[2]p()*1/2-^ [1]p1-^ TJ
310000pt

CanModule Developer Documentation

Release 1.1.1

Michael Ludwig

May 21, 2019

Contents

CHAPTER 1

Downloading

1.1 The sources

CanModule can be cloned from github, and are integrated in a cmake build chain:

the latest stable release (preferred): git clone **-b latestStable** https://github.com/quasar-team/CanModule.git

or

a specific version: git clone **-b 1.1.0** https://github.com/quasar-team/CanModule.git

1.2 The binaries

(bin/exe) and libraries (so/dll) for the target platforms

- CERN CC7
- windows 2016 server
- windows 10
- and windows 2008r2 (server) LEGACY

are available at NexusCan from inside CERN.

CHAPTER 2

Objective

CanModule is a software abstraction layer, written in C++, to simplify integration of CAN bridges from different vendors into cmake (C++) projects needing CAN connectivity for windows and linux. A CAN bridge is - usually - an external module which is connected on one side to a computer or network and offers CAN ports on the other side where CAN buses can be connected.

The original authors are the CERN Atlas-DCS team, support is now done by BE-ICS-FD.

2.1 Supported OS

These operating systems are supported for all vendors

- CERN CC7
- Windows 2016 Server and Windows 10

other OS versions might be available on special request, or easily be ported to.

2.2 compatible vendors

CAN bridges from vendors are compatible with CanModule and are tested

- AnaGate
- SysTec
- Peak

These vendors produce many types of CAN bridges, **CanModule supports presently the USB (Peak, Systec) and ethernet (Anagate) bridges**. Flexible datarate bridges (PEAK FD) and other interface types (PEAK .m2 or also PCI) might be added if needed.

2.3 Integration into projects

In order to use CanModule in your project/server/code there are two ways

- by source-clone from CanModuleGitlab and integrated into a cmake build chain.
- binaries to copy for other build chains are available CERN-wide from NexusCan .

CHAPTER 3

Standard API

The user API is using **3 classes and hides all vendor specific details**. Only a few common methods are needed:

- A **vendor is chosen** and specific details are then taken care of by loading any further dependencies:

class CanLibLoader
: Subclassed by CanModule::CanLibLoaderLin, CanModule::CanLibLoaderWin

- Connection/access details to the **different vendors** are managed by sub-classing:

class CCanAccess
: Subclassed by *AnaCanScan*, *CSockCanScan*, MockCanAccess, *PKCanScan*, *STCanScan*

- The **access to a CAN port** is through:

class CanBusAccess
: *CanBusAccess* class ensure a connection to can hardware. it can create the connection to different hardware at same time the syntax of the name is "name of component:name of the channel"

- This **snippet** gives an overview how the API is used to hide vendor details and achieve CAN connectivity:

```
string libName = "sock";          // here: systec or peak through socketCan, linux
string port = "sock:can0";        // here: CAN port 0 via socket CAN, linux
string parameters = "Undefined"; // here: use defaults
CanMessage cm;
CanModule::CanLibLoader *libloader = CanModule::CanLibLoader::createInstance( libName␣
↪);
cca = libloader->openCanBus( port, parameters );
cca->sendMessage( &cm );
cca->canMessageCame.connect( &onMessageRcvd ); // connect a reception handler
```

- Only two strings, "port" and "parameters"

have to defined to communicate with a CAN port for a module from a vendor.

- a connection handler method

must be registered to treat received messages (boost slot connected to boost signal):

```
connection.h :
class CONNECTION {
   (...)
   public:
      static void onMessageRcvd(const CanMsgStruct/*&*/ message);
}
```

```
connection.cpp :
/* static */ void CONNECTION::onMessageRcvd(const CanMsgStruct/*&*/ message){
   MYCLASS *myObject = MYCLASS::getMyObject( ... );
   myObject->processReceivedMessage( message );
}
```

- you can take a look at CANX for a full multithreaded example using CanModule (CERN, gitlab)

CHAPTER 4

C-Wrapper

in order to call **CanModule** also from low-level vendor specific code, which is often implemented in C, CanModule has a C-wrapper as well. The objective is to call the C++ objects of CanModule from C-code. Still, a C++ compiler is needed for the build, but this type of compiler is usually downward compatible to C.

CHAPTER 5

SysTec

All modules from vendor *SysTec* are handled by class STCanScan (windows) or CSockCanScan (linux) which manage the modules through their underlying vendor specific API and provide the standard generic CanModule API. Here the underlying vendor specific classes and the specific parameters are documented.

