RDL_API rb9005_amplifier_isChannelSequenceMode (**int32** handle, **int32** *pFlag)

  true if a channel sequence is active.

- EXPORT **int32** RDL_API rb9005_amplifier_setChannelSequenceMode (**int32** handle, **int32** flag)

  enable or disable channel sequencing.

- EXPORT **int32** RDL_API rb9005_amplifier_isClockSequenceMode (**int32** handle, **int32** *pFlag)

  true if a clock sequence is active.

- EXPORT **int32** RDL_API rb9005_amplifier_setClockSequenceMode (**int32** handle, **int32** flag)

  enable or disable clock sequencing.

2.19 RB9603 Monochromator Controller

2.19.1 Detailed Description

Purpose Control a Bausch & Lomb monochromator remotely.

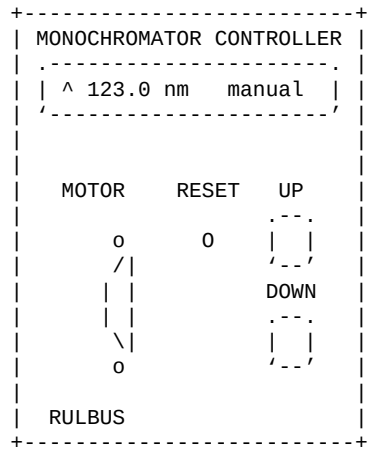
Description The RB9603 Rulbus module enables remote control of a Bausch & Lomb monochromator by driving its nonius with a stepping motor.

The target wavelength can be set manually with Up and Down keys on the Rulbus module and programmatically via the Rulbus computer interface.

The module has a one-line display that shows the target wavelength along with indications if the monochromator is still moving up or down and if the wavelength was set manually or via program control.

Besides setting the wavelength, several properties may be requested from the controller via the computer interface: the target wavelength, the current wavelength, the minimum and maximum wavelengths and the current wavelength as seen by the potentiometer on the stepping motor–potentiometer assembly (see below).

The module's front-panel looks as follows.



The controller module itself sits in a Rulbus rack. A stepping motor–potentiometer assembly is located on the Bausch & Lomb monochromator to drive it. Via a cable the assembly is connected to the MOTOR connector on the controller.

The potentiometer in the assembly records the absolute position of the monochromator. It is used to calibrate the monochromator wavelength at 500 nm initially and to check the position when a new wavelength has been reached.

Configuration The minimum wavelength that can be used can be set at production time with jumper CN4 to be 0 nm or 100 nm. Minimum and maximum wavelength may be obtained with functions rb9603_monochromator_getMinWavelength() and rb9603_monochromator_getMaxWavelength().

Default configuration

```
rb9603_monochromator "name"
{
   address = 0xCE
}
```

When the Rulbus configuration file is read, no configuration steps are made with respect to the monochromator Rulbus module.

Usage Besides a function to set the target wavelength, there are several functions to request properties, for example: the target wavelength, the current wavelength, the minimum and maximum wavelengths and the current wavelength as seen by the potentiometer on the stepping motor–potentiometer assembly.

```
int32 handle;
int32 and_wait = true;

rb9603_monochromator_open( &handle, "lamp" );
rb9603_monochromator_setTargetWavelength( handle, 500.0e-9, and_wait );  // in [m]
rb9603_monochromator_close( handle )
```

Open and close

- EXPORT **int32** RDL_API rb9603_monochromator_open (**int32** ∗pHandle, CCstr name)

  open a monochromator.

- EXPORT **int32** RDL_API rb9603_monochromator_close (**int32** handle)

  close a monochromator.

Status, wait and calibrate

- EXPORT **int32** RDL_API rb9603_monochromator_isReady (**int32** handle, **int32** ∗pFlag)

  pass true if monochromator is at target wavelength.

- EXPORT **int32** RDL_API rb9603_monochromator_wait (**int32** handle)

  wait until monochromator is at target wavelength.

- EXPORT **int32** RDL_API rb9603_monochromator_calibrate (**int32** handle, **int32** and_wait)

  move monochromator to 500 nm for calibration.

Wavelength

- EXPORT **int32** RDL_API rb9603_monochromator_setTargetWavelength (**int32** handle, **float32** length, **int32** and_wait)

  set target wavelength [m].

- EXPORT **int32** RDL_API rb9603_monochromator_getTargetWavelength (**int32** handle, **float32** ∗pLength)

  get the target wavelength [m].

- EXPORT **int32** RDL_API rb9603_monochromator_getCurrentWavelength (**int32** handle, **float32** ∗pLength)

  get the current wavelength [m].

- EXPORT **int32** RDL_API rb9603_monochromator_getAdcWavelength (**int32** handle, **float32** *p-Length)

  get the current wavelength [m] according to the measured voltage.

- EXPORT **int32** RDL_API rb9603_monochromator_getAdcMeanWavelength (**int32** handle, **float32** *pLength)

  get the current wavelength [m] according to the mean-measured voltage.

- EXPORT **int32** RDL_API rb9603_monochromator_getMinWavelength (**int32** handle, **float32** *p-Length)

  get the lowest valid wavelength [m].

- EXPORT **int32** RDL_API rb9603_monochromator_getMaxWavelength (**int32** handle, **float32** *p-Length)

  get the highest valid wavelength [m].

2.20 PIA Motorola Peripheral Interface Adapter MC6821

```
Registers       Offset  Rd/Wr   Description
------------------------------------------------------------------------
PRDA            0       rd/wr   port A peripheral data register (CRA bit 2 = 1)
DDRA            0       rd/wr   port A data direction register  (CRA bit 2 = 0)
CRA             1       rd/wr   port A control register
PRDB            2       rd/wr   port B peripheral data register (CRB bit 2 = 1)
DDRB            2       rd/wr   port B data direction register  (CRB bit 2 = 0)
CRB             3       rd/wr   port B control register
```

Each of the peripheral data lines (PA0..PA7, PB0..PB7) can be programmed to act as an input or as an output. This is accomplished by setting a "1" in the corresponding Data Direction Register bit for those lines which are to be outputs. A "0" in a bit of the Data Direction Register causes the corresponding peripheral data line to act as an input.

