

The BuildingBus Development Guide

By Glenn Weinreb, CTO, Manhattan 2, Printed Jan 30, 2021

Overview

BuildingBus is a hardware and software system that networks together devices in a building and this document is a guide to developing BuildingBus software. The topics discussed are in no particular order and it is therefore recommended that one use the Navigation System to locate areas of interest. For an overview of this project, please refer to the Manhattan 2 [Smart Building Development Initiative](#).

Operating System for Device in a Building

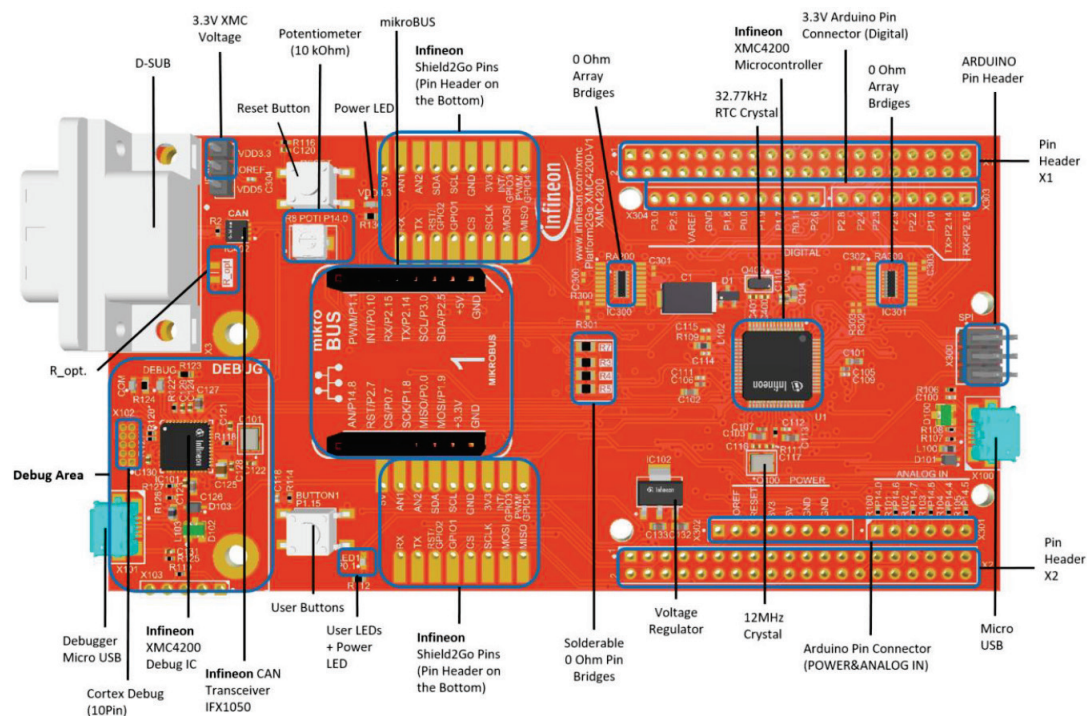
There is an operating system in Windows Computers called "Windows OS" and an operating system in Google phones called "Android". Yet no standard operating system exist for devices in a building. We aim to change this, with free and open C/C++ BuildingBus software.

This software is installed on all devices in the BuildingBus system, which means each device knows what is happening on other devices. Also, this means we have an agreed upon method by which devices interact, which facilitates local intelligence (smarter devices) and fault tolerance (each device is less dependent on external resources).

Xmc4200 Platform2Go Development Board

The BuildingBus Development Initiative uses an Xmc4200 [Platform2Go](#) development board for most projects. This provides an Arduino shield socket, a Mikro Click socket, and a powerful Xmc4200 microcontroller.

The Xmc4200 is one \$3 IC that provides 256KB of flash memory, 40KB



of RAM memory, two 1Ms/sec 12bit A/D converters, 8 analog input channels, two CANbus interfaces, several SPI interfaces, many counter/timers, and about 30 digital I/O pins. It can directly control a DC-to-DC converter, LED dimmable current source, and motor controller; for example. For details, please see its [datasheet](#) and [product page](#).



See Also

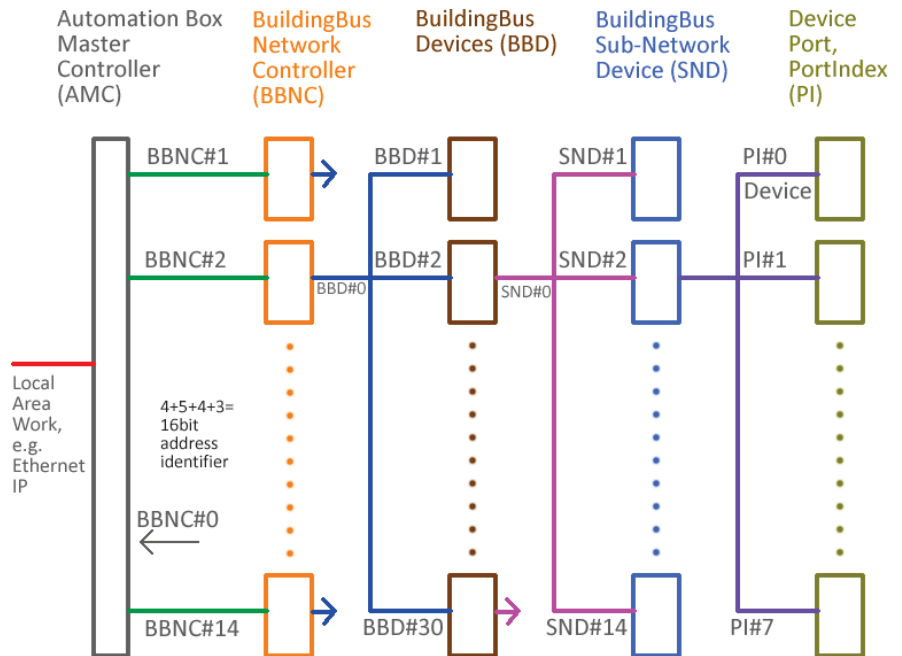
- [Smart Building R&D Initiative](#)
- [Rollable Solar R&D Initiative](#)
- [Fan and Damper R&D Initiative](#)
- [BuildingBus Development Guide](#). For free and open source code, click [here](#).
- [IoT Reference Guide](#)

Chapter 1) The BuildingBus System

Network Hierarchy

The BuildingBus network connects together multiple devices via CANbus. This supports sending 12byte (8+4) message packets along a wire.

The system is hierarchical in the sense that a Master Controller ("AMC") manages multiple Network Controllers, each Network Controller manages multiple Devices, and each Device optionally manages multiple Subnetwork Devices, as illustrated to the right.



An example network is a Network Controller that routes its data wire along an 110VAC power wire from the fuse box in the basement to multiple devices in the living room. One device might be a manager for 10 light sockets in the living room ceiling, each of which is a subnetwork device. Each physical window might have several motors (e.g. curtains, thermal cover), each of which are subnetwork devices, with one window controller device managing the subnetwork.

Reliability, Revenue and Relevance

Breaking this up into many different networks means that if one wire breaks, the entire system is not affected. Wired (not wireless) communication typically results in reliability better than >99.999% of the time being operational. Wireless is more like 90% to 99%, in many cases. Reliability is a thing, and people who design and build buildings need it in order to facilitate happy customers. If we want to be relevant; we need to provide good reliability, fault tolerance, and quality.

Network Address

Each device in the system is identified with a three number address (network #, device #, subnetwork #). Each device internally contains three to eight software ports, each of

which are a C++ class. When one talks to a device, they specify the portIndex they are talking to (0...7).

This is illustrated in the above picture, which is referred to as one "BuildingBus System". The master controller connects to the outside world via IP (e.g. Ethernet). When one goes out IP, they are "outside" the system. One can have multiple BuildingBus Systems in one building, connected together via IP.

Time and Design

A typical internet connection moves continuous data at 10M to 100M bits-per-second with a 30mSec to 300mSec delay between transmission and reception, with each data packet being thousands of bytes in size.

BuildingBus is different in many ways. Our delay between transmission and reception is more like 10uSec to 5mSec (due to multiple bridges), our packet size is 12bytes, and typical bit rates are 100K bits-per-second. BuildingBus is not designed to move computer data, audio or video. It is designed to implement building automation and control, reliably, and at low cost. Reliability in the 99.999% operational range entails wire (not wireless or powerline communication). Wire entails CANbus, since that is the wired multi-device communication system built into low cost microcontroller IC's. CANbus entails 12byte data packets, since that is CANbus. Building 10m to 50m physical distances entails ~100K bits-per-second, since that is what you get when you have tree topology wiring of that size (not daisy-chain, no termination resistors, $\sim 60\text{pF/m} \times \sim 330\Omega \text{ source impedance} \times \sim 50\text{m} = \sim 1\mu\text{Sec}$).

The 12byte data packets are sufficient since automation and control often entails reading a sensor or setting a control system, which can be done with several bytes. BuildingBus sometimes move thousands of bytes at a time, yet that is often done with immutable data that never changes, and therefore only needs to be done once. If 1K bytes of immutable data characterizes each device, and you have 300 devices, then a 300KB device library can be copied to servers, computers, tablets and smartphones. This can help them fully understand the system in great detail, without passing data on the network. Maintaining device libraries is part of the BuildingBus system.

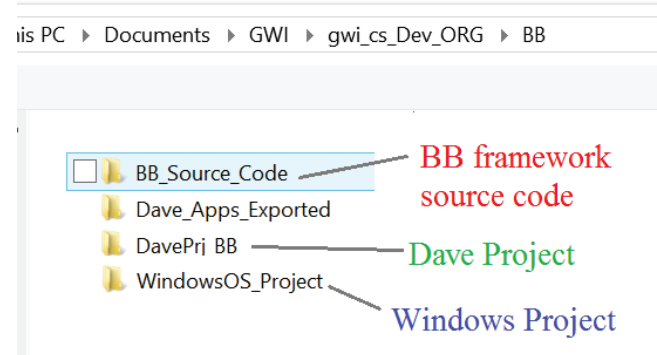
Chapter 2) Getting Started With the BuildingBus Source Code

Downloading BB

To download BuildingBus source code and projects, please see the following URL. We show a picture instead of a URL since security software sometimes resists zip. One needs to type this URL into their browser to download.

www.ma2life.org/doc/src/BB.zip

This file contains the BuildingBus framework source code (folder BB_Source_Code), the DavePrj_BB project (folder DavePrj_BB), Dave Workspace (folder DavePrj_BB_Workspace), and a Windows OS project (folder WindowsOS_Project). The Windows project creates multiple devices in a system; whereas the DAVE project creates one device that runs on one Xmc4200 Platform2Go hardware board.



To view in Visual Studio, open project file:

BB\WindowsOS_Project\BuildingBus_Development\BuildingBus_Development.sln

To view in Dave IDE, open workspace folder:

BB\DavePrj_BB\DavePrj_BB_Workspace\

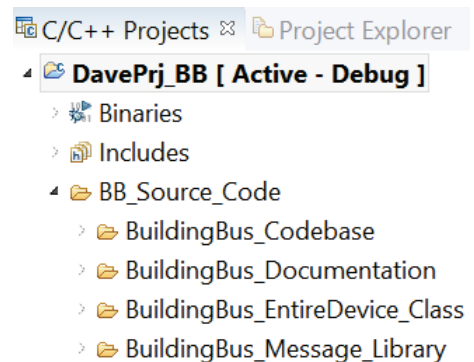
It is better to open the Dave workspace than import the DavePrj_BB project file, since the workspace contains Debug Configuration files that relate to the project.

One Project Can Create Different Devices

One can set compiler switches as needed to create the following devices with the DavePrj_BB project: window thermal cover controller, window curtain motor controller, window rolled blind motor controller, window venetian blind motor controller, window controller, network controller, automation master controlled, LED light socket subnetwork device, LED light device controller, window wall light switch/dimmer control, hvac variable speed or on/off controller, hvac duct damper controller, and radiator valve controller. To view compiler switches that control the project, see file "MY_Project_Compiler_Switches.h".

The DavePrj_BB Project

The DavePrj_BB project contains the BuildingBus framework and supports debugging using free Infineon [DAVE](#) tools. This project creates one device, as determined by the BB_BUILD_STRATEGY compiler symbol. For example, set it to ...Window_Motor... to build a Window Motor device; set it to ...Light_Socket... to build a light socket device; or set it to ...SolarArrayElement to build a power converter that attaches to a solar panel.