SysTec modules USB-CAN bridges are supported: sysWORXX 1,2,8,16

5.1 The connection

To connect to a specific port for I/O, and send CAN messages, the following methods are used.

5.1.1 windows

The connection to a specific port for I/O is created by calling

class STCanScan : **public** CanModule::*CCanAccess*

Public Functions

bool **createBus**(**const** string *name*, **const** string *parameters*)

Method that initialises a CAN bus channel for systec. All following methods called on the same object will be using this initialized channel.

Return was the initialisation process successful?

Parameters

- `name`: = 2 parameters separated by ":" like "n0:n1"
  - n0 = "st" for systec
  - n1 = CAN port number on the module, can be prefixed with "can"

  - ex.: “st:can1” speaks to port 1 on systec module at the ip
  - ex.: “st:1” works as well
- `parameters`: one parameter: “p0”, positive integers
  - ”Unspecified” (or empty): using defaults = “125000” // params missing
  - p0: bitrate: 50000, 100000, 125000, 250000, 500000, 1000000 bit/s i.e. “250000”

bool **sendMessage**(short *cobID*, unsigned char *len*, unsigned char **message*, bool *rtr* = false)
Method that sends a message trough the can bus channel. If the method createBUS was not called before this, sendMessage will fail, as there is no can bus channel to send a message through.

Return Was the sending process successful?

Parameters

- `cobID`: Identifier that will be used for the message.
- `len`: Length of the message. If the message is bigger than 8 characters, it will be split into separate 8 characters messages.
- `message`: Message to be sent trough the can bus.
- `rtr`: is the message a remote transmission request?

and communication takes place through systec’s closed-source windows library.

5.1.2 linux

The open-source socketcan interface is used on top of systec’s open source driver:

class CSockCanScan : **public** CanModule::*CCanAccess*

Public Functions

bool **createBus**(**const** string *name*, string *parameters*)
Method that initializes a can bus channel. The following methods called upon the same object will be using this initialized channel.

- i.e. “250000”

Parameters

- `name`: = 2 parameters separated by “:”, like “n0:n1”
  - n0 = “sock” for sockets, used by systec and peak
  - n1 = CAN port number on the module, can be prefixed with “can”
  - ex.: “sock:can1” speaks to port 1 on systec or peak module
  - ex.: “sock:1” works as well
- `parameters`: one parameter: “p0”, positive integer
  - ”Unspecified” (or empty): using defaults = “125000” // params missing
  - p0: bitrate: 50000, 100000, 125000, 250000, 500000, 1000000 bit/s, other values might be allowed by the module

Return Was the initialization process successful?

bool **sendMessage**(short *cobID*, unsigned char *len*, unsigned char **message*, bool *rtr* = false)
Method that sends a message trough the can bus channel. If the method createBUS was not called before this, sendMessage will fail, as there is no can bus channel to send a message through.

Return Was the initialisation process successful?

Parameters

- `cobID`: Identifier that will be used for the message.
- `len`: Length of the message. If the message is bigger than 8 characters, it will be split into separate 8 characters messages.
- `message`: Message to be sent trough the can bus.
- `rtr`: is the message a remote transmission request?

sockets are used normally, using linux' built-in CAN protocols:

```
mysock = socket(domain=PF_CAN, type=SOCK_RAW, protocol=CAN_RAW)
```

5.2 standard CanModule API example

This is how the CanModule standard API is used for systec for linux.

```
libloader = CanModule::CanLibLoader::createInstance( "sock" ); // linux, use "st" for␣
↪windows
cca = libloader->openCanBus( "sock:can0", "250000" ); // termination on frontpanel
CanMessage cm; // empty
cca->sendMessage( &cm );
```

CHAPTER 6

AnaGate

All modules from vendor *AnaGate* are handled by class AnaCanScan which manages the modules through their underlying vendor specific API and provides the standard generic CanModule API.

We support Anagate CAN-ethernet gateways: uno, duo, quattro, and the X2, X4 and X8.

Since these modules communicate to the host computer only via ethernet, at the fundamental level only classical tcp/ip ethernet is needed. Nevertheless the specific contents of the IP frames are wrapped up in an Anagate API for convenience, which is linked into the user code as a library. There are therefore no implementation differences between Linux and Windows. Here the underlying vendor specific classes and the specific parameters are documented.

The downside of Anagate CAN-ethernet modules is of course that the latency of the network has to be added to the bridge latency.