Peripheral Data Register:

```
76543210 : data bits 0..7, 0 for low, 1 for high output.
```

Data Direction Register:

```
76543210 : data direction bits, 0 for input, 1 for output.
```

Control Register (CRA):

```
76543210

7        : IRQA1 interrupt flag
           Goes high on active transition of CA1.
           Automatically cleared by MPU read of output register A.
           May also be cleared by hardware reset.

 6       : IRQA2 interrupt flag
           When CA2 is an input, IRQA2 goes high on active transition of CA2.
           Automatically cleared by MPU read of output register A.
           May also be cleared by hardware reset.

  543    : CA2 control
           543 : CA2 input
           5   : 0 : CA2 input
            4  : CA2 active transition for setting IRQA2 flag
                 0 : IRQA2 set by high-to-low transition on CA2
                 1 : IRQA2 set by low-to-high transition on CA2
              3 : CA2 interrupt request enable
                 0 : disable IRQA2 interrupt on active transition of CA2
                 1 : enable  IRQA2 interrupt on active transition of CA2

           543 : CA2 output
           5   : 1 : CA2 output
            43 : CA2 output control
                 0x: strobed CA2 output (operation of CB2 is not identical)
                 1x: programmed CA2 output with level of bit 3
                 00: CA2 Read Strobe with CA1 restore
                 01: CA2 Read Strobe with E   restore
                (00: CB2 Read Strobe with CB1 restore)
                (01: CB2 Read Strobe with E   restore)
                 10: CA2 low
                 11: CA2 high

     2   : data direction register access
           0 : Data Direction Register selected
```

```
          1 : Peripheral Data Register selected

      1   : CA1 active transition for setting IRQA1
          0 : IRQA1 set by high-to-low transition on CA1
          1 : IRQA1 set by low-to-high transition on CA1

       0 : CA1 interrupt enable
          0 : CA1 interrupt disabled
          1 : CA1 interrupt enabled
```

2.21 VIA Rockwell Versatile Interface Adapater R6522

```
Registers       Offset  Rd/Wr   Description
------------------------------------------------------------------------------
PDRB             0      rd/wr   Port B Peripheral Data Register
PDRA             1      rd/wr   Port A Peripheral Data Register
DDRB             2      rd/wr   Port B Data Direction Register
DDRA             3      rd/wr   Port A Data Direction Register
T1CL             4      rd/wr   Timer 1 counter LSB
T1CH             5      rd/wr   Timer 1 counter MSB
T1LL             6      rd/wr   Timer 1 latch   LSB
T1LH             7      rd/wr   Timer 1 latch   MSB
T2L              8      rd/wr   Timer 2 LSB, write latch, read counter
T2H              9      rd/wr   Timer 2 MSB, write latch, read counter
SR              10      rd/wr   Shift Register
ACR             11      rd/wr   Auxiliary Control Register
PCR             12      rd/wr   Peripheral Control Register
IFR             13      rd/wr   Interrupt Flag Register
IER             14      rd/wr   Interrupt Enable Register
PDRA            15      rd/wr   Port A Peripheral Data Register, no handshake
```

Each of the peripheral data lines (PA0..PA7, PB0..PB7) can be programmed to act as input or output. This is accomplished by setting a "1" in the corresponding Data Direction Register bit for those lines which are to be outputs. A "0" in a bit of the Data Direction Register causes the corresponding peripheral data line to act as an input.

Peripheral Data Register:

```
76543210 : data bits 0..7, 0 for low, 1 for high output.
```

Data Direction Register:

```
76543210 : data direction bits, 0 for input, 1 for output.
```

Peripheral Control Register (PCR):

```
76543210

765      : CB2 control; see CA2 control (bits 321)
   4     : CB2 interrupt control; see CA2 interrupt control (bit 0)

    321  : CA2 control
           321 : CA2 input
           3   : 0 : CA2 input
            2  : CA2 active transition for setting IRQA2 flag
                 0 : IRQA2 set by high-to-low transition on CA2
                 1 : IRQA2 set by low-to-high transition on CA2
             1 : independent interrupt
                 0 : CA1 dependent interrupt
                 1 : independent interrupt; see *), IFR

           321 : CA2 output
           3   : 1 : CA2 output
            21 : CA2 output control
                 0x: strobed CA2 output
                 1x: programmed CA2 output with level of bit 3
                 00: CA2 Handshake output
                 01: CA2 Pulse output
                 10: CA2 low
                 11: CA2 high

       0 : CA1 active transition for setting IRQA1
           0 : IRQA1 set by high-to-low transition on CA1
           1 : IRQA1 set by low-to-high transition on CA1
```

Auxiliary Control Register (ACR):

```
76543210 : timer control (not used)
```

Interrupt Flag Register (IFR):

```
76543210 : what  set                    cleared
7        : CA2   CA2 active edge        read or write PDRA *)
 6       : CA1   CA1 active edge        read or write PDRA *)
  5      : SR    complete 8 shifts      read or write SR
   4     : CB2   CB2 active edge        read or write PDRB *)
    3    : CB1   CB1 active edge        read or write PDRB
     2   : T2    time out of T2         read T2L or write T2H
      1  : T1    time out of T1         read T1L or write T1H
       0 : IRQ   any enabled interrupt  clear all interrupts

       *): if the CA2/CB2 control in the PCR is selected as "independent"
           interrupt input, then reading or writing the output register
           PDRA/PDRB will NOT clear the flag bit. Instead, the bit must
           be cleared by writing a one into the appropriate bit of the IFR.
```

Interrupt Enable Register (IER):

```
76543210
7        : set/clear (reads as "1")
           0 : disable interrupts corresponding to bits set
           1 : enable  interrupts corresponding to bits set
 6       : T1
  5      : T2
   4     : CB1
    3    : CB2
     2   : SR
      1  : CA1
       0 : CA2
```

2.22 Developer Manual

2.22.1 Detailed Description

The Rulbus Device Library contains two groups of functions:

- general library functions, Rulbus DLL Interface
- Rulbus module related functions, for example RB8510 12-bit DAC

When you want to develop a driver for a Rulbus module, please first read How to Develop a Rulbus Device Driver.