This project can read from five analog input channels, write to one digital output bit attached to an LED, and write to one digital output bit attached to an oscilloscope to measure timing. This project uses counter/timers to create a 64bit 128uSec hardware counter that establishes time since midnight, Jan 1, 2020. Also, it includes a hardware interrupt system that generates interrupts every 10mSec. This drives one master thread that executes tasks at a multiple of this rate. We keep it simple, with one thread, to make it easier for programmers.

Compiler Switches

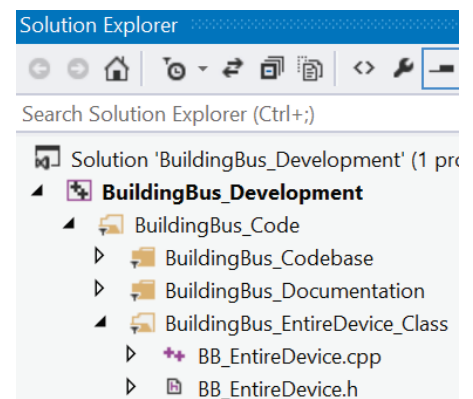
Compiler switches in file MY_Project_Compiler_Switches.h determine what is built and how it is built. For example, switches in this file can turn simulation on or off, determine how many devices are created, and determine which devices are created.

Getting Started with the DavePrj_BB Project

To get started install the Infineon DAVE software on your computer, download the above zip file, open the Workspace (BB\DavePrj_BB\DavePrj_BB_Workspace\), compile the DavePrj_BB project, run it under the DAVE debugger, and see if you can print to the console window. The TestSystemA class might be running, in which case, it will send test messages to a test device.

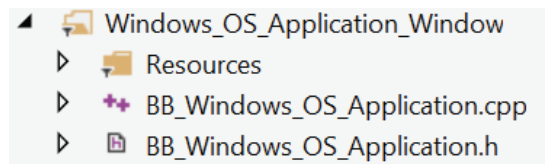
BuildingBus_Development Visual Studio Project

The BB zip file also includes the BuildingBus_Development Visual Studio project, which creates multiple devices and tests them in a system. It tests message passing, moving large blocks of data between devices ("streams"), building device libraries, and the coordination of devices in a system.



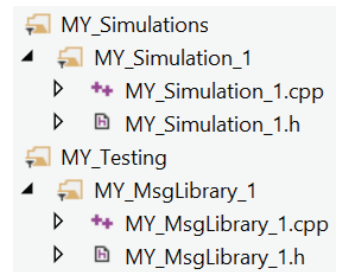
Windows Application Program

Files `BB_Windows_OS_Application.cpp/.h` create a Windows Application program that provides a console window for printing. All code that is specific to Microsoft Windows is contained in these two files; and they are compiled out when building one device under Infineon DAVE software.



Simulate 20 Devices in a System

The visual studio project creates 20 devices and tests them in a system. To see the code that creates these devices, search ":Create_Devices" from within the visual studio project. 95% of the BuildingBus code was developed using this environment. The simulation files are compiled out when building one device under DAVE.



BB and MY

File names with a "BB_" prefix are maintained by "BuildingBus Engineers", which are people that develop the BuildingBus system. File names with a "MY_" prefix, are maintained by Industry Engineers, who are responsible for designed and manufacturing products. Industry engineers do not change the BB files. They use them and rely on them to maintain the system.

BuildingBus Strategy

BB files, in effect, define the BuildingBus software standard. Your traditional networking standard does not include software, and therefore needs to maintain a degree of simplicity in order to gain support from participants, who need to write code that implements the standard. Alternatively, in the case of BuildingBus, the software that manages the device is supplied, and therefore can be significantly more complicated than your traditional networking protocol. To encourage adoption, BuildingBus is free and open software -- anyone can copy and modify at no cost. Normally, business do not develop free and open since they want to make money. However, in our case, we write code to make buildings smarter, to reduce CO₂ emissions.

Student Contribution

BuildingBus is complicated and can therefore be overwhelming to students. Subsequently, it is suggested that each focus on one somewhat small area and develop C or C++ code, or electrical schematics, that can be used by others. For example, one might create a device that controls a 10W LED light bulb, focusing on the hardware interface to the bulb, while BuildingBus manages the interface to other devices on the network. In another example,

one might write code that scans through device libraries and identifies devices that satisfy a specific filter requirement. This could then be folded into the BuildingBus framework, to be used by any programmer working with any device.

Chapter 3) The DavePrj_BB Project

Overview

The DavePrj_BB Project compiles under the Infineon DAVE development environment and supports the downloading and debugging of code on an Xmc4200 Platform2Go Board.

One Thread Takes Care of Most Business

The system utilizes one thread. This makes it easier for programmers since one does not need to be concerned with memory contention or blocking.

A hardware interrupt occurs once every 10mSec and calls EXECUTE_MasterIdleChore (), which periodically calls other routines.

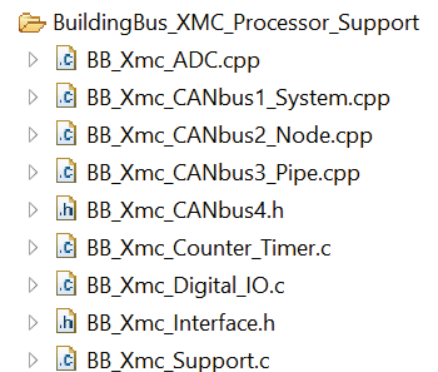
The following are set up to execute once every 100mSec: Stream Manager, State Machine, and Device Manager. Also, processing of CANbus messages occurs once every 10mSec via EXECUTE_Foreground_Processes_Multiple_INCOMING_CanBus_Msgs ().

For code that manages counters, timers, and processing; see file BB_EntireDevice_Processing.cpp.

XMC Processor Code

All code that interacts with the XMC microcontroller hardware is contained in several files, pictured to the right. These files are compiled out when working with Windows Visual Studio.

- BB_Xmc_Counter_Timer.c sets up and maintains hardware timers and counters.
- BB_Xmc_ADC.cpp creates a class that manages the processor's A/D converter and measures voltages at processor analog input pins.
- BB_Xmc_Cpu_Digital_IO.c manages digital I/O bits.
- BB_Xmc_CANbus... manages the CANbus interface.



The [DAVE](#) Apps do much of the work; subsequently, little programmer code interacts with the XMC microcontroller.

BB EntireDevice C++ Class

In software, each device is one instance of the BB_EntireDevice class. This class creates other classes, as needed, and maintains pointers to them. For example, BB_EntireDevice creates an instance of the BB_LibraryManager class, which manages libraries of other devices. See file BB_EntireDevice.h for a description of this class.

- ◀ BuildingBus_EntireDevice_Class
 - > BB_EntireDevice_Addressing.cpp
 - > BB_EntireDevice_NetworkRouter.cpp
 - > BB_EntireDevice_Processing.cpp
 - > BB_EntireDevice_Testing.cpp
 - > BB_EntireDevice.cpp
 - > BB_EntireDevice.h

Each product creates a child class of the BB_EntireDevice class within a "MY" file. As noted previously, BB files do not change from device to device, yet MY files contains information unique to one's device. In other words, the BB_EntireDevice class source code is the same in all devices in the system (except for updates to code, which is beyond the scope of this discussion). For an example of a child class that creates a motor controller for a physical wall window, see file MY_Dev555_EntireDevice.cpp.

In summary, MY files tend to contain child classes to base classes that were defined in BB files, children add to parents, children define unique attributes, and parents do ~90% of the work.

Global Pointer to Device (gDevP)

When compiling one device under DAVE, a pointer to the created BB_EntireDevice is loaded into global variable gDevP. Subsequently, any code can access any BB_EntireDevice method, or class variable, via gDevP. For details, search the DavePrj_BB project for "gDevP".

A/D Analog Measurement

Class BB_XMC_Hardware_ADC_Measurement_System maintains an interface to the XMC microcontroller internal A/D converter and analog input channels. There are only two XMC library C subroutines that talk to the A/D hardware:

ADC_MEASUREMENT_StartConversion () starts conversions and
ADC_MEASUREMENT_GetResult () returns the result.

Interrupt service routine ADC_Complete_ISR_HigherPriority () is called when the A/D completes. This sets a global flag, while the main thread sits in a while () loop waiting for this flag to be set. This is all implemented with Measure_One_ADC_Channel (); therefore one does not need to deal w/ the XMC A/D support routines, and instead can work with this higher level routine that takes two parameters: channelIndex (selects one of the

analog input channels) and `uSec_integrationTime` (sets how many samples are average before the average is returned).

Integration is powerful. If a signal is noisy and returns codes that bounce up and down ± 10 samples, and then average 100 samples, they will reduce noise to ± 1 .

For a routine that test the A/D measurement system, see `Test_XMC_ADC_Measurement_NOISE_VS_INTEGRATION ()`.

The system measures the time it takes to read one a/d sample and then uses this to calculate the number of samples to average, given a requested integration time (i.e. # of microseconds). To see this measurement, search "`weAreMeasuringAdcConversionTime = true`".

If your device has sensors, it will managed them with an instance of the `MeasurementSys` class. Code within the device will then read those sensors via this class's `Measure_One_Analog_Input_Channel ()` method, which in turn calls lower level XMC `Measure_One_ADC_Channel ()`.

Debugging via `LogErrorCode ()`

When we first incur an error, we call `LogErrorCode ()`. This prints an error message to the debugging console window, if it is set up. Yet more importantly, we place a break point inside `LogErrorCode ()`, and use debugging features to resolve the bug before continuing (e.g. via stack crawl, view variables, etc).

```
*BbError_int16 LogErrorCode(BbError_int16 errorCode)
{
    // -----
    // !!! PLACE BREAK POINT HERE !!!
    // -----

    gPutDebuggerBreakpointHere++;
}
```

As of 2/1/2021, there are no known bugs in the BuildingBus framework software. This is in part due to this system of chasing down the source of a problem before continuing with coding. In other words, we never continue execution after stopping inside `LogErrorCode ()`.

Code is constantly looking for problems, and calls `LogErrorCode` when found.

Simulations entail passing hundreds of messages between dozens of devices, and when an error is incurred, `LogErrorCode` is called.

When a subroutine hits an error, it returns an error code, which is seen by the calling function, which in turn returns an error code in a nested fashion. We only place `LogErrorCode ()` at the lowest level, since we only need to print once to the console

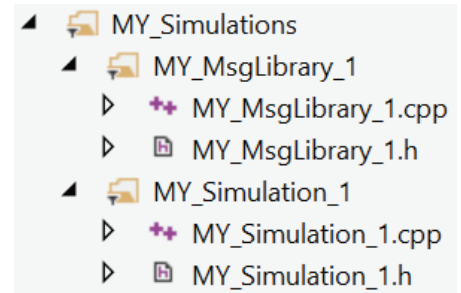
window. Yet more importantly, when an error occurs, we want to debug as close to the problem as possible, as opposed to far from the cause.

Chapter 4) Simulating Multiple Devices

Overview

The BuildingBus_Development Visual Studio project creates multiple devices, in software, and tests them in a system. This does not use microcontroller hardware.

As of Jan 2021, the simulation creates 1 to 20 devices, depending on the maxNumOfDevices parameter passed to CreateDevices_CreateWires_ConnectDevicesToWires (). To see which devices are created and their addresses, view Create_Devices_TOTAL ().



Creating Devices

In BuildingBus terminology, a "device" is one microcontroller IC, in one product, that is attached to the network. Each device is identified with a network #, device #, and subnetwork # address. A "BuildingBus System" involves multiple devices connected together, with a master controller at address 0/0/0. For examples of routines that create devices, see:

- Dev555_BUILD_AMC_MasterController_0_0_0 () creates a Master Controller (AMC) at address 0/0/0.
- Create_Devices_WINDOW_THERMAL_COVERS () creates multiple window motor subnetwork devices, and multiple window controller devices.
- Create_Devices_LIGHTING () creates multiple light sockets (e.g. living room ceiling is one subnetwork and kitchen ceiling is another).
- Create_Devices_SOLAR_ARRAY () creates multiple solar panel power converters (e.g. one 300Watt DC-to-DC converter subnetwork device for each 3x5ft solar panel).