6.1 The connection

To connect to a specific port for I/O, and send CAN messages, the following methods are used.

class AnaCanScan : **public** CanModule::*CCanAccess*

Public Functions

bool **createBus**(**const** string *name*, **const** string *parameters*)

Method that initialises a CAN bus channel for anagate. All following methods called on the same object will be using this initialized channel.

Return was the initialisation process successful?

Parameters

- `name`: 3 parameters separated by ":" like "n0:n1:n2" n0 = "an" for anagate n1 = port number on the module, 0=A, 1=B, etc etc n2 = ip number ex.: "an:can1:137.138.12.99" speaks to port B (=1) on anagate module at the ip ex.: "an:1:137.138.12.99" works as well

- `parameters`: up to 6 parameters separated by whitespaces : "p0 p1 p2 p3 p4 p5" in THAT order, positive integers
  - "Unspecified" (or empty): using defaults = "125000 0 0 0 0 0" // all params missing
  - p0: bitrate: 10000, 20000, 50000, 62000, 100000, 125000, 250000, 500000, 800000, 1000000 bit/s
  - p1: operatingMode: 0=default mode, values 1 (loop back) and 2 (listen) are not supported by CanModule
  - p2: termination: 0=not terminated (default), 1=terminated (120 Ohm for CAN bus)
  - p3: highSpeed: 0=deactivated (default), 1=activated. If activated, confirmation and filtering of CAN traffic are switched off
  - p4: TimeStamp: 0=deactivated (default), 1=activated. If activated, a timestamp is added to the CAN frame. Not all modules support this.
  - p5: syncMode: 0=default, unused but reserved for future use i.e. "250000 0 1 0 0 1" (see anagate manual for more details)

bool **sendMessage**(short *cobID*, unsigned char *len*, unsigned char **message*, bool *rtr* = false)

send a CAN message frame (8 byte) for anagate Method that sends a message through the can bus channel. If the method createBus was not called before this, sendMessage will fail, as there is no can bus channel to send a message trough.

Return Was the sending process successful?

Parameters

- `cobID`:: Identifier that will be used for the message.
- `len`:: Length of the message. If the message is bigger than 8 characters, it will be split into separate 8 characters messages.
- `message`:: Message to be sent trough the can bus.
- `rtr`:: is the message a remote transmission request?

6.2 standard CanModule API example

This is how the CanModule standard API is used for anagate. The code is identical for linux and windows.

```
libloader = CanModule::CanLibLoader::createInstance( "an" );
cca = libloader->openCanBus( "an:can0", "250000 0 1" ); // termination, ISEG␣
↪controllers, p3, p4, p5 defaults
CanMessage cm; // empty
cca->sendMessage( &cm );
```

CHAPTER 7

Peak

All modules from vendor *Peak* are handled by class PKCanScan (windows) or CSockCanScan (linux) which both manage the modules through their underlying vendor specific API according to the OS. Both classes provide the standard generic CanModule API. Here the underlying vendor specific classes and the specific parameters are documented.

The modules from the families PCAN USB and USB Pro are supported.

7.1 The connection

To connect to a specific port for I/O, and send CAN messages, the following methods are used.

7.1.1 windows

the connection to a specific port for I/O is created by calling

class PKCanScan : **public** CanModule::*CCanAccess*

Public Functions

bool **createBus**(**const** string *name*, **const** string *parameters*)

Method that initialises a CAN bus channel for peak (using PEAK Basic) All following methods called on the same object will be using this initialized channel. Only USB interfaces for PEAK modules, and only NON FD modules are supported for now.

Return was the initialisation process successful?

Parameters

- `name`: = 2 parameters separated by ":" like "n0:n1"
  - n0 = "pk" for peak
  - n1 = CAN port number on the module, can be prefixed with "can": 0..N

  - ex.: “pk:can1” speaks to port 1 (the second port) on peak module at the ip
  - ex.: “pk:1” works as well
- `parameters`: one parameter: “p0”, positive integer
  - ”Unspecified” (or empty): using defaults = “125000” // params missing
  - p0: bitrate: 50000, 100000, 125000, 250000, 500000, 1000000 bit/s i.e. “250000”

bool **sendMessage**(short *cobID*, unsigned char *len*, unsigned char **message*, bool *rtr* = false)
method to send a CAN message to the peak module. we use the standard API “PCAN-Basic” for this for windows and we talk only over USB to fixed bitrate modules. The flexible bitrate (FD) modules can be implemented later as well: for this we need more parameters to pass and a switch between CAN_Write and CAN_WriteFD.

and communication takes place through peak’s open-source PCAN-Basic windows library. Only “plug-and-play” modules with USB interface and fixed datarate are supported by CanModule for now. PEAK’s flexible datarate (FD) modules can be added later on (they need some different API-calls and more complex parameters), and also other interfaces like PCI are possible for windows.The implementation is based on the PCAN-Basic driver.