When you want to study the implementation of a specific Rulbus module, look up its reference documentation page by Rulbus number in the **Rulbus Device Class Library**.

Modules

- How to Develop a Rulbus Device Driver

  open – use – close.

- Rulbus DLL Implementation

  Rulbus DLL implementation.

2.23 How to Develop a Rulbus Device Driver

How to Develop a Rulbus Device Driver A driver for a Rulbus device consists of the following parts:

- a C-language application programming interface (API) in the Rulbus Device Library, RDL (this library)
- a C++ class for the device in the Rulbus Device Class Library (RDCL, see **there**)

The C-language API is the DLL's interface to the actual Rulbus Device objects in the DLL.

The C-language API for a Rulbus module consists of three types of functions:

- a function to *open* the device
- functions to *use* the device
- a function to *close* the device

The protypes for the programming interface functions of all Rulbus modules are located in file `rulbus.h`, whereas the implementation of these functions are placed in a separate file for each Rulbus module, named rb*yydd_name*.cpp, like `rb8510_dac12.cpp`.

The following sections show several API-functions for the 12-bit DAC module, RB8510.

open

```
extern "C" EXPORT int32 RDL_API rb8510_dac12_open( int32 *pHandle, CCstr name )
{
   return RulbusDevice_open( pHandle, name );
}
```

use - obtain a value

```
extern "C" EXPORT int32 RDL_API rb8510_dac12_getVoltage( int32 handle, float32* pVoltage )
{
   try         { *pVoltage = to_rb8510( TheRulbusDeviceList::instance().get( handle ) ).voltage(); }
   catch (...) { return 1; }

   return 0;
}
```

use - set a value

```
extern "C" EXPORT int32 RDL_API rb8510_dac12_setVoltage( int32 handle, float32 voltage )
{
   try         { to_rb8510( TheRulbusDeviceList::instance().get( handle ) ).setVoltage( voltage ); }
   catch (...) { return 1; }

   return 0;
}
```

close

```
extern "C" EXPORT int32 RDL_API rb8510_dac12_close( int32 handle )
{
   return RulbusDevice_close( handle );
}
```

There are several things worth to mention here.

A Rulbus device is accessed via a handle, a number. This handle is obtained by the call to open and it is the link to the actual Rulbus device object. The handle–device object relation is maintained by `TheRulbusDeviceList`, of which there is and can be only one in the DLL (hence the `The`).

Further, the C++ style error handling through exceptions of the Rulbus Device Class Library is tranformed here to an error return value. This is the usual mechanism to report errors in for example LabVIEW. More information on an error can be obtained with rdl_getLastError().

It is also interesting to note the dual purpose of **to_rb8510()**:

- enable the protection of the Rulbus device (see class **RulbusDeviceProxy** in the RDCL)
- change the more general type RulbusDevice collected in **TheRulbusDeviceList** to the specific type **RB8510_Dac12**

2.24 Rulbus DLL Implementation

2.24.1 Detailed Description

Functions

- int __rdl_initialize ()

  initialize without exception handling; used by rdl_initialize(), DllMain().

- int __rdl_finalize ()

  finalize without exception handling; used by rdl_finalize(), DllMain(); do nothing: another process may be using it (do not close global semaphores!); maybe use shared data in the future (see CreateFileMapping()).

Chapter 3

Rulbus Device Library for Microsoft Windows Directory Documentation

3.1 H:/myprojects/bf/prj/rulbus-rdl/librdl/src/ Directory Reference

Files

- file **rb8506_pport.cpp**
- file **rb8506_sifu.cpp**
- file **rb8509_adc12.cpp**
- file **rb8510_dac12.cpp**
- file **rb8513_timebase.cpp**
- file **rb8514_delay.cpp**
- file **rb8515_clock.cpp**
- file **rb8905_adc12.cpp**
- file **rb9005_amplifier.cpp**
- file **rb9603_monochromator.cpp**
- file **rulbus-types.h**
- file **Rulbus.cpp**
- file **rulbus.h**
- file **sources.inc**
- file **version.h**

Chapter 4

Rulbus Device Library for Microsoft Windows Example Documentation

4.1 dac.cpp

This is an example that shows how the RB8510 12-bit DAC may be used.

```
/*
 * dac.cpp - generate a staircase voltage.
 *
 * compile: bcc32 dac.cpp rulbus.lib
 */

#include <stdio.h>      // for printf() etc.
#include <stdlib.h>     // for strtol()
#include <windows.h>    // for Sleep()
#include "rulbus.h"     // rulbus interface

static int usage();     // print program usage, return EXIT_FAILURE
static int error();     // print error, return EXIT_FAILURE

/*
 * main - handle commandline arguments and generate staircase voltage on DAC.
 */

int main( int argc, char *argv[] )
{
   /*
    * handle commandline arguments:
    */

   if ( argc < 2 )
      return usage();

   /*
    * the name on the commandline must correspond to the name of a 12-bit
    * DAC in the Rulbus device configuration file, typically rulbus.conf.
    */

   char *name = argv[1];

   /*
    * open the DAC:
    */
```

```
   int32 handle;
   if ( rb8510_dac12_open( &handle, name ) )
      return error();

   /*
    * generate 11 one volt steps, one per second:
    *
    * Note that the last step generates a RulbusRangeError, because the
    * voltage is outside [-10.235 .. +10.24 V].
    */

   for ( int i = 0; i <= 11; i++ )
   {
      fprintf( stdout, "[%d]", i );

      if ( rb8510_dac12_setVoltage( handle, i ) )
         return error();

      Sleep( 1000 );                    // delay one second
   }

   /*
    * close the DAC:
    */

   if ( rb8510_dac12_close( handle ) )
      return error();

   return EXIT_SUCCESS;
}

/*
 * usage - print program usage.
 */

static int usage()
{
   fprintf( stdout, "Usage: dac device-name\n" );
   return EXIT_FAILURE;
}

/*
 * error - retrieve and print Rulbus error.
 */

static int error()
{
   const int len = 100; char msg[len];

   rdl_getLastError( msg, len );
   fprintf( stdout, "%s\n", msg );

   return EXIT_FAILURE;
}
```