Creating CANbus Cables

Create_Wires () creates multiple CANbus cables, and Add_Devices_To_Wires () attaches devices to those wires, as needed. Later, when a device pushes a message into a cable, it is copied into the receive FIFO buffer of all devices connected to that cable. Subsequently, multiple devices can pass messages to each other in a network like environment, without the actual hardware.

Simulating Multiple Devices

Several different simulations are set up with functions in file MY_Simulation_1.cpp:

- `RUN_Simulation_ToTest_SIMPLE_MSG_PASSING ()`: Simulates passing a few simple messages between devices.
- `RUN_Simulation_ToTest_MSG_ROUTING ()`: Every device passes a message to every other device, and the system checks to make sure it is received. If you have 20 devices, then this will test $(20-1)*(20-1) = 361$ different permutations. The simulation system assigns a unique ID to each device when the device is created (`globalDeviceID`). In this simulation, this value is sent to a 16bit test register within each device (`TestRegister16`). After the message is passed, the system checks this test register to make sure it was set properly, and also looks at test registers in other devices to make sure they were not inadvertently set.
- `RUN_Simulation_ToTest_BROADCAST_TO_ALL_DEVICES ()`: Messages can be sent to multiple devices via a system called "broadcast". One type of broadcast involves sending one message to all devices in the system, which is tested in this simulation.
- `RUN_Simulation_ToTest_BROADCAST_TO_SPECIFIC_ZONES ()`: This is similar to the above simulation, yet tests another type of broadcast that involves sending one message to a subset of devices within the entire system. For details on how this works, search the project for "Devices That Receive Broadcast".
- `RUN_Simulation_ToTest_READ_1x_DATA_CAPSULE_VIA_DATA_STREAM ()`: This simulation tests the reading of an entire data capsule (multiple fields) via `StreamIO` (i.e. multiple messages transfer one large binary block).
- `RUN_Simulation_TEST_DeviceCtrl_Manager_Discovery_Process ()`: When devices first boot up, they build libraries that contain information about other devices. This involves state machines and communication between devices, as described in the below Device Library discussion.

Simulation Engine

The actual simulation involves giving processor time to each device to digest incoming message packets, and moving messages between devices. This is implemented with `Execute_Simulation_Engine ()`, which simulates a fixed number of OS "ticks", where each tick is a call to a device's 10mSec Master interrupt service routine (ISR). For example, if one runs 10K ticks, they would simulate 100 seconds of activity ($10\text{mSec} * 10\text{K} = 100$). The simulation engine iterates through all devices in the system and calls their 10mSec ISR to simulate one tick for the entire system.

Running a Simulation

To run a specific simulation, search for "void MY_Project_Run_Simulations", select a simulation, and set compiler switches as needed.

```
void MY_Project_Run_Simulations( void )
{
    #ifdef START_BuildingBus_Simulation_SIMPLE_MSG_PASSING
        (new MY_Simulation_1())->RUN_Simulation_ToTest_SIMPLE_MSG_PASSING();
    #endif

    #ifdef START_BuildingBus_Simulation_MSG_ROUTING
        (new MY_Simulation_1())->RUN_Simulation_ToTest_MSG_ROUTING();
    #endif

    #ifdef START_BuildingBus_Simulation_BROADCAST_TO_ALL_DEVICES
        (new MY_Simulation_1())->RUN_Simulation_ToTest_BROADCAST_TO_ALL_DEVICES();
    #endif

    #ifdef START_BuildingBus_Simulation_BROADCAST_TO_SPECIFIC_ZONES
        (new MY_Simulation_1())->RUN_Simulation_ToTest_BROADCAST_TO_SPECIFIC_ZONES();
    #endif

    #ifdef START_BuildingBus_Simulation_READ_1x_DATA_CAPSULE_VIA_DATA_STREAM
        (new MY_Simulation_1())->RUN_Simulation_ToTest_READ_1x_DATA_CAPSULE_VIA_DATA_STREAM();
    #endif

    #ifdef START_BuildingBus_Simulation_TEST_DeviceCtrlr_Manager_Discover_Process
        (new MY_Simulation_1())->RUN_Simulation_TEST_DeviceCtrlr_Manager_Discovery_Process();
    #endif
}
```

Overview

Each device maintains 3 to 8 software ports (C++ classes that handle messages), each port maintains 0 to 3 different sets of fields, and each set of fields consists of 0 to 127 fields. A field is a single number, or an array, or a string. Each field element is of type floating point or integer. Each field element is 1, 2, 4, or 8 bytes in size. An example field might be the DeviceType, which indicates the type of device (e.g. window motor). Each set of fields is called a "data capsule". We call this a "capsule" since it is stored in one continuous binary block that is designed to be portable. We call this "portable" since it is designed to be moved throughout the network. Any device can ask any other device for a copy of one or more capsules. Each capsule is typically 50 to 500 bytes long. One can also request a set of capsules packed into one binary block called a "BuildingBus Library".

ImmutableData, ImmutableStrings, and PortRegisters

Each port maintains up to 3 different capsules. One is called "ImmutableData", which consists of numbers that never change (e.g. device type). Another is called "ImmutableStrings", which are strings that never change (e.g. vender name). And the third is called "PortRegisters", which are numbers that do change (e.g. window motor position, temperature sensor °C value).

Internal Structure

Capsules start with a header struct (GENERIC_BbCapsule), which is followed by information that describes the contents of the capsule, which is followed by field data. A struct that defines the entire capsule sometimes does not exist. In those cases, one reads from fields, and writes to fields, via subroutines that use capsule internal information to determine field type, and position, within the capsule.

Reading and Writing Fields within a Portable Capsule

For an example routine that reads a field, see Set_INT64_Value_WithinCapsule (). For an example of a routine that writes to a field, see Get_FLT32_Value_FromCapsule (). For routines that return information about a field, see Get_ExtendedInfo_OneField_BbCapsule () and Get_Info_OneField_BbCapsule (). These C routines operate on capsules which may have been created by other devices. Subsequently, they do not have access to C struct definitions that tell them about the capsule. For C routines that work with portable data in capsules, see file BB_Capsule_Interface.c.

Capsule Container Class

Each device typically maintains 5 to 10 capsules, and for each, an instances of the BB_CapsuleContainer wrapper class provides additional methods. For example, method `Get_INT64_Value ()` reads a field and return its value in an int64 variable; and `Set_INT64_Value ()` sets a field's value given an int64 variable. For details, search "class BB_CapsuleContainer".

Flt32 and Int64 Interface

Some of the read/write field routines are passed 32bit floating point variables, whereas others work with 64bit integer. If a field internally stores the value 3 in an int8, and you read it with `Get_FLT32_Value ()`, it will set your flt32 to 3.0, for example. Alternative, if you read the same byte with `Get_INT64_Value ()`, it will set your int64 variable to 3. This might seem wasteful, yet the alternative of providing more r/w routines, which is more wasteful.

Flexibly Sized Signed Integer

When reading/writing fields across the network, the # of bytes used to represent the data varies depending on the value. For example, if a field internally stores a number in an int64 variable, and it contains the value 3, then reading this field across the network results in 1 byte being transferred (since more bytes are not needed to represent a low value). For details, see `CONVERT_FROM_FlexiblySizedSignedInteger_TO_INT64 ()`.

Iterating Through Ports, Capsules and Fields

For an example of a routine that iterates through all ports, all capsules, and all fields within one device; see `TEST_EveryFieldInEveryCapsule ()`. This reads each field with both a C routine that reads the capsule directly; and a C++ container class method. Also, this gets information on each field, by both the direct C routine and C++ container method.

For examples of routines that locate ports and capsules, that are called from within any port class (not device class), see `GET_AnyPortInThisDevice ()`, `GET_CapsuleContainerP ()`, and `GET_CapsuleContainerP_FromAnyPortInThisDevice ()`.

Device Common Port

All devices provide a common interface between the network and the entire device via an instance of the BB_DeviceCommon_Class at Port #0 (portIndex = 0 = PortIndex_0_DeviceCommon). Its ImmutableStrings contain things like manufacturer name and its ImmutableData contain fields like ProductType.

BB_DeviceCommon_Class, defined in a BB file (same in all devices in network), does much of the work, while the child class in the MY file contains information unique to the device (e.g. class MY_Dev555_port_DeviceCommon). Together, these files manage capsules of fields, and provide function handlers at Port #0.

For information on fields maintained by the ImmutableData, ImmutableStrings, and PortRegister capsules (3 different sets of fields) within the DeviceCommon port, see file BB_DeviceCommon_Interface.h.

Measurement System Port

Each device maintains up to 127 sensors via the BB_MeasurementSys class at port #2 (portIndex = 2 = PortIndex_2_MeasurementSystem). If you are talking to a device's port #2, you are talking to a child of this class. And this child maintains three capsules (i.e. sets) of fields (ImmutableData, ImmutableStrings, and PortRegisters).

MeasurementSys PortRegister Field #15

(msrFI_Sensor_Measurement_Channel) is the most interesting, since it is the interface to sensors (e.g. A/D analog input channels). This field is an array, where each element of the array corresponds to a different sensor. For example, if you have 5 sensors, your channelIndex would vary from 0 to 4, and would correspond to an index into this 5 element array. Anyone that reads an element in this array is getting a pseudo real-time reading of that sensor. When a device reads a sensor on another device, it

- ✦ BuildingBus_Port_DeviceCommon
 - BB_DeviceCommon_Class.cpp
 - BB_DeviceCommon_Class.h
 - BB_DeviceCommon_Func_Library.cpp
 - BB_DeviceCommon_Func_RdWrField.cpp
 - BB_DeviceCommon_Func_TestSuite.cpp
 - BB_DeviceCommon_Handlers.cpp
 - BB_DeviceCommon_Interface.h
 - BB_DeviceCommon_Setup.cpp
- ✦ MY_Dev555_port_DeviceCommon
 - MY_Dev555_port_DeviceCommon_setup.cpp
 - MY_Dev555_port_DeviceCommon.cpp
 - MY_Dev555_port_DeviceCommon.h

- ✦ BuildingBus_Port_MeasurementSystem
 - BB_Port_MeasurementSys_Class.cpp
 - BB_Port_MeasurementSys_Class.h
 - BB_Port_MeasurementSys_Functions.cpp
 - BB_Port_MeasurementSys_Handlers.cpp
 - BB_Port_MeasurementSys_Interface.h
 - BB_Port_MeasurementSys_Setup.cpp
- ✦ MY_Dev555_port_MeasurementSys
 - MY_Dev555_port_MeasurementSys_setup_ncr.cpp
 - MY_Dev555_port_MeasurementSys.cpp
 - MY_Dev555_port_MeasurementSys.h

reads `PortIndex_2_MeasurementSystem` (`portIndex = 2`), `BbCapsuleType_PortRegisters` (`capsuleType = 2`), and `msrFI_Sensor_Measurement_Channel` (field index).

The `MeasurementSys ImmutableData` capsule contains fields like sensor type, which tells us the type of sensor (e.g. temperature). To see a list of different sensor types, search "enum `BB_SensorType`". This capsule also provides things like sensor minimum value, sensor maximum value, measurement accuracy, and measurement noise. These are immutable, which means they never change. Subsequently, this information ends up all over, including one's smartphone. This means your smartphone has access to information on all sensors in your house, which means it can easily determine how to utilize them without using the network.

For a list of `ImmutableData` fields in `MeasurementSys Port #2`, search "enum `FIELD_INDEX_MeasurementSys_ImmutableData`".

The `MeasurementSys PortRegisters` capsule contains fields that contain values that change. For a list of these, search "enum `FIELD_INDEX_MeasurementSys_PortRegister`". Example fields are sensor value and sensor status.

The `BB_MeasurementSys` class is identical on all devices (it is a "BB" file); whereas the child class is specific to a particular device (it is a "MY" file). For an example of MY file code that implements sensors, see `Setup_Hardware_ADC_Measurement_System()`, which sets up A/D channels, and `Measure_One_Analog_Input_Channel()`, which measures them.

VariableControl Port

Port #1 is the `DeviceType` handler port, which means it contains a class that implements the `DeviceType`. In many cases, we place a `VariableControl` class here, which manages one or more output channels. One writes an `int16` value to this port to control something. In some cases, 0 is off and any other value is on. In other cases, the device uses a value between 0 and 32500 to control a variable quantity.

All of the following products utilize variable control: on/off motors, variable speed motors, 0 to 100% position stepper motors, on/off lights, 0 to 100% illumination lights, on/off fans in ducts, and 0 to 100% open dampers in ducts and at vent openings.