7.1.2 linux

The open-source socketcan interface is used on top of peak’s open source netdev driver. Both Peak’s fixed and flexible datarate are working, although the fixed modules are recommended for bus compatibility. Only modules with USB interface are supported. The peak driver source is freely available and it can be configured to build several types of drivers, where we use peak’s netdev driver only. See PeakDriver for details on this. A PCAN-Basic driver is also available but the netdev driver is more performant and modern. The PCAN-Basic driver is used nevertheless for windows, and it offers better compatibility for all module families.

class CSockCanScan : **public** CanModule::*CCanAccess*

Public Functions

bool **createBus**(**const** string *name*, string *parameters*)
Method that initializes a can bus channel. The following methods called upon the same object will be using this initialized channel.

- i.e. “250000”

Parameters

- `name`: = 2 parameters separated by “:”, like “n0:n1”
  - n0 = “sock” for sockets, used by systec and peak
  - n1 = CAN port number on the module, can be prefixed with “can”
  - ex.: “sock:can1” speaks to port 1 on systec or peak module
  - ex.: “sock:1” works as well
- `parameters`: one parameter: “p0”, positive integer
  - ”Unspecified” (or empty): using defaults = “125000” // params missing
  - p0: bitrate: 50000, 100000, 125000, 250000, 500000, 1000000 bit/s, other values might be allowed by the module

Return Was the initialization process successful?

bool **sendMessage**(short *cobID*, unsigned char *len*, unsigned char **message*, bool *rtr* = false)
Method that sends a message trough the can bus channel. If the method createBUS was not called before this, sendMessage will fail, as there is no can bus channel to send a message through.

Return Was the initialisation process successful?

Parameters

- `cobID`: Identifier that will be used for the message.
- `len`: Length of the message. If the message is bigger than 8 characters, it will be split into separate 8 characters messages.
- `message`: Message to be sent trough the can bus.
- `rtr`: is the message a remote transmission request?

sockets are used normally, using linux' built-in CAN protocols:

```
mysock = socket(domain=PF_CAN, type=SOCK_RAW, protocol=CAN_RAW)
```

7.2 standard CanModule API example

This is how the CanModule standard API is used for peak for windows.

```
libloader = CanModule::CanLibLoader::createInstance( "pk" ); // windows, use "sock"␣
↪for linux
cca = libloader->openCanBus( "pk:can0", "250000" ); // termination on frontpanel
CanMessage cm; // empty
cca->sendMessage( &cm );
```

CHAPTER 8

Logging

CanModule uses LogIt for reporting information during runtime. LogIt uses the four components

- CanModule: general messages concerning CanModule itself
- CanModuleAnagate: messages related to AnaGate modules
- CanModuleSystec: messages related to SysTec modules
- CanModulePeak: messages related to Peak modules

for managing logging levels per vendor. The logging level of each component, if the component is used, can be set individually at any time once initialized. For windows the component names are as listed above, for linux the component CanModuleSock is used for Systec and Peak modules, but also both CanModuleSystec and CanModulePeak are mapped to CanModuleSock for convenience.

You can of course add your own components for specific logging, like MYCOMP in the code below.

The calling program ("main") uses CanModule and LogIt like this

```
#include <CanBusAccess.h>
#include <LogIt.h>
...
Log::LogComponentHandle myHandle = 0;
Log::LogComponentHandle canModuleHandle = 0;
Log::LOG_LEVEL loglevel = Log::INF;    // recommended default for production is WRN
bool ret = Log::initializeLogging( loglevel );
if ( ret )
        cout << "LogIt initialized OK" << endl;
else
        cout << "LogIt problem at initialisation" << endl;
Log::setNonComponentLogLevel( loglevel );
std::cout << "Log level set to " << loglevel << endl;
LogItInstance *logIt = LogItInstance::getInstance();

/**
 * LogIt component MYCOMP for main
 */
```

(continues on next page)