4.2 error.cpp

This is an example of how you can use function rdl_getLastError().

```
/*
 * error - retrieve and print Rulbus error.
 */

int error()
{
   const int len = 100; char  msg[len];

   rdl_getLastError( msg, len );
   fprintf( stdout, "%s\n", msg );

   return 1;
}
```

4.3 pattern.cpp

This example shows how you may write a data pattern to the Rulbus, using the general interface functions.

```
/*
 * pattern - write a data pattern to the Rulbus.
 */

#include "rulbus.h"             // header
#include <stdio.h>              // for fprintf()
#include <stdlib.h>             // for EXIT_SUCCESS
#include <conio.h>              // for kbhit()

int error() { return EXIT_FAILURE; }

int main()
{
   int32 pattern = 0x5E;        // test pattern
   int32 rack    = 0;           // rulbus rack number
   int32 addr    = 0;           // rulbus base address
   int32 offset  = 0x12;        // rulbus address offset
   int32 handle  = 0;           // handle to generic rulbus device
                                // open the generic rulbus device
   if ( RulbusDevice_open( &handle, "Rulbus-test-device" ) )
      return error();

   RulbusDevice_getRack   ( handle, &rack );
   RulbusDevice_getAddress( handle, &addr );

   fprintf( stdout, "Writing [%d:0x%02X] <- 0x%02X\n", rack, addr + offset, pattern );
   fprintf( stderr, "\nPress a key to stop..." );

   while( !kbhit() )            // write pattern until a key is pressed
      if ( 0 > RulbusDevice_putByte( handle, offset, pattern ) );
         return error();

   (void) getch();              // eat character

   if ( RulbusDevice_close( handle ) )
      return error();           // close the generic rulbus device

   return EXIT_SUCCESS;
}
```

4.4 pport.cpp

This is an example that shows how the RB8506 dual parallel interface (PIA/VIA) may be used.

```
/*
 * pport.cpp - switch a LED on and off, a parallel interface demonstration.
 *
 * The switch is connected to PB3 of parallel interface, Pia 2,
 * the LED is connected to PB0.
 *
 * The program continuously reads the switch and puts the LED on
 * when the switch is closed and puts the LED off when the switch
 * is open. Note the inverted logic of the switch (pull-up) as
 * well as of the LED (pulled down).
 *
 *  +5V           parallel interface           +5V
 *   +             +------------+              +
 *   |             |    pia2    |   \\         |
 *   +---===---+---| PB3    PB0 |---|<|---===---+
 *        /    |   |            |
 *   +---/  ---+   +------------+
 *   | switch  LED
 *  ---
 *
 * The program uses two methods:
 * 1. mask: read and set the i/o-pins by reading and writing port B
 * 2. line: read and set the i/o-pins by reading and wrting a single line
 *
 * The Rulbus configuration file must define the port as "pport" at address 0x94:
 *
 * rack "a rack" {
 *    rb8506_pia "pport" {
 *       address = 0x94;
 *    }
 * }
 *
 * Compile: bcc32 par.cpp rulbus.lib
 */

#include "rulbus.h"
#include <stdio.h>      // for printf()
#include <conio.h>      // for kbhit(), getch()
#include <windows.h>    // for Sleep()

enum  Pin { pin0, pin1, pin2, pin3, pin4, pin5, pin6, pin7, };

const char* portName   = "pport"; // the name in rulbus.conf

const Pin   pinLED     = pin0;    // the LED is at PB0
const Pin   pinSwitch  = pin3;    // the switch is at PB4

const char* portLED    = "PB";    // for port functions
const char* portSwitch = "PB";

const char* lineLED    = "PB0";   // for line functions
const char* lineSwitch = "PB3";

const int   rack       = 0;
const int   address    = 0x94;    // this is the parallel interface, Pia 2

inline int  mask( int pin ) {  return 1 << pin; }

inline int  tstbit( int32  byte, int pin ) {  return ( byte &   mask(pin) ) != 0; }
inline int  clrbit( int32& byte, int pin ) {  return   byte &= ~mask(pin); }
inline int  setbit( int32& byte, int pin ) {  return   byte |=  mask(pin); }
inline int  xorbit( int32& byte, int pin ) {  return   byte ^=  mask(pin); }
```

```
static void setPortDirection( int handle );
static void setLineDirection( int handle );
static void switchLED_mask  ( int handle );
static void switchLED_line  ( int handle );

/*
 * the program
 */

int main()
{
   printf( "Switch a LED on and off, a parallel interface demonstration." );

   int32 handle;
   rb8506_pport_open( &handle, portName );

   // using port functions:

   printf( "\nsetting port direction..." );
   setPortDirection( handle );

   printf( "\nusing port data with bitmasks; press a key to contine..." );
   switchLED_mask( handle );

   // using line functions:

   printf( "\nsetting line direction..." );
   setLineDirection( handle );

   printf( "\nusing single line functions; press a key to contine..." );
   switchLED_line( handle );

   printf( "\n" );

   return rb8506_pport_close( handle );
}

//--- solution using port-i/o with bitmasks

/*
 * Set data direction, using bitmasks.
 */

static void setPortDirection(int handle)
{
   int32 dataSwitch, dataLED;
   rb8506_pport_getPortDir( handle, portSwitch, &dataSwitch );
   rb8506_pport_getPortDir( handle, portLED   , &dataLED    );

   rb8506_pport_setPortDir( handle, portSwitch, dataSwitch & ~mask(pinSwitch) );
   rb8506_pport_setPortDir( handle, portLED   , dataLED    |  mask(pinLED   ) );
}

/*
 * Read the switch and control the LED using bit-masks.
 */

static void switchLED_mask(int handle)
{
   while ( !kbhit() )
   {
      int32 dataSwitch, dataLED;
      rb8506_pport_getPortData( handle, portSwitch, &dataSwitch );
      rb8506_pport_getPortData( handle, portLED   , &dataLED    );

      if ( tstbit( dataSwitch, pinSwitch ) ) setbit( dataLED, pinLED );
```

```
      else                                  clrbit( dataLED, pinLED );

      rb8506_pport_setPortData( handle, portLED, dataLED );

      Sleep( 10 );
   }

   (void) getch();
}

//--- solution using line-i/o

/*
 * Set data direction, using line-functions.
 */

static void setLineDirection(int handle)
{
   rb8506_pport_setLineDir( handle, lineSwitch, 'i' );
   rb8506_pport_setLineDir( handle, lineLED   , 'o' );
}

/*
 * Read the switch and control the LED using single line i/o-functions.
 */

static void switchLED_line(int handle)
{
   while ( !kbhit() )
   {
      int32 switchOpen;

      rb8506_pport_getLineLevel( handle, lineSwitch, &switchOpen );
      rb8506_pport_setLineLevel( handle, lineLED   ,  switchOpen );

      Sleep( 10 );
   }

   (void) getch();
}
```