When one device writes to the control output of another device, it writes to `PortIndex_1_DeviceTypeHandler` (`portIndex = 1`), `BbCapsuleType_PortRegisters` (`capsuleType = 2`), and `vcorFI_ControlOutput_CommandOutput_int16` (field index).

The Industry Programmer fills in the body of the `Update_One_Control_Output_Channel()` method in order to implement control.

For information on fields used by the capsules within the VariableControl port, see file `BB_Port_VariableControl_Interface.h`.

Chapter 7) Transferring Data across the Network

Working with Addresses

Devices are identified in the system with a network #, device #, and subnetwork # address. Also, a 0...7 portIndex refers to a specific port class within a device. These four values are packed into a uint16 number and placed into the 29bit identifier transmitted within CANbus packets. Also, in code, addresses are maintained inside a BB_AddressIdentifier struct. To load this struct with a 4 number address, one calls LOAD_AddressIdentifier (). For functions that work with network addresses, see file BB_EntireDevice_Address.cpp.

Broadcast and Groups

One can send a message to a specific device via a network #, device #, and subnetwork # address; or broadcast one message to all devices in the system; or broadcast one message to a zone within the system; or broadcast one message to a group of devices. For details, see "Broadcast and Groups" within [this](#) document.

Each Device can Read/Write any Field, in any other Device, within the System

Any device can read (and possibly write) to/from any other field within the system. Each field is identified with 6 or 7 parameters: network #, device #, subnetwork #, port#, capsule type (immutable data, immutable string, port register), field number, and index into an array if field is an array.

File "BB_DeviceCommon_Func_RdWrField.cpp" contains subroutines that read/write fields in other devices. When a device does this, they are referred to as the "initiator" or "client", and the contacted device is called the "target" or "server". The server responds to a R/W field request via method HANDLER__DeviceCommon_RdWrField ().

An example initiator routine is

APPEND_MSG__Read_INT64_AnyField_AnyCapsule_AnyPort (), which returns a 64bit integer value, independent of a field's internal data type. If a value of 3 is found within a field, it will respond by transmitting one byte back to the initiator, loaded with 3, and the caller's int64 will in turn be loaded with the value 3. For details on how values of any size are transferred using a variable number of bytes, search "FlexiblySizedSignedInteger".

One does not need to be concerned with how the system implements read/write field, since the code takes care of it. However; if curious, one can see file

"BB_CANbus_FramePacking.c" for details on how CANbus messages are packed and unpacked.

Field Index

Enums are used to keep track of which field does what, within each capsule, within each port. These are referred to as "field index". To view field indices used by the ImmutableData capsule of the DeviceCommon port, for example, search "typedef enum FIELD_INDEX_DeviceCommon_ImmutableData".

For enums that define all field indices, search "FIELD INDEX ENUM".






Field indices never change, which makes it possible for one device to r/w fields in any other device. If one updates their software, they can add a capsule, yet never delete, and never reorder.

Shown below are several fields within the DeviceCommon ImmutableData capsule, for example.

```
typedef struct DeviceCommon_ImmutableData_ADDITIONAL_FIELDS
{
    int32_t                manufacturerID_int32;           // Unique :
                                                                // Primary
                                                                // For deti
                                                                // is used
    int32_t                primary_UniqueSerialNumber_int32;
                                                                // If one r
                                                                // For deti
    int32_t                SECONDARY_UniqueSerialNumber_int32;
                                                                // Date and
    int64_t                dateAndTimeDeviceWasManufactured_1mSecUnits;
    int32_t                manufacturerDeviceVersion_int32; // Manufact
    int16_t /* BB_DeviceType_int16 */ deviceType_int16;    // Device '
                                                                // that is
                                                                // For deti
    int16_t /* BB_ProductType_int16 */ productType_int16; // Product
                                                                // Product
                                                                // For deti
```


Get Stream

Each device can request more than 8bytes from any other device via the Stream Manager, which moves large blocks of binary data via multiple CANbus messages. When one requests a stream, they specify what they want to receive, which is one of: one capsule (GetOneDataCapsule), multiple capsules in a library (OneDevice_MultipleCapsule_Library), information on multiple devices packed into a library (e.g. array of BB_DeviceSummary structs), multiple fields (GetMultipleCapsuleFields), or structs w/ important information. For details, search for "enum BB_StreamType".

- ✚ StreamIO
 - >  BB_ClientDataStream_.cpp
 - >  BB_ClientDataStream_Manager.cpp
 - >  BB_DataStream_Interface.h
 - >  BB_ServerDataStream_Manager.cpp
 - >  BB_ServerDataStream.cpp

To request a stream from another device, one first creates a message requesting the stream. For details on how to do this, search ":APPEND_MSG__StreamIO".

When a target receives a request for a stream, it processes it via method HANDLER__Server_Responds_To_StreamIO_Request ().

When the stream is complete, the client receives a callback (i.e. a specified functions is called).

An instance of class BB_ServerDataStream_Manager manages a set of BB_ServerDataStream instances, where each of the later maintains one stream on the target server device. And an instance of class BB_ClientDataStream_Manager manages a set of BB_ClientDataStream instances, where each of the later maintains one stream on the client initiator device.

The BB_EntireDevice class creates and maintains a server stream manager, and a client stream manager.

```
// create client DataStream manager, maintains a list of Client DataStream:
clientDataStream_ManagerP = new BB_ClientDataStream_Manager(this, client_sti

// create server DataStream manager, maintains a list of Server DataStream:
serverDataStream_ManagerP = new BB_ServerDataStream_Manager(this, serverStri
```

Streams enable one to move large blocks of data; yet more importantly, they facilitate the movement of important device information.

Message Builder

Each device uses the Message Builder class (BB_MsgBuilder) to create messages and move them through the network. This class maintains a queue (i.e. list) of messages. One begins by resetting the queue via `Reset_MsgQueue ()` and then calls `LOAD_TargetServerAddress_via_AddressIdentifier_uint16 ()` to specify the target device address. Then, one calls multiple `APPEND_MSG...` routines to append messages to the queue. For example, `APPEND_MSG__WriteInt64_DeviceCommon_PortRegister ()` appends a message that writes a value to a field in another device.

After appending messages to the queue, one calls `Push_entire_MsgQueue_into_OutgoingFifo ()` to flush the queue (i.e. empty it) and move its contents to an outgoing message FIFO buffer. Later, other code moves messages from this FIFO buffer out to the CANbus cable.

There are many different types of messages that one can transmit. To see these, search `":APPEND_MSG_"` and search `"int16 APPEND_MSG"`. In all cases, one appends messages to the message queue, and then flushes via `Push_entire_MsgQueue_into_OutgoingFifo ()`.

Message Callbacks

In some cases, one sends a message that later needs attention. For example, one might send a message that reads a field, where the data is received 10 to 100mSec later. The code does not sit and wait for the response, since most code only consumes 10 to 100uSec of microcontroller time and does not want to block other activity. Instead, the code releases the processor after sending the request, and sets up a callback routine that is called when the data is received.

When one appends the read request to the message queue, they also specify this callback function. Same with streams. When one ask for a stream (large # of bytes), they specify a function that is later called when the entire stream is been received.

To see an example of a read field callback, search `"PrepareCallback_ReadInt64_ClassWithCallbackMethod(this"`, and to see an example of a stream IO callback, search `"PrepareCallback_StreamIO_ClassWithCallbackMethod(this"`. The read INT64 callback calls method `ReadInt64_CallbackMethod ()` and the stream IO callback calls method `StreamIO_CallbackMethod ()`.

Message FIFO

Two FIFO (first in, first out) buffers are attached to each CANbus cable ("node" in XMC lingo). One FIFO for incoming messages, and one for outgoing messages. The message builder feeds the outgoing FIFO, and the CANbus receiver feeds the incoming FIFO. A

message processing function pulls messages out of the incoming FIFO every 10mSec and processes them via `EXECUTE_Foreground_Processes_Multiple_INCOMING_CanBus_Msgs()`. And a CANbus transmit function pulls messages out of the outgoing message FIFO and pushes them out the CANbus cable.

Chapter 8) Device Libraries

Device Summary Libraries

The DeviceSummary struct (BB_DeviceSummary_Timestamp) is ~90bytes long and contains approximately 40 parameters that summarize one device. These are packed into an array and maintained in a device library, which holds information on multiple devices, one DeviceSummary struct per device.

Each Network Device (brown in illustration) and Network Controller (orange) maintain a library of device summaries one level below. For example, a Network Device with five subnetwork Devices below it will maintain a library of five DeviceSummary structs, one per below device. All devices that have devices below them in the hierarchy, illustrated above, maintain these "1xLevelDown" libraries. For details, search "deviceSummary_1xLevelDown_LibraryP".

Each Network Controller (orange) maintains a library of device summaries that contains all devices one and two levels below (deviceSummary_Level234_LibraryP). This includes Devices (brown) and Subnetwork devices (blue).

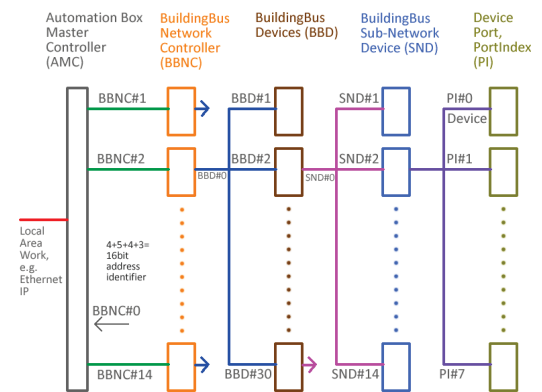
Each Master Controller (AMC) maintains a library of device summaries of all devices in the system (deviceSummary_Level_1234_LibraryP). This includes all Network Controllers, all Devices, and all Subnetwork devices.

Pointers to these libraries are members of the BB_EntireDevice class, therefore devices can easily access them.

Libraries are stored in device flash memory; therefore, a device does not need to rebuild from scratch each time it boots up. Instead, devices update existing libraries when they boot, and from time to time.

Libraries are maintained by the BB_LibraryManager class. A pointer to an instance of this class is a member of the BB_EntireDevice class; and is therefore easily accessible (libraryManagerP).

Libraries help devices understand their neighborhood.



Library

- > BB_Library_DataComponents.h
- > BB_Library_Documentation.h
- > BB_Library_Generic.c
- > BB_Library_Interface.h
- > BB_Library_Manager.cpp
- > BB_Library_Manager.h
- > BB_Library_NetworkMessages.h
- > BB_Library_Uilities.c

Building Libraries from Scratch

When a device first boots up, it builds and/or updates its libraries. This might take 1 to 100 seconds, depending on the situation. Devices gather information using a state machine that executes one state every 100mSec. Each state performs a different function. For example one state sends a message requesting a stream containing a device summary, another state checks if the stream is finished, and a third state processes the received data when complete. If it takes 1 second to receive the stream, the state machine would use very little processor time during that 1 second, since each state performs a task and then returns control back to the processor. Over time, it gets things done.

```
└─ StateMachine
  > BB_BASE_StateMachine.cpp
  > BB_BASE_StateMachine.h
  > BB_DeviceCtrl_StateMachine.cpp
  > BB_DeviceCtrl_StateMachine.h
```

An instance of the BB_DeviceCtrl_Manager class manages devices one level below (e.g. a set of subnetwork devices under a device). The DeviceCtrl manager creates an instance of the

```
└─ DeviceController
  > BB_DeviceCtrl_Interface.h
  > BB_DeviceCtrl_Manager.cpp
  > BB_OneDevice1xLowerThanMe.cpp
```

BB_OneDevice1xLowerThanMe class for each below

device, and then is considered to be their "controller". Controllers collect device summaries from their below devices, and place these into a library maintained by the library manager. For details, search "deviceSummary_1xLevelDown_LibraryP".

Three types of libraries

BuildingBus supports three types of libraries:

- AllDevices1xLevelDown: Contains one entry for each lower device (e.g. all subnetwork devices under one device).
- MultipleDevices: Contains one entry per device, supporting any number of devices, in any order, from any defining set (e.g. all light sockets in living room).
- OneDeviceMultipleDataCapsules: Contains copies of data capsules (sets of fields) from one device.

Locating Information in a Library

One does not need to know how libraries are built, or how they are packed, in order to use them.

To find a device in a library, call FindDeviceWithin_BbLibrary_NoError ().