(continued from previous page)

```
Log::registerLoggingComponent( "MYCOMP", loglevel );
logIt->getComponentHandle( "MYCOMP", myHandle );
Log::setComponentLogLevel( myHandle, loglevel );
LOG(Log::INF, myHandle) << argv[ 0 ] << " logging for component MYCOMP";
// hourray, we should see this message because we are at INF

/**
 * LogIt component CanModule for generic Canmodule
 */
Log::registerLoggingComponent( CanModule::LogItComponentName, loglevel );
logIt->getComponentHandle(CanModule::LogItComponentName, canModuleHandle );
Log::setComponentLogLevel( canModuleHandle, loglevel );
LOG(Log::INF, canModuleHandle) << " logging for component " <<␣
↪CanModule::LogItComponentName;
```

then, some work is done i.e. like that:

```
// do sth useful with CanModule, i.e. talk to a port
string libName = "st";           // here: systec, windows
string port = "st:can0";         // here: CAN port 0, windows
string parameters = "Undefined"; // here: use defaults
CanMessage cm;
CanModule::CanLibLoader *libloader = CanModule::CanLibLoader::createInstance( libName␣
↪);
cca = libloader->openCanBus( port, parameters );
cca->sendMessage( &cm );
cca->canMessageCame.connect( &onMessageRcvd ); // connect a reception handler, see␣
↪standardApi for details
```

and at any time the logging levels of the components can be changed. This is typically done by a user interaction on a running server instance.

```
/**
 * manipulate LogIt levels per component for testing, during runtime. Set loglevel to␣
↪TRC (max verbosity)
 */
loglevel = LOG::TRC;
Log::LogComponentHandle anagateHandle;
logIt->getComponentHandle(CanModule::LogItComponentNameAnagate, anagateHandle );
Log::setComponentLogLevel( anagateHandle, loglevel );
LOG(Log::INF, anagateHandle) << " logging for component " <<␣
↪CanModule::LogItComponentNameAnagate;

#ifdef _WIN32

Log::LogComponentHandle stHandle;
logIt->getComponentHandle(CanModule::LogItComponentNameSystec, stHandle );
Log::setComponentLogLevel( stHandle, loglevel );
LOG(Log::INF, stHandle) << " logging for component " <<␣
↪CanModule::LogItComponentNameSystec;

Log::LogComponentHandle pkHandle;
logIt->getComponentHandle(CanModule::LogItComponentNamePeak, pkHandle );
Log::setComponentLogLevel( pkHandle, loglevel );
LOG(Log::INF, pkHandle) << " logging for component " <<␣
↪CanModule::LogItComponentNamePeak;
```

(continues on next page)

(continued from previous page)

```
#else
// for linux we can also just use LogItComponentNameSystec and LogItComponentNamePeak
Log::LogComponentHandle sockHandle;
logIt->getComponentHandle(CanModule::LogItComponentNameSock, sockHandle );
Log::setComponentLogLevel( sockHandle, loglevel );
LOG(Log::INF, sockHandle) << " logging for component " <<␣
↪CanModule::LogItComponentNameSock;

#endif
```

CHAPTER 9

Building

CHAPTER 10

Running

The proper kernel modules, drivers and libraries for the vendors used through CanModule have to be present during runtime. Please also refer to Status for an overview over runtime conditions (april 2019).

10.1 general dependencies

- CanModule.dll/.so
- LogIt (cloned from github during cmake)
- boost 1.64.0
- xerces 3.2 (xerces-c_3_2D.dll)

10.2 Anagate

- libancan.dll/.so (standard API)
- linux: libAPIRelease64.so, libCANDLLRelease64.so, libAnaGateExtRelease.so, libAnaGateRelease.so
- windows: ANAGATECAN64.dll

10.3 Systec

- linux: libsockcan.so (standard API), driver kernel module systec_can.ko and dependent modules
- windows: libstcan.dll (standard API), USBCAN64.dll

10.4 Peak

- linux: libsockcan.so (standard API), driver kernel module pcan.ko and dependent modules
- windows: libpkcan.dll (standard API), PKCANBASIC.dll

CHAPTER 11

Support

Support for CanModule is given CERN wide.

Problems, Issues and Requests should be created as CERN Jira Tickets:

- Project= OPC UA in JCOP (OPCUA)
- Components= CanModule
- Assignee= Michael Ludwig

Please provide traces of your situation and information about your project context. We will sort it out together.

More **personal** ways to get help or report problems:

- You can send me an Email or call me 163095 or visit me for a coffee.
- Please also take a look at JiraSearch for already existing issues.

CHAPTER 12

alphabetical index

- genindex