4.5 rulbus.conf

This example shows what a Rulbus configuration file may look like.

```
/*
 * rulbus.conf - Rulbus rack and cards configuration file.
 *
 * This file is part of the Rulbus Device Class Library (RDCL).
 *
 * Copyright (C) 2003-2004, Leiden University.
 *
 * This library is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * The library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with mngdriver; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * $Id: rulbus.conf 19 2005-04-06 06:26:31Z moene $
 */

/*
 * //
 * // small example rulbus.conf:
 * //
 *
 * rack "toprack"
 * {
 *    address = 7                                // this rack has address 7
 *    rb8509_adc12 "adc"                         // uses defaults
 *    rb8509_dac12 "dac-ch0"                     // uses defaults
 *    rb8509_dac12 "dac-ch1" { address = 0xD2 } // address specified
 * }
 *
 * rack "bottomrack"
 * {
 *    address = 8                                // this rack has address 8
 *    rb8515_clock "clock"   { }                 // uses defaults
 *    rb8514_delay "delay0"  { }                 // uses defaults
 *    rb8514_delay "delay1"  { address = 0xA4 } // address specified
 * }
 */

/*
 * keywords (case sensitive):
 *
 * - rack
 * - rb_generic
 * - rb8506_pia
 * - rb8506_sifu
 * - rb8506_via
 * - rb8509_adc12
 * - rb8510_dac12
 * - rb8513_timebase
 * - rb8514_delay
 * - rb8515_clock
 * - rb8905_adc12
 * - rb9005_amplifier
 * - rb9603_monochromator
```

```
 *
 * - address          (all cards)
 * - bipolar          (rb8509_adc12, rb8510_dac12)
 * - has_ext_trigger  (rb8509_adc12)
 * - intr_delay       (rb8514_delay)
 * - num_channels     (rb8509_adc12)
 * - volt_per_bit     (rb8509_adc12, rb8510_dac12)
 *
 * - false, no
 * - true, yes
 *
 * - [fpnumkMGT] characters used as suffix on a number (unit) have the
 *   following meaning
 *
 *   sfx | meaning
 *   ----+-------------
 *    f  | femto  1e-15
 *    p  | femto  1e-12
 *    n  | pico   1e-9
 *    u  | micro  1e-6
 *    m  | milli  1e-3
 *    k  | kilo   1e+3
 *    M  | mega   1e+6
 *    G  | giga   1e+9
 *    T  | tera   1e+12
 */

/*
 * Rulbus rack address
 *
 * The address of a Rulbus rack can be set via a 4-bit DIP-switch on its
 * rear-side.
 *
 *
 *    Secondary Address                    Secondary Address
 *             1  0                                 1  0
 *   weight +------+  bit                 weight +------+  bit
 *        1 | .  o | DS 0                      1 | o  . | DS 0
 *        2 | .  o | DS 1                      2 | o  . | DS 1
 *        4 | .  o | DS 2                      4 | o  . | DS 2
 *        8 | .  o | DS 3                      8 | o  . | DS 3
 *          +------+                            +------+
 *
 *  0: 0*8 + 0*4 + 0*2 + 0*1           15: 1*8 + 1*4 + 1*2 + 1*1
 *
 *
 * So there are 16 addresses to select from [0..0xF].
 *
 * Address 15 (0xF) is a special address: when a Rulbus rack has its
 * DIP-switch set to 15, it is always selected, no matter what the
 * computer tells the rack-selector.
 *
 * Please specify the DIP-switch setting for the address in a rack's
 * declaration.
 *
 * When you do not specify a rack address it will be 0xF.
 */

rack "top" =                    // this is a line comment, the '//' starts it
{
   address = 3                  // the required rack address [0..0xF]

   rb8506_pia "pport"           // the simplest card definition, using default properties
   {
      address = 0x94;
   }
```

```
   rb8509_adc12 "adc12" =       // the most verbose card definition ( with '=' and ';' )
   {                            // the initializer
      address = 0xC0;           // the card address property

      has_ext_trigger = false   // or: no; no external trigger input on front
      num_channels    = 8;      // or: 4 on some modules
      bipolar         = true;   // or: yes
      volt_per_bit    = 5e-3;   // or: 5m, or 5 m
   };                           // end of initializer

   rb8510_dac12 "dac12-ch0"     // the simplest card definition with initializer
   {
   }

   rb8510_dac12 "dac12-ch1" =
   {
      address      = 0xD2
      bipolar      = true
      volt_per_bit = 5 m
   }

   rb8514_delay "delay0" =
   {
      /* address = 0xA0 */
   }

   rb8514_delay "delay1" =
   {
      address    = 0xA4
      intr_delay = 50 n         // 50 ns intrinsic delay
   }

   rb8515_clock "clock1";       // using defaults
}

/*
 * end of file
 */
```

4.6 run-daccs.cmd

client-server thread example

The following example shows that the same DAC channel, named "dac-ch0", can be shared among two threads of the same process.