If a library is of type `AllDevices1xLevelDown`, one identifies a device with `addressIndex_1xLevelDown_base0`, which is the address of the lower device (e.g. subnetwork # when working with a set of subnetwork devices).

If a library is of type `MultipleDevices`, one identifies a device with its network address (i.e. network #, device #, subnetwork #).

If a library is of type `OneDeviceMultipleDataCapsules`, one identifies an entry (i.e. one data capsule) with its `PortIndex` (e.g. 0...7) and `BB_CapsuleType` (e.g. `ImmutableData`, `ImmutableStrings`, `PortRegisters`).

Library Element Index

Each entry in a library (e.g. device summary or data capsule) is identified with an index into the library. This index is referred to as "elementIndex_base1". If

`FindDeviceWithin_BbLibrary_NoError ()` finds what you are looking for, it will return to you the `elementIndex_base1` for that item. Then, you pass that to `CALCULATE_DataElementPosition_GIVEN_ElementIndexBase1 ()`, which returns a pointer to your data (`dataElementPosition.dataElementPtr`).

Iterating Through Libraries

To iterate through a library, one can scan `elementIndex_base1` from 1 to the # of entries in the library. For an example of this, search "for (elementIndex". For each `elementIndex`, one can call `CALCULATE_DataElementPosition_GIVEN_ElementIndexBase1 ()` to get a pointer to the data. This routine loads a `BB_LibraryElementIdentifier` struct, which contains a pointer to the data.

Viewing Internal Library Structure with the Debugger

To view internal library information with the debugger, call `ViewLibraryIndicesWithDebugger ()` and place a break point at the end of this routine. One might need to `#define VIEW_WITH_DEBUGGER_BuildingBus_Library` in order to enable this routine.

Device Summary

For details on the device summary struct, search "struct `BB_DeviceSummary_Timestamp`". This struct contains other structs and one can view them to learn more. For details, see:

- struct `BB_Device_SUMMARY_Immutable`
- struct `BB_Device_STATUS_Volatile`

- struct BB_Device_LOCATION_Volatile
- struct BB_Device_DISCOVERY_1xLevelDown_Volatile
- struct BB_Device_ACCUMULATION_AllLevelDown_Volatile

The word "volatile" is appended to struct names to indicate they are copies of data that changes. Subsequently, its data might not be current when one uses it.

The word "timestamp" is appended to struct names to indicate that the collection date/time is included in the struct. If it was collected a long time ago, it is more likely to be stale (i.e. not match data in actual device).

The Device_SUMMARY_Immutable struct contains immutable data (never changes) such as device type, product type, vender ID, and device serial number. The Device_LOCATION struct contains physical location information, such as "living room" and "room 103" within a building. These two structs help one locate devices of interest.

Routing of CANbus Messages through the System

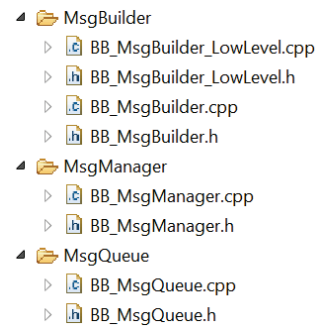
Many devices have two physical CANbus cables, and therefore act as a bridge when moving messages through the system (message passes from one CANbus wire to the other). For details on how a device figures out what to do after it receives a message (i.e. process, ignore or bridge), see file `BB_EntireDevice_NetworkRouter.cpp`.

Working with Messages

The `BB_MsgBuilder` class helps to create network messages, and the `BB_MsgQueue` class maintains a list of messages. However, in many cases, one interacts at a higher level and instead calls an `APPEND_MSG` routine, which makes use of these two classes. For details, search `":APPEND_MSG"`.

C functions in files `BB_CANbus_FramePacking.c/.h` help to pack and unpack messages, yet these can be ignored as well since higher level code is easier to work with.

For details on how messages are packed, see "Moving Data from One Device to Any Other Device" within [this](#) document.



CANbus Communication

Each device attaches to one or more CANbus cables, and each cable is managed by an instance of the `BB_CANbusCableInterface` class. For example, a window controller attaches to other window controllers with one cable, and attaches to its subnetwork devices (e.g. motors in window) with another. The cable that routes toward the master controller is referred to as "upstream" (cable index #0) and the one that routes toward subnetwork devices is referred to as "downstream" (cable index ≥ 1).

Each CANbus cable has both a receive FIFO buffer and a transmit FIFO buffer (class `BB_FifoBuffer_MsgPacket`). When a device transmits data, it pushes a message packet out the transmit buffer (`CANbusMsgPacket_Fifo_OUTGOING_MessagesP`), and when it receives a message, it gets the data from a receive buffer (`CANbusMsgPacket_Fifo_INCOMING_MessagesP`).

Every 10mSec, the device receives an interrupt and checks the receive buffers for new incoming messages. If it sees them, it processes them via method `Process_Multiple_CANbusMessages()`.

Messages Packets

Each message consists of a 4 byte CANbus identifier and an 8 byte data payload. All 12 bytes are transmitted from an initiator device to a target device. C/C++ code maintains these 12 bytes in a BB_CANbusMsgPacket struct. After receiving a packet, a device interrogates it via function UNPACK_CANbusMessage () and places the results of the interrogation into struct BB_CANbusMsgDescriptor. For details on what the code sees after the message is unpacked, search "struct BB_CANbusMsgDescriptor".

CANbus Hardware Interface

Several classes interface the outgoing/incoming FIFO buffers to the CANbus hardware. Class BB_XMC_CANbus_System manages a set of CANbus cables, class BB_XMC_CANbus_OneNode manages one cable, and class BB_XMC_CANbus_OnePipe manages one FIFO attached to a cable. Method TryTo_Transmit_One_CANbus_MsgPacket () pushes one data packet into the CANbus transmit hardware. More specifically, it tries to do this. If the hardware is busy, it returns quickly without success. When a data packet is received, an interrupt service routine calls Receive_One_CANbus_MsgPacket (), which reads the data and pushes it into the incoming FIFO buffer.

One Main Thread plus ISR's

The main thread does 99% of the work. Also, interrupt service routines (ISR's) can interrupt this main thread at any time. Several ISR's relate to CANbus and they are synchronized with the main thread via FIFO buffers. For example, the receive ISR (ISR_CAN_x_Rcv ()) pushes a message into a FIFO while the main thread later pulls it out. The FIFO's are designed to allow one thread to push while the other pops. This is a bit tricky, since an interrupt could occur while the main thread is interacting with a FIFO, yet the code expects this and is designed to support this.

Mutex & Critical Sections

In some of our code, we have [mutexes](#) (only one thread interacts with a resource at a time) and critical sections (interrupts turned off for a short duration). To see these, search "K_MUTEX" and "__disable_irq ()".

CANbus Status

One can get status information on a CANbus node (cable) by calling GetStatus_CANbusNode_AlreadyLocked (). This reads the hardware status register, evaluates it via Calculate_nodeStatusBits (), and places the evaluation results in the

"nodeStatusBits" struct. If it sees a problem, it increments a counter associated with that type of problem via `Increment_errorCounters ()`.

CANbus Bus-Off

If no other devices are attached to the CANbus cable, an acknowledgement is not seen by the transmitting device and this is considered a CANbus error condition. After multiple attempts, the transmitting device enters a "Bus-Off" state, which means it can no longer transmit. When this occurs, we set our

thisNodeIsInBusOffStateProbablyDueToNoDeviceOnCableToAcknowledgeMsg flag. Every ~10 seconds after setting this flag, we reset CANbus via `IfCurrentlyInBusOffStateThen_Reset_CANbus ()`, which enables us to check if we can transmit (i.e. we check if someone attached a device to the cable within the previous 10seconds). For details on this, search this file for "Bus-Off".

Print to Console and Receive CANbus Message Do Not Mix

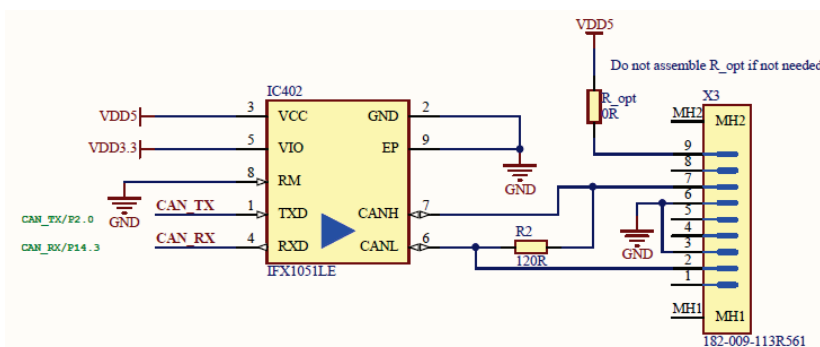
The XMC print to console pane turns off interrupts for ~10mSec while printing. This can cause one to miss CANbus messages since one might receive two or more CANbus interrupts during this 10mSec. When this occurs, only one is serviced. There are several Debug Configurations -- those labeled "FreeRun" do not print and do not hamper CANbus, and those labeled "DavelDE" do. For details, search this file for "Debug Configuration".

Monitoring Processor Activity with an Oscilloscope via ChZ

The DavePrj_BB project is set up to control a digital output bit that is connected to an oscilloscope (DIO_Bit_ChZ). One sets ChZ_CONTROL to determine when this bit goes up and when it goes down. For example, set it to `CONTROL_ChZ_ISR_Rcv_Canbus_Node_x` to set this bit high while in the CANbus receive ISR. Alternatively, set it to `"CONTROL_ChZ_MasterIdleChore"` to monitor the 10mSec main thread activity. For a list of options, search "Digital I/O ChZ".

Monitoring CANbus with an Oscilloscope

One can connect the CANbus Transmit (TX) signal to an oscilloscope to see when a device is transmitting. Alternatively, one can monitor the CANbus Hi signal to see CANbus activity on a cable, yet in this case, one does not know who is transmitting.



CANbus Testing

For details on debugging CANbus, search this document for "CANbus Testing".

Overview

In many cases, one can interact with devices via Read/Write field routines. For example, one device can read the sensors in another device by reading the Sensor_Measurement_Channel field within that device. Or control the position of a motor or LED light illumination by writing to a device's ControlOutput_CommandOutput field.

Devices can work with larger blocks of data via the Get Stream support, described previously.

However, there is one more way to interact with devices, which is through a function handler. In this case, one device calls a function within another device. The initiator passes parameters to the target, the target does something, and the target optionally sends data back to the initiator. In the underlying code, both read/write field and read stream are implemented with function handlers. To see these, search "HANDLER__DeviceCommon_RdWrField" or search "HANDLER__DeviceCommon_StreamIO".

A 5-bit Major Function Code (0...31) in the CANbus 29bit identifier is transmitted along with the 8byte data packet and this specifies which handler receives the incoming message. For example, if set to 1, the message is passed to the stream IO handler; and if set to 0, the message is passed to the read/write field handler.

Dispatching to a Handler

Recall that CANBus messages contain a 0...7 portIndex, and are therefore directed to a port once received. From there, they are dispatched to one of several major function handlers, as determined by the MajorFunctionCode (0...31). Handler dispatching for the DeviceCommon class at portIndex #0 is shown here. To see all handlers, search "HANDLER__".

```
switch (received_CANbusMsgP->msgDescriptor.majorFunctionCode) {  
  
    case DeviceCommon_port_MajorFunctionCode_0_RdWrField: // Read/Write  
                                                           // For detail:  
        errorCode = HANDLER__DeviceCommon_RdWrField( received_CANbusMsgP );  
        break;  
  
    case DeviceCommon_port_MajorFunctionCode_1_StreamIO: // Read/Write  
                                                         // For detail:  
        errorCode = HANDLER__DeviceCommon_StreamIO( received_CANbusMsgP );  
        break;  
  
    case DeviceCommon_port_MajorFunctionCode_2_TestSuite: // DeviceCommon  
                                                           // For detail:  
        errorCode = HANDLER__DeviceCommon_TestSuite( received_CANbusMsgP );  
        break;  
  
    case DeviceCommon_port_MajorFunctionCode_3_DeviceManager: // DeviceCommon  
                                                                // For detail:  
        errorCode = HANDLER__DeviceCommon_DeviceManager( received_CANbusMsgP );  
        break;  
  
    case DeviceCommon_port_MajorFunctionCode_4_InfoOnLowerDevices: // Handle  
                                                                    // For detail:  
        errorCode = HANDLER__DeviceCommon_ReceiveDeviceInfoFromLowerDevices( received_CANbusMsgP );  
        break;  
}
```

Handlers Digest Information and Respond

Handlers receive messages in the form of a `BB_CANbusMsg_FullInformation` struct, which is an unpacked version of the message. This means that parameters coded into the message have been extracted and placed into easy-to-read fields within this struct. For details, search "`struct BB_CANbusMsg_FullInformation`" and search "`struct BB_CANbusMsgDescriptor`".

The handler might digest further. For example, the read/write field handler uses `Unpack_RdWrField_MsgPacket ()` to extract parameters packed into the 8byte data payload, and places them into a `ReadWriteFieldOperation` struct. For details, search "`struct ReadWriteFieldOperation`". After unpacking, the handler implements the required function.

We refer to the 0...31 function code as "major" since each handler provides support for additional functions as determined by parameters in its 8byte data payload. For example, the read/write field handler enables one to read, or write, as determined by a bit within the data payload. And `HANDLER__DeviceCommon_StreamIO ()` returns a block of data, where the specific block is determined by parameters in the data payload (e.g. data capsule, library of device summaries, or multiple fields).

Conclusion

At a low level, devices interact with each other via read/write field, get stream, and function handlers.

Overview

In BuildingBus lingo a "machine" is a class that contains the following features:

- Executes periodically (e.g. once a second)
- Is controlled with a C struct that contains control parameters
- Maintains data parameters in a C struct

Machines optionally allow other network devices to adjust their control parameters and/or read their data parameters.

Execution Rate

The main thread executes EXECUTE_MasterIdleChore () once every 10mSec. This calls Calculate_Execution_Anniversaries (), which calculates Boolean variables that are set true at fixed rates using a hardware-based counter. For example, executionRate.executing_approximately_once_every_100mSec is set true once every 100mSec. The executionRate struct maintains 11 different times, as shown to the right.

```
bool executing_approximately_once_every_10mSec;  
bool executing_approximately_once_every_30mSec;  
bool executing_approximately_once_every_100mSec;  
bool executing_approximately_once_every_300mSec;  
bool executing_approximately_once_every_1_second;  
bool executing_approximately_once_every_3_second;  
bool executing_approximately_once_every_10_second;  
bool executing_approximately_once_every_30_second;  
bool executing_approximately_once_every_100_second;  
bool executing_approximately_once_every_300_second;  
bool executing_approximately_once_every_1000_second;
```

Machines use 'executionRate' to execute their idle chores (i.e. processing functions) at fixed intervals. For example, within EXECUTE_MY_Device_Child_IdleChore (), one can place the following code to execute at a fixed rate:

```
If (executionRate.executing_approximately_once_every_...) {  
    ... do processing  
}
```

Execution Rates are not Accurate

ExecuteRates are not accurate since one might miss a 10mSec interrupt if another subroutine consumed more processor time than expected. If you are pacing yourself at 100mSec, and another routine consumes the microcontroller for 1 second, for example, then your 100mSec task would be given time at the next opportunity. After that, it would wait another 100mSec before it executed again. In this example, 9 execution times are missed.

Even if other routines behave, the system might not calculate 100mSec exactly, and instead might provide an approximate rate accurate to +/-30%.

If you want accurate time, read the 64bit 1uSec hardware counter (i.e. call `GET_DateTime_1uSec_Units_1Jan2020_int64 ()`).

Machines are Smart

BuildingBus machines are where one might implement so called "intelligence". For example, let's assume we have a window thermal cover that closes when all of the following conditions are met: room is not occupied, we are not trying to heat the floor with sun while the sun is shining and room is colder than desired temperature, and we are not trying to cool the room when outside air is colder than inside air. To implement this, a BuildingBus machine in the motor device (which controls the thermal cover position) might interact with sensors to perform this logic.

We do not need to be Too Smart

Our machines do not need to be very smart or very complicated. We are not playing chess. Instead, we need to be reliable, long lasting, low cost, easy to maintain, and easy to set up. In a sense, we need to do the simple things well.

Overview

An example BuildingBus machine is the BB_TestSystemA class. This is executed every ~300mSec via Execute_TestSystemA_IdleChore () and tests multiple devices in a system. One can use this class to test multiple Platform2Go boards CANbussed together with real physical wire. This machine is controlled with its 32bit BB_TestSystemA_Control struct and maintains state and data in its BB_TestSystemA_Data struct. Other devices can write to this control struct since it sits in a 32bit integer field within the Device Common Port Registers Capsule. In other words, other devices can set this register via APPEND_MSG__WriteInt64_DeviceCommon_PortRegister (). Also, other devices can read its data via stream IO. For an example, search "Read struct BB_TestSystemA_Data in target test".

Read/Write TestRegister16

This machine does testing with a target device whose address is specified in the 32bit control struct (target_address_uint16). One test involves writing to the target device's TestRegister16, which is a 16bit register in the Device Common Port Register capsule. After writing value X, the register is read back and the callback routine looks for X. If it passes, a counter is incremented, and if it fails, a different counter is increment. All counters are maintain in the result data struct.

Read Streams

The machine reads streams periodically from the target device and checks if they are received and contain a key field in the last byte.

Wait for Response before Testing

The machine continuously reads from the target test device and does not begin testing until a response is received. This is helpful in cases where the target is not set up when the test begins. After receiving a response, it waits an additional 3 seconds since messages might still be in transit.

The Primary Device Controls the Test Target Device's Control Struct

One of the devices in the system is considered the "Primary" device while the others are considered to the "Test" devices. The primary writes to the TestSystem Control struct in the Test device and tells it to do testing with the primary as its target. We end up with multiple devices simultaneously performing tests on each other.

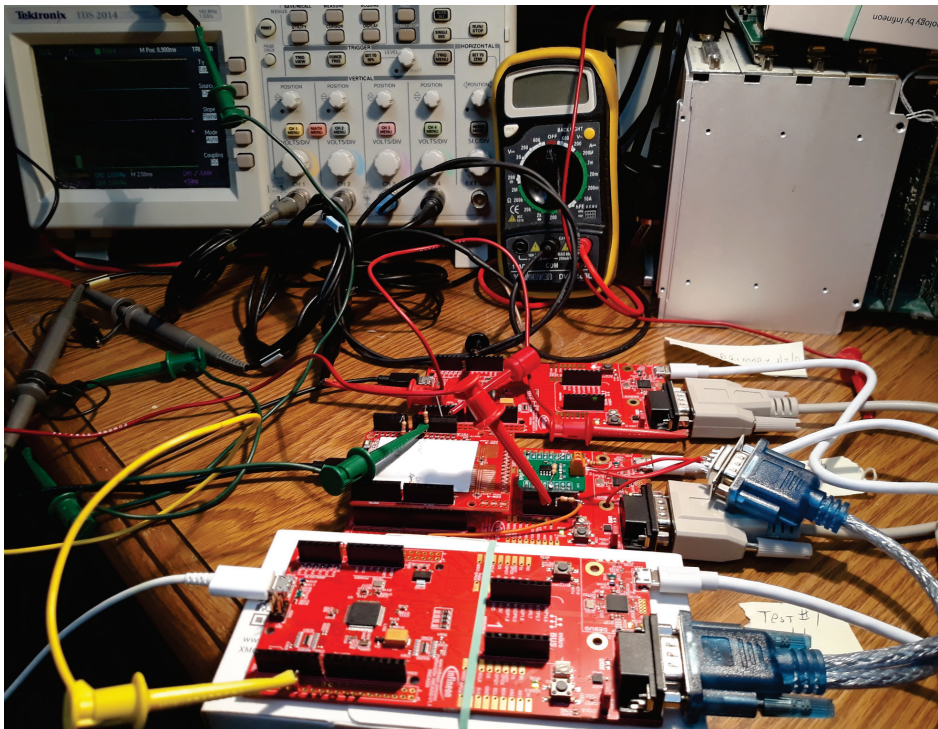
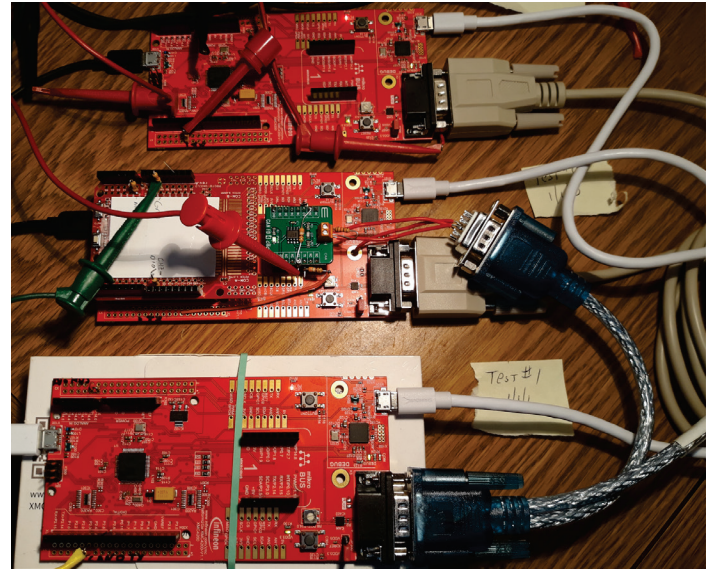
The primary device periodically reads the test device's TestSystem data struct to see how it is doing and stores this in class member "otherDevice_TestSystemA_Data".

Be careful with printing to the console pane. This can turn off interrupts and cause us to miss data packets, which show up as errors in the result data structs.

It is helpful to press the Reset button (near DB9 connector) on each Test Device before the Primary Device begins, so that they start from a known state.

Overview

Pictured here are multiple Platform2Go boards cabled together via CANbus. In this setup, we move millions of messages over 24 hours without error. This includes bridging messages from one CANbus cable to the other (i.e. through a device); reading and writing test registers, reading data streams, and moving data simultaneously between multiple devices. For details on the Mikroe CANbus Node #1 PCB that supports downstream networks (toward subnetwork devices), search this file for "NCV7344".



Addressing, Cabling, Connectors, Devices and TestSystem Instances

In the typical case, the Primary Device is at address 1/2/0 (network #, device #, subnetwork #), Test Device #0 is at address 1/1/0, and Test Device #1 is at address 1/1/1. To see where addresses are set, search project for "Define Network Address for PrimaryDevice".

One downloads FreeRun images into the Test Device(s) and disconnects the USB debug cable after each download. While testing, the Primary Device is attached to the computer via the USB debug cable while running the "Debug_DaveIDE" or "Debug_FreeRun" image. The "DaveIDE" image prints yet also incurs occasional CANbus errors due to the CANbus/printing critical section conflict described previously. The "FreeRun" image does not print and can run overnight w/o error. For details on managing and creating images, search this document for "Build Configuration".

Devices 1/1/0 and 1/2/0 attach to each other via their DB9 connectors (upstream cable). Device 1/1/0 and 1/1/1 attach to each other via a Mikroe board on the 1/1/0 device (downstream cable) and the DB9 on the 1/1/1 device (upstream cable).

One can purchase DB9 female-to-female pin N-to-N cables from Amazon ([cable](#)) or Digikey ([cable](#)). Also, one can purchase from Amazon DB9 Female-Female-Male [Y-Cables](#), DB9-Female [breakout](#) connectors, DB9 [Connectors](#), and [330ohm](#) protection resistors.

The Platform2Go DB9 connector and the Mikroe CANbus Click boards already have 120ohm terminations resistors built-in, therefore one does not need to add these.

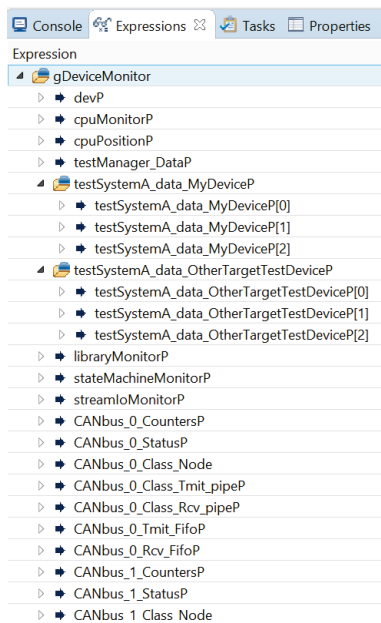
When an image begins, it needs to decide how many instances of the TestSystem class to create and run, and this is determined by define symbol

"NUM_OF_TestSystemA_TO_SET_UP_AND_START_RUNNING". Typically, the Primary device sets this to 3 and Test Devices set it to 0 (since Primary Device controls TestSystem class on Test Devices).