Please read the output the test produces.

```
Run-daccs - setup environment to use .\rulbus.conf, run daccs.

Using RULBUS=isa
___

DacCS  1.0 (22 Jan 2004)  Use Rulbus DAC from a client and a server thread.

This program creates a server thread and a client thread.

Server: generate a squarewave on DAC channel 0 [data 1000,3000].
Client: read the data from DAC channel 0 and check its values;
        also copy data read to DAC channel 1.

The server thread is intentionally started somewhat later than
the client thread to show an error report from the latter.

ISA Rulbus Interface at [0x200], using CanIO port I/O

opened Rulbus device 'dac-ch0' at [0:0xD0]
opened Rulbus device 'dac-ch1' at [0:0xD2]

creating thread for server
creating thread for client

Press a key to stop...

client: reading data from   Rulbus device 'dac-ch0' at [0:0xD0]
client: copying data to     Rulbus device 'dac-ch1' at [0:0xD2]
client: dac handle 0 data 1234 not in [1000,3000]
client: dac handle 0 data 1234 not in [1000,3000]
server: write squarewave on Rulbus device 'dac-ch0' at [0:0xD0]

Terminating...
prompt>
```

Here is the Rulbus configuration file used.

```
# dac client--server test

rack "top"
{
   address = 0

   rb8510_dac12 "dac-ch0" { address = 0xD0 }
   rb8510_dac12 "dac-ch1" { address = 0xD2 }
}
```

The C++ source of the program.

```
/*
 * daccs.cpp - dac client-server.
 *
 * compile: bcc32 -P -q -tWC -tWR -tWM -I..\src .\daccs.cpp ..\bin\rulbus.lib
```

```
 */

#include <iostream>                    // for std::cout, std::cerr
#include <conio.h>                     // for kbhit()
#include <windows.h>                   // for DWORD HANDLE, Sleep()

#include "rulbus.h"                    // for rb8510_dac12_open() etc.

namespace ClientServer
{
   const char *title = "DacCS  1.0 (22 Jan 2004)  Use Rulbus DAC from a client and a server thread.\n";

   int           threads ( );
   HANDLE        mkThread( const char *msg, DWORD WINAPI (*ThreadFunc)(LPVOID), int32 *pHandle );
   DWORD  WINAPI client  ( LPVOID arg );
   DWORD  WINAPI server  ( LPVOID arg );
   int           error   (            );

   volatile bool runthread = true;      // true while threads should run

   const   int   NoCode   = 1234;       // DAC 'no' code
   const   int   LoCode   = 1000;       // DAC lower code
   const   int   HiCode   = 3000;       // DAC higher code

   /*
    * Rulbus address tuple for channel 0 and 1
    */

   struct Tuple
   {
      int32 dac0;
      int32 dac1;
   };

   #pragma argsused
   int main( int argc, char *argv[] )
   {
      std::cout << title << std::endl;

      return threads();
   }

   int threads()
   {
      std::cout <<
         "This program creates a server thread and a client thread.\n"
         "\n"
         "Server: generate a squarewave on DAC channel 0 [data " << LoCode << ',' << HiCode << "].\n"
         "Client: read the data from DAC channel 0 and check its values;\n"
         "        also copy data read to DAC channel 1.\n"
         "\n"
         "The server thread is intentionally started somewhat later than\n"
         "the client thread to show an error report from the latter.\n" << std::endl;

      /*
       * report on Rulbus Interface:
       */

      rdl_printRulbusInterface();

      /*
       * open two DAC channels:
       */

      Tuple tuple;

      if ( rb8510_dac12_open( &tuple.dac0, "dac-ch0" ) ||
```

```
        rb8510_dac12_open( &tuple.dac1, "dac-ch1" )   )
    {
       return error();
    }

    std::cerr << "opened "; RulbusDevice_print( tuple.dac0 );
    std::cerr << "opened "; RulbusDevice_print( tuple.dac1 );
    std::cerr << std::endl;

    /*
     * initialize DAC outputs to be different from server output:
     */

    rb8510_dac12_setValue( tuple.dac0, NoCode );
    rb8510_dac12_setValue( tuple.dac1, NoCode );

    /*
     * create and resume client and server threads;
     * intentionally resume the server thread somewhat later
     * to notice the error report from the client thread.
     */

    HANDLE thread0 = mkThread( "server", server, reinterpret_cast<LONG *>( &tuple ) );
    HANDLE thread1 = mkThread( "client", client, reinterpret_cast<LONG *>( &tuple ) );

    std::cerr << "\nPress a key to stop...\n" << std::endl;

    ResumeThread( thread1 ); Sleep( 2000 );
    ResumeThread( thread0 );

    /*
     * wait for key pressed; eat character:
     */

    while ( !kbhit() )
      Sleep( 10 );

    (void) getch();

    /*
     *  stop and remove threads and close DACs:
     */

    std::cerr << "\nTerminating..." << std::endl;

    runthread = 0; Sleep( 10 );

    CloseHandle( thread0 );
    CloseHandle( thread1 );

    if ( rb8510_dac12_close( tuple.dac1 ) ||
         rb8510_dac12_close( tuple.dac0 )   )
    {
       return error();
    }

    return EXIT_SUCCESS;
}

HANDLE mkThread( const char *msg, DWORD WINAPI (*ThreadFunc)(LPVOID), int32 *pTuple )
{
    std::cerr << "creating thread for " << msg << std::endl;

    DWORD id;

    return CreateThread(
       NULL,             // pointer to thread security attributes
```

```
      0,                // initial thread stack size, in bytes
      ThreadFunc,       // pointer to thread function
      pTuple,           // argument for new thread
      CREATE_SUSPENDED, // creation flags
      &id               // pointer to returned thread identifier
   );
}

DWORD WINAPI server( LPVOID arg )
{
   int32 dac0 = reinterpret_cast<Tuple *>(arg)->dac0;

   std::cerr << "server: write squarewave on "; RulbusDevice_print( dac0 );

   while ( runthread )
   {
      rb8510_dac12_setValue( dac0, LoCode ); Sleep( 0 );
      rb8510_dac12_setValue( dac0, HiCode ); Sleep( 0 );
   }

   return 0;
}

DWORD WINAPI client( LPVOID arg )
{
   int32 dac0 = reinterpret_cast<Tuple *>(arg)->dac0;
   int32 dac1 = reinterpret_cast<Tuple *>(arg)->dac1;

   std::cerr << "client: reading data from   "; RulbusDevice_print( dac0 );
   std::cerr << "client: copying data to     "; RulbusDevice_print( dac1 );

   while ( runthread )
   {
      int32 n;

      if ( rb8510_dac12_getValue( dac0, &n ) ||
           rb8510_dac12_setValue( dac1,  n )    )
      {
         return error();
      }

      if ( n != LoCode && n!= HiCode )
      {
         std::cerr << "client: dac handle " << dac0 << " data " << n << " not in [" << LoCode << ',' <<
         Sleep  ( 1000 );
      }
      else
      {
         // do nothing
         // std::cerr << n << " "; Sleep(300);
      }

      Sleep(0);
   }

   return 0;
}

int error()
{
   const int len = 100; char  msg[len];

   rdl_getLastError( msg, len );
   std::cerr << msg << std::endl;

   return EXIT_FAILURE;
}
```

```
} // namespace ClientServer

/*
 * End of file
 */
```

This is the batch command file used to invoke the program.

```
@echo off
rem
rem run-daccs.bat - setup environment to use .\run-daccs.conf, run daccs.
rem

cls
echo Run-daccs - setup environment to use .\run-daccs.conf, run daccs.
echo.

set RULBUS.ORG=%RULBUS%
set RULBUS_CONFIG_FILE.ORG=%RULBUS_CONFIG_FILE%

:set RULBUS=epp,0x378;nocheck
set RULBUS=isa

echo Using RULBUS=%RULBUS%
echo ___
echo.

set RULBUS_CONFIG_FILE=run-daccs.conf

copy ..\bin\rulbus.dll . >nul

daccs

rem
rem restore environment:
rem

set RULBUS=%RULBUS.ORG%
set RULBUS.ORG=

set RULBUS_CONFIG_FILE=%RULBUS_CONFIG_FILE.ORG%
set RULBUS_CONFIG_FILE.ORG=

rem
rem end of file
rem
```

4.7 run-daccs2.cmd

client-server process example

The following example shows that the same DAC channel, named "dac-ch0", cannot be shared among two processes: the client process cannot see the changes that the server process makes to the DAC channel.

Please read the output the test produces.

Server process output.

```
DacServer  1.0 (22 Jan 2004)  Use Rulbus DAC from client and server processes.

This program generates a squarewave on DAC channel 0 [voltage -5,5 V].

ISA Rulbus Interface at [0x200], using CanIO port I/O

opened Rulbus device 'dac-ch0' at [0:0xD0]

[5][-5][5][-5]
```

Client process output.

```
DacClient  1.0 (22 Jan 2004)  Use Rulbus DAC from client and server processes.

This program reads the voltage from DAC channel 0 [expecting voltage -5,5 V],
and copies the voltage to DAC channel 1.

ISA Rulbus Interface at [0x200], using CanIO port I/O

opened Rulbus device 'dac-ch0' at [0:0xD0]
opened Rulbus device 'dac-ch1' at [0:0xD2]

[0][0][0][0][0][0]
```

Here is the Rulbus configuration file used.

```
# dac client--server test

rack "top"
{
   address = 0

   rb8510_dac12 "dac-ch0" { address = 0xD0 }
   rb8510_dac12 "dac-ch1" { address = 0xD2 }
}
```

The C++ source of the server program.

```
/*
 * dacserver.cpp - generate alternating -5V, +5V on DAC channel 0.
 */

#include "rulbus.h"
#include <iostream>

/*
 * the Server namespace
 */
```

```
namespace Server
{
   const char *title = "DacServer  1.0 (22 Jan 2004)  Use Rulbus DAC from client and server processes.\n";

   const double LoVoltage = -5.0;       // DAC lower voltage
   const double HiVoltage = +5.0;       // DAC higher voltage

   int   server( int handle );
   int   error ();

   /*
    * the program
    */

   int main()
   {
      std::cout <<
         title <<
         "\n"
         "This program generates a squarewave on DAC channel 0 [voltage " <<
           LoVoltage << ',' << HiVoltage << " V].\n" << std::endl;

      /*
       * report on Rulbus Interface:
       */

      rdl_printRulbusInterface();

      /*
       * open DAC channel 0:
       */

      int32   dac0;
      float32 vpb0;

      if ( rb8510_dac12_open( &dac0, "dac-ch0"     ) ||
           rb8510_dac12_getVoltperbit( dac0, &vpb0 )   )
      {
         return error();
      }

      std::cerr << "opened "; RulbusDevice_print( dac0 );
      std::cerr << std::endl;

      return server( dac0 );
   }

   /*
    * server -
    */

   int server( int handle )
   {
      while( true )
      {
         float32 v;

         Sleep( 2000 );

         if ( rb8510_dac12_setVoltage( handle, +5.0 ) ||
              rb8510_dac12_getVoltage( handle,  &v  )    )
              return error();

         std::cerr << "[" << v << "]";

         Sleep( 2000 );
```

```
         if ( rb8510_dac12_setVoltage( handle, -5.0 ) ||
              rb8510_dac12_getVoltage( handle,  &v  )    )
              return error();

         std::cerr << "[" << v << "]";
      }

      //return EXIT_SUCCESS;
   }

   /*
    * error - retrieve and print Rulbus error.
    */

   int error()
   {
      const int len = 100; char  msg[len];

      rdl_getLastError( msg, len );
      std::cerr << msg << std::endl;

      return EXIT_FAILURE;
   }

} // namespace Server

/*
 * end of file
 */
```