TestSystem Instance #N does testing with Test Device #N (e.g. TestSystem Instance #0 does testing between Primary Device and Test Device #0). If your Test Device is not physically connected, then the Primary will send messages to an address with no response. This does not do harm, and will show up in the result data as a non-responsive device.

gDeviceMonitor

The global gDeviceMonitor struct contains pointers to important structs, as shown below-left. One can place this into the Expression pane and view key information after stopping at a break point. Below-right shows TestSystem result data with no error after running 24 hours (this is real data).



testSystemA_data_MyDeviceP	0x1fffd0
testSystemA_data_MyDeviceP[0]	0x20002
TestRegister16_WelInitiated_numOfMsgs_Write_uint32	235613
TestRegister16_WelInitiated_numOfMsgs_Read_uint32	235613
TestRegister16_ResponseFromOurInitiation_numOfMsgs_Received_uint32	235614
TestRegister16_ResponseFromOurInitiation_numOfMsgs_MsgSessionID_match_FAIL_uint16	0
TestRegister16_ResponseFromOurInitiation_numOfMsgs_Token32_match_FAIL_uint16	0
TestRegister16_ResponseFromOurInitiation_numOfMsgs_Token32_match_OK_uint32	235614
TestRegister16_ResponseFromOurInitiation_numOfMsgs_ErrorCodes_uint16	0
TestRegister16_ResponseFromOurInitiation_numOf_Pings_uint16	0
TestSystemA_Streams_weAreClient_numRequested_uint32	32725
TestSystemA_Streams_weAreClient_numReceived_uint32	32725
TestSystemA_Streams_weAreClient_numFailedKeyField_uint16	0
OtherDevice_TestRegister16_ReadMsgs_match_OK_uint32	235585
OtherDevice_TestRegister16_ReadMsgs_match_FAIL_uint16	0
OtherDevice_TestRegister16_Reading_32100_Ping_uint16	1
OtherDevice_TestRegister16_Writing_32100_Ping_uint16	1
targetTestDeviceTestSystemAControlRegister_numOfTimesWeTurnedOn_uint16	1

Multiple Images

The DavePrj_BB project can be set up to build images for a Primary Device, Test Device #0, Test Device #1, and Test Device #2. Build Configurations and Debug Configurations are associated with these different images. Preprocessor symbols in a Build Configuration enables the compiling of a specific image (e.g. `ENABLE_BuildImage_TEST_TARGET_DEVICE_0` builds Test Device #0). All compiler switches are maintained in one file (`MY_Project_Compiler_Switches.h`).

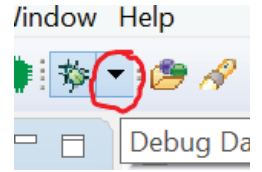
Multiple Instances of TestSystem Class

BuildingBus includes a `BB_TestManager` class that maintains multiple instances of the `TestSystem` class. Subsequently, one instance can be set up to do testing with one target device while another instance with a different target. As shown to the right, we have 3 instances in `gDeviceMonitor` where each exposes the result data from its own tests, along with the result data from test target devices. In other words, each test target device has its own result data and it is read by the Primary device every 10seconds. The 32bit control word for each of these 3 instances is maintained in a 3-element array within the `TestSystemA` field of the Port Register capsule (`dcrFI_TestSystemA_x_Control_int32`).

testSystemA_data_MyDeviceP
testSystemA_data_MyDeviceP[0]
testSystemA_data_MyDeviceP[1]
testSystemA_data_MyDeviceP[2]
testSystemA_data_OtherTargetTestDeviceP
testSystemA_data_OtherTargetTestDeviceP[0]
testSystemA_data_OtherTargetTestDeviceP[1]
testSystemA_data_OtherTargetTestDeviceP[2]

Working with Configurations


To set the Active Configuration, right click on project name, select Build Configurations, and select Set Active. To debug using a specific Debug Configuration, click down arrow next to Debug button, picture to the right.




When working with the Primary device w/o printing, set Active Configuration to:

▲  **DavePrj_BB [Active - Debug_FreeRun_PrimaryDevice]**


and use this Debug Configuration:

 Debug_FreeRun_PrimaryDevice

When downloading an image into Test Device #0, set Active Configuration to:

▲  **DavePrj_BB [Active - Debug_FreeRun_TestTarget_0]**

and use this Debug Configuration:

 Debug_FreeRun_TestTarget_0

Testing Comments

Below are several comments on testing.

- Blinking of LED at a 1Hz rate means the software is running ok and has not crashed. You want to see all your devices blinking while testing.
- Press Reset Button on Test Devices before you begin, so they start from a known state.
- Place a breakpoint at LogErrorCode () and look for errors (search "gPutDebuggerBreakpointHere_LogErrorCode"). If a message packet data is wrong, you will not enter LogErrorCode and instead an error counter will increment in the result data struct. LogErrorCode is for more serious errors.
- The Platform2Go board has 2 USB connectors. The board receives power from either. One can attach to both at the same time (diodes block contention).
- CANbus enters a suspended Bus Off state if there is no device on the cable to acknowledge a transmitted message packet. To see this state being entered, place a break point at: "thisNodeIsInBusOffStateProbablyDueToNoDeviceOnCableToAcknowledgeMsg = true". If this occurs, the device will try to recover from this state every 10 seconds. Adding a working device to the cable is needed for a device to recover from this state.

- After the bus-off state has been entered, all transmissions are deferred (do not occur) due to the following code. To see this, place a break point at this location:

```
if ( canbus_nodeP->thisNodeIsInBusOffStateProbablyDueToNoDeviceOnCableToAcknowledgeMsg ) {
    return BbError CANbus CannotTransmitBecauseBusIsInTheBusOffState;
}
```

- To see 10mSec master idle chore routine (main routine that runs every 10mSec), place a break point at ":EXECUTE_MasterIdleChore(".
- To see CANbus receive a data packet, place a break point at ":Receive_One_CANbus_MsgPacket".
- To see CANbus transmit a data packet, place a break point at ":TryTo_Transmit_One_CANbus_MsgPacket".
- One can press RESET button on the Platform2Go to reset PCB and have code start at the beginning of main. This will also pull the board out of a Bus-Off state, if present. If another device is not on the cable, it will re-enter Bus-off after first transmission.
- One can place an oscilloscope on DB9 connector pin 7 (CANBus hi) to monitor CANbus activity initiated by multiple devices (you do not know who is transmitting).
- One can place oscilloscope on CAN_TX (Platform2Go X2 CAN_TX/P2.0 is also routed to X1 connector pin 29) to see a specific Platform2Go board transmit data. If you monitor the CAN_TX line of two PCB's, you can see which one is transmitting. CAN_TX is a signal between the microcontroller and the CANbus transceiver IC. CAN_TX shows a microcontroller IC transmitting, and does not reflect activity on the bus due to other microcontrollers transmitting.
- Controllers sometimes output data frames on the CANbus continuously, as quickly as possible, trying to recover from an error (e.g. every 1mSec). This is seen by the software as a "Bus-Off" state.
- If the oscilloscope shows CANbus activity on CANbus_HI, and you see one pulse on CAN_TX at the end of each message, then that pulse is acknowledging packets transmitted by a different PCB.

Debugging CANbus

To debug CANbus, place 'gMonitorCanbusP' into the Expression pane. This contains much information about the 0th CANbus node (upstream cable). The 'thisNodeIsInBusOffState...' parameter will be set if the node is in the Bus-Off state, probably due to no device on the cable providing an acknowledgement. If one does enter Bus-Off, the system will try to clear the condition every 10seconds.

The following are sometimes helpful after being added to the Expression pane. They all relate to the 0th CANbus node (upstream cable headed toward AMC); however, one could replace gCanBusNodesP [0] with gCanBusNodesP [1] to see 1st node.

- "gCanBusNodesP[0]": 0th CANbus node. Open "monitor" and "nodeStatusBits" for more information.
- "gCanBusNodesP[0]->rcv_pipeP" - pipe used to receive CANbus messages.
- "gCanBusNodesP[0]->rcv_pipeP->FifoBuffer_MsgPacketP": fifo buffer used to receive messages from CANbus cable.
- "gCanBusNodesP[0]->tmit_pipeP": system that transmits CANbus messages.
- "gCanBusNodesP[0]->tmit_pipeP->FifoBuffer_MsgPacketP": fifo buffer used to receive CANbus messages.

gMonitorCanbusP	0x200
cpuMonitorP	0x1fff
errorCounters	{...}
u	{...}
struct_	{...}
tmitFailedUpdate	0
tmitFailedTransmission	0
tmitTimeout	0
stillTransmitting	0
rcvFailed	0
NodeSuspended_DueTo_BadErr_Or	0
overflowed_rcv_fifoBuffer_AndOver	0
LEC_ack	0
EWRN	0
BOFF	0
LLE	0
LOE	0
SusAck	0
padding	0
array_	0x200
numOfTimesWeClearedBusOffState_uint16	0
numOfTimesWeEnteredBusOffState_uint16	0
node_status_LastReading_uint16	0
ISR_counter_TransferOk_uint16	0
ISR_counter_FrameCounterIsr_uint16	0
ISR_counter_LastErrorCode_uint16	0
ISR_counter_NodeAlert_uint16	0
ISR_counter_RcvFrame_OK_uint16	0
ISR_counter_TmitFrame_OK_uint16	0
thisNodeIsInBusOffStateProbablyDueToNoDc	0 '\0'

Review

As noted previously:

- "BB" files do much of the work and are not changed by Industry Programmers.
- Industry Programmers create products.
- Industry Programmers place their code in "MY" files.
- MY files contain children of classes defined in BB files.
- All source code and reference designs are free and open to encourage adoption, to reduce CO₂ emissions. Anyone can copy and modify at no charge.

One Device

Pictured to the right are MY files that create a VariableControl device. This is a device that is controlled with an int16 value. This supports: on/off motors, variable speed motors, 0 to 100% position stepper motors, on/off lights, 0 to 100% illumination lights, fans in ducts, and 0 to 100% open dampers in ducts and at vent openings.

Three Ports

This device has three ports: DeviceCommon, MeasurementSys, and VariableControl. DeviceCommon relates to the entire device, MeasurementSys measures sensors, and VariableControl implements int16 control.

```

└─ MY_Dev555_VariableControl
  └─ MY_Dev555_port_DeviceCommon
    > MY_Dev555_port_DeviceCommon_setup_ncr.cpp
    > MY_Dev555_port_DeviceCommon.cpp
    > MY_Dev555_port_DeviceCommon.h
  └─ MY_Dev555_port_MeasurementSys
    > MY_Dev555_port_MeasurementSys_setup_ncr.cpp
    > MY_Dev555_port_MeasurementSys.cpp
    > MY_Dev555_port_MeasurementSys.h
  └─ MY_Dev555_port_VariableControl
    > MY_Dev555_port_VariableControl_setup_ncr.cpp
    > MY_Dev555_port_VariableControl.cpp
    > MY_Dev555_port_VariableControl.h
  > MY_Dev555_BuildOneDevice.cpp
  > MY_Dev555_EntireDevice.cpp
  > MY_Dev555_EntireDevice.h
  > MY_Dev555_Interface.h
  > MY_Dev555_Master_Include.h
```

A folder is dedicated to each port and in this folder we place a setup_ncr.cpp file, port.cpp file, and port.h file.

Setup_ncr.cpp sets up the port and is not changed by the Industry Programmer. Instead, we let the compiler build internal capsules (sets of fields) using this file. The filename contains "ncr", which stands for "no change required".

The port.h file defines structs and enums used by the port class. In many cases, the industry programmer does touch these. An example is the definition of the MeasurementSys immutable data capsule (i.e. struct MY_Dev555_MeasurementSys_ImmutableData). This is influenced by the # of sensors in

the device, which is determined by the industry programmer. This number sets the size of arrays within the capsule, and since BB files are the same in all devices, we define this unique struct in the MY file.

Note to Programmer

MY files contain a note to Industry Programmers telling them what needs attention and what can be ignored, an example of which is shown below. Most MY code is marked "No change required" and can be ignored.

```
// =====  
// CAPSULE DEFINITION:      DEVICE TYPE HANDLER PORT: Me  
// INDUSTRY PROGRAMMER:    No change required.  
// =====
```

R/W Field Intervention

The port .cpp files contain class definitions for each port, followed by methods that operate on capsule fields before they are read from, or written to, by the network. Below is a summary of these methods:

- OpportunityToUpdate_StatusCode_BEFORE_ReadByNetwork (): Update status fields immediately before they are read by the network.
- RespondTo_PortRegister_ControlCommand (): Respond immediately after the network sets a control field.
- OpportunityToRespond_BEFORE_NetworkReadsOrWrites_OnePortRegister (): Respond to port register read/write field, *before* being read by the network.
- OpportunityToRespond_AFTER_NetworkReadsOrWrites_OnePortRegister (): Same as above, yet *after* the network reads or writes to or from a port register field.

These methods are in MY files, and the Industry Programmer updates as needed. An example is the reading of a sensor. When the network measures the Sensor_Measurement_Channel field, the Industry Programmer needs to make sure the A/D measures the sensor before the value is returned, and does this with code added to OpportunityToRespond_BEFORE...().

Defining Immutable String and Data Fields

The industry programmer is responsible for defining immutable data and string fields near the bottom of the port .cpp files (e.g. set vender name string to "Dell Corporation"). For details, search "DEFINE IMMUTABLE".

Capsule Internal Components

If one wants to learn more about the data capsule's internal components, search for the following text:

- "CAPSULE FIELD STRUCT": Structs that define components within capsules.
- "CAPSULE DEFINITION": Structs that define entire data capsules.
- "FIELD GROUP CALCULATION": Calculations of internal capsule size and field information, which is placed into a FieldGroupInfo_OneCapsule struct.
- "FIELD DESCRIPTOR CALCULATION": Calculations of internal field descriptions (i.e. 16bit value that defines field size and position of field within capsule).

Declare Supported Functions

One declares which Major Functions are supported by each port. For details, search "DECLARE SUPPORTED FUNCTIONS".

Port Class Constructor Methods

Constructor methods for each port class set up each port, and are often not changed significantly by the Industry Programmer.

MeasurementSys Sensor Measurement

The MeasurementSys port maintains sensors. Sensors are declared in an enum and have a big effect on data capsules since they determine the size of field arrays, shown below. For examples of sensors being declared, search "SENSOR CHANNEL DECLARATION".

```
MY_Dev555_port_MeasurementSys
  MY_Dev555_port_MeasurementSys_setup_ncr.cpp
  MY_Dev555_port_MeasurementSys.cpp
  MY_Dev555_port_MeasurementSys.h
```

```
// Sense input via -32500 to +32500 value. For details search "enu
// e.g. wall switch dimmer potentiometer measurement, 0 to 32500 v
MY_Dev555_FlexType /* int16 or flt32 */ Sensor_Measurement_Channel_int16_or_flt32
[ MY_Dev555_NUMBER_OF_MeasurementSys_Sensor_Measurement_Channels ];
```

Sensor values are maintained internally as 32bit floating point values, or 16bit integer, as determined by the FlexType_System definition. For details on how this works, search "PORT DATA TYPE DECLARATION".

Sensors are measured via the Measure_One_Analog_Input_Channel () method, which reads the A/D immediately before the network measures a sensor field. This method is typically called by base class (BB file) Measure_Sensor_BASE_Class (), which measures the

A/D, scales the value, and stores it, as needed. For details, search `":Measure_One_Analog_Input_Channel"` and search `":Measure_Sensor_BASE_Class"`.

Before sensor measurement occurs, the class constructor calls `Setup_Hardware_ADC_Measurement_System ()` to set up a/d channels, and calls `Setup_Measurement_Scaling_System ()` to set up scaling between A/D units (e.g. 0 ... 4095) and engineering units (e.g. -32500 to 32500 int16 corresponds to -325.0 to +325.0 °C). For details, please see those routines.

The `MeasurementSys ImmutableData` capsule contains fields that specify the minimum and maximum values returned by sensors; along with maximum accuracy and noise; and sensor type. For details, search `"enum BB_SensorType"`, and search `"struct MY_Dev555_MeasurementSys_ImmutableData_CHANNEL_FIELDS"`.

Device Type

Each device implements a specific device type, which determines how it interacts with the network. For a list of device types, search `"enum BB_DeviceType_int16"`.

Many devices are of `DeviceType VariableControl`, which means you control them with an int16 value. All of the following devices are variable control: on/off motors, variable speed motors, 0 to 100% position stepper motors, on/off lights, 0 to 100% illumination lights, fans in ducts, and 0 to 100% open dampers in ducts and at vent openings. These all place a child of the `BB_VariableControl_Class` at portIndex #1. This is similar to what we do with the `MeasurementSys` class and sensors, yet the data goes in the opposite direction (measure vs. control).





Product Type

Every product implements a specific product type, which is how the end user thinks about the device (e.g. "light socket", "window curtain motor", and "wall light switch"). For a list of these, search `"enum BB_ProductType_int16"`. Again, device type is how the network interacts with the device, and product type is what the end user thinks about the device.

Both product type and device type are specified in fields within the `DeviceCommon ImmutableData` capsule, at specific field indices. Also, these reside in the `Device Summary` structs, which are maintained throughout the network within device summary libraries (see `"productType_int16"` and `"deviceType_int16"`). Subsequently, each device has access to information on all other devices that includes for each: network address, device type, and product type (among other parameters).

VariableControl Port

Port #1 is the DeviceType handler port, which means that it contains a class that implements the DeviceType.

- ◀  MY_Dev555_port_VariableControl
 - >  MY_Dev555_port_VariableControl_setup_ncr.cpp
 - >  MY_Dev555_port_VariableControl.cpp
 - >  MY_Dev555_port_VariableControl.h

DeviceType Variable Control supports multiple output channels (e.g. several light bulbs, several motors), as defined in the MY VariableControl port.h file (search "OUTPUT CHANNEL DECLARATION"). For each output channel, one specifies a type (shown below). For a list of these, search "enum BB_VariableControlChannelType".

```
// Type of each output channel. See BB_VariableControlChannelType
// SingleValue or Array. If length of array is 1 then we use same
int16_t /* BB_VariableControlChannelType */ VariableControlChannelType_int16
[ MY_Dev555_VariableControl_NUMBER_OF_OUTPUT_CHANNELS ];
```

The MY VariableControl .h file defines the port's capsules, and needs little attention -- much of the source code is marked "No change required".

The MY VariableControl setup_ncr.cpp file compiles internal structures and is exclusively ncr (no change required).

The MY VariableControl port class .cpp file (e.g. MY_Dev555_port_VariableControl.cpp) contains methods that allow one to intervene before, and after, fields are read from, or written to, by the network. Also, it is here that one defines immutable data and immutable string fields.






The industry programmer updates the Update_One_Control_Output_Channel () method, as needed, to implement control.

MY Port Setup NCR .cpp Files

The MY port setup.cpp files are marked "No change required", and can therefore be ignored. These files define the internal structure of data capsules. This includes defining the 16bit integer that pertains to each field and is embedded into each capsule (search "FIELD DESCRIPTOR CALCULATION"), and defining the size of capsule components (search "FIELD GROUP CALCULATION"). Also, this file provides routines that help build capsules, and check their internal data structure after built.

Entire Device

Several MY files, shown to the right, create the entire device. Much of the work is done by a child of the BB_EntireDevice class (e.g. MY_Dev555_EntireDevice), which creates and maintains ports. For an example, search "MY_Dev555_EntireDevice".

- >  MY_Dev555_BuildOneDevice.cpp
- >  MY_Dev555_EntireDevice.cpp
- >  MY_Dev555_EntireDevice.h
- >  MY_Dev555_Interface.h
- >  MY_Dev555_Master_Include.h

The MY BuildOneDevice.cpp file contains functions (not class methods) that build entire devices. These create instances of entire device class and sets them up with different network address, product types and device types; as desired.

Conclusion

The BuildingBus framework enables the world to develop smarter and more reliable devices with relatively little effort by industry engineers.

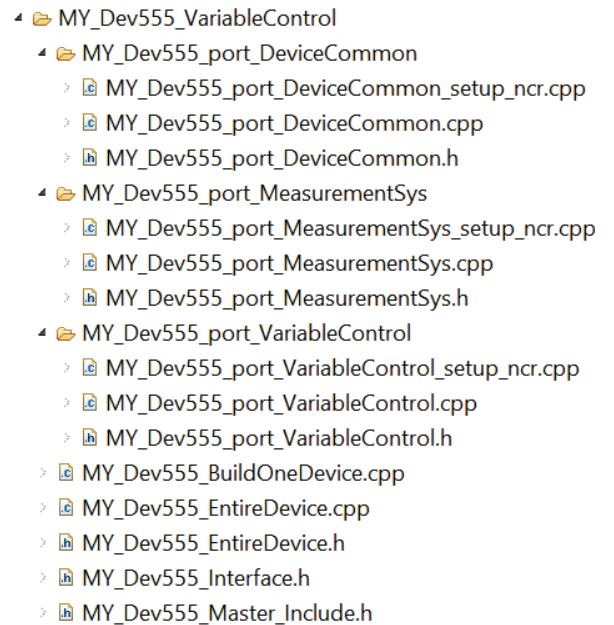
Overview

To create a new device, one can begin with an existing similar device, copy, and modify.

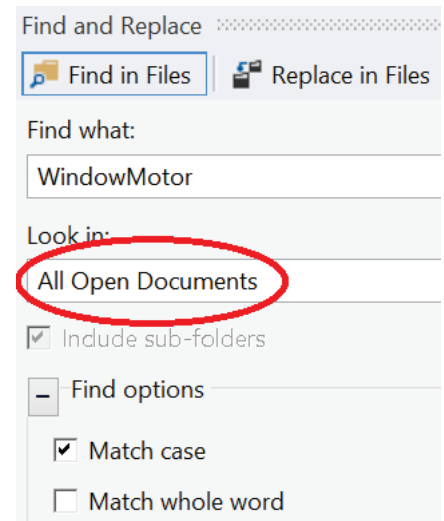
Copying an Existing Device

Below is a list of steps that create a new device.

- Duplicate the MY files of a similar device and rename as needed (e.g. change "Dev555" to "Dev444" within filename). An example of device MY files are shown to the right.
- Add these new files to both the Visual Studio and DAVE projects. Visual studio is faster and it is therefore recommended that you get things working there first.
- Open all your new MY files and replace your new "Dev" number with the original "Dev" number, *only* in open files (e.g. change "Dev555" to "Dev444"). Make sure you do not do this in all project files, only open files.
- We are assuming your new device has the same device type as the original device (e.g. VariableControl). If this is not the case, you need to change your DeviceType Handler port (e.g. VariableControl at Port#1).
- Your device has a name (e.g. "WindowMotor"). Search and replace this text in your open MY files as needed. Shown to the right is the Visual Studio Find in Files panel. Notice how Look In "All Open Documents" is selected. This enables us to focus only on the open files.
- Open file MY_Project_Compiler_Switches.h and perform the following steps:
 - #define a BB_BUILD label that pertains to your device. For an example, see "BB_BUILD_Dev555_VariableControl".
 - Add this label to file MY_Master_Include_File.h, in a manner similar to that shown below:



```
MY_Dev555_VariableControl
├── MY_Dev555_port_DeviceCommon
│   ├── MY_Dev555_port_DeviceCommon_setup_ncr.cpp
│   ├── MY_Dev555_port_DeviceCommon.cpp
│   └── MY_Dev555_port_DeviceCommon.h
├── MY_Dev555_port_MeasurementSys
│   ├── MY_Dev555_port_MeasurementSys_setup_ncr.cpp
│   ├── MY_Dev555_port_MeasurementSys.cpp
│   └── MY_Dev555_port_MeasurementSys.h
├── MY_Dev555_port_VariableControl
│   ├── MY_Dev555_port_VariableControl_setup_ncr.cpp
│   ├── MY_Dev555_port_VariableControl.cpp
│   └── MY_Dev555_port_VariableControl.h
└── MY_Dev555_BuildOneDevice.cpp
    MY_Dev555_EntireDevice.cpp
    MY_Dev555_EntireDevice.h
    MY_Dev555_Interface.h
    MY_Dev555_Master_Include.h
```



```
#ifdef BB_BUILD_Dev555_VariableControl
#include "../MY_Devices/