The C++ source of the client program.

```
/*
 * dacclient.cpp - read voltage from DAC channel 0, copy it to DAC channel 1.
 */

#include "rulbus.h"
#include <iostream>

/*
 * the Client namespace
 */

namespace Client
{
   const char *title = "DacClient  1.0 (22 Jan 2004)  Use Rulbus DAC from client and server processes.\n";

   const double LoVoltage = -5.0;       // DAC lower voltage
   const double HiVoltage = +5.0;       // DAC higher voltage

   int   client( int handle0, int handle1 );
   int   error ();

   /*
    * the program
    */

   int main()
   {
      std::cout <<
         title <<
         "\n"
         "This program reads the voltage from DAC channel 0 [expecting voltage " <<
           LoVoltage << ',' << HiVoltage << " V],\n"
         "and copies the voltage to DAC channel 1.\n" << std::endl;
```

```
      /*
       * report on Rulbus Interface:
       */

      rdl_printRulbusInterface();

      /*
       * open DAC channel 0:
       */

      int32 dac0, dac1;
      float32 vpb0;

      if ( rb8510_dac12_open( &dac0, "dac-ch0" ) ||
           rb8510_dac12_open( &dac1, "dac-ch1" ) ||
           rb8510_dac12_getVoltperbit( dac0, &vpb0 ) )
      {
           return error();
      }

      std::cerr << "opened "; RulbusDevice_print( dac0 );
      std::cerr << "opened "; RulbusDevice_print( dac1 );
      std::cerr << std::endl;

      return client( dac0, dac1 );
   }

   /*
    * client - read DAC channel0 and copy to DAC channel 1
    */

   int client( int handle0, int handle1 )
   {
      while( true )
      {
         float32 v;

         if ( rb8510_dac12_getVoltage( handle0, &v ) ||
              rb8510_dac12_setVoltage( handle1,  v )    )
         {
              return error();
         }

         std::cerr << "[" << v << "]"; Sleep( 750 );
      }

      //return EXIT_SUCCESS;
   }

   /*
    * error - retrieve and print Rulbus error.
    */

   int error()
   {
      const int len = 100; char  msg[len];

      rdl_getLastError( msg, len );
      std::cerr << msg << std::endl;

      return EXIT_FAILURE;
   }

} // namespace Client

/*
```

```
 * end of file
 */
```

This is the batch command file used to invoke the client and server programs.

```
@echo off
rem
rem run-daccs2.bat - setup environment to use .\run-daccs.conf, run daccs.
rem

cls
echo Run-daccs2 - setup environment to use .\run-daccs.conf, run dacclient and -server.
echo.

set RULBUS.ORG=%RULBUS%
set RULBUS_CONFIG_FILE.ORG=%RULBUS_CONFIG_FILE%

:set RULBUS=epp,0x378;nocheck
set RULBUS=isa

echo Using RULBUS=%RULBUS%
echo ___
echo.

set RULBUS_CONFIG_FILE=run-daccs.conf

copy ..\bin\rulbus.dll . >nul

start dacclient
start dacserver

rem
rem restore environment:
rem

set RULBUS=%RULBUS.ORG%
set RULBUS.ORG=

set RULBUS_CONFIG_FILE=%RULBUS_CONFIG_FILE.ORG%
set RULBUS_CONFIG_FILE.ORG=

rem
rem end of file
rem
```

Chapter 5

Rulbus Device Library for Microsoft Windows Page Documentation

5.1 Acknowledgements

It was Henk Klein Wolterink's vision in 1985 to design the Rulbus architecture and bridge the worlds of different microprocessors that existed at the various electronic departments of Leiden University (then Rijksuniversiteit Leiden) and at the same time bring the departments closer to each other by means of the meetings held to achieve the goal. That Rulbus is still actively used is a clear indicator of the quality of his vision. So is the fact that the University now has a united electronic department. Many thanks. In 1999 Henk Klein Wolterink established Applied Instruments [APPLINSTR].

Jens Thoms Törring designed a Rulbus configuration file for the Rulbus support in his fsc2 [FSC2] spectrometer application. Thanks for his idea and support to create a common format for it after I copied and altered it for the RDL.

Thanks to Joris Slob who reported 'strange behaviour' of various modules that turned out to be real bugs.

5.2 References

URL catalog

[APPLINSTR] Henk Klein Wolterink's Applied Instruments.

http://www.applied-instruments.com/

[BCBDEVDLL] Harold Howe's Creating DLLs in BCB that can be used from Visual C++.

http://www.bcbdev.com/articles/bcbdll.htm

[FSC2] Jens Thoms Törring's software package for remote control of spectrometers.

http://www.physik.fu-berlin.de/~toerring/fsc2.phtml

[IEEE1284] IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers (1284-2000).

http://www.ieee.org/

[MSDLL] Microsoft. Dynamic-Link Libraries.

http://msdn.microsoft.com/library/en-us/dllproc/base/dynamic_-link_libraries.asp

[RULBUS] Leiden University. Rulbus specifications and modules.

http://www.eld.LeidenUniv.nl/~moene/rulbus/

Book references

[Gamma et al., 1995] Erick Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns; Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley. ISBN 0-201-63361-2.

[Kernighan & Pike, 1999] Brian W. Kernighan and Rob Pike. 1999. *The Practice of Programming.* Reading, Massachusetts: Addison-Wesley. ISBN 0-201-61586-X.

[Stroustrup, 2000] Bjarne Stroustrup. 2000. *The C++ Programming Language: Special Edition, 3/E.* Boston: Addison Wesley Professional. ISBN 0-201-70073-5.

5.3 Todo List

Group rdl_library_compilers find out how to use rulbus.dll with LabWindows/CVI

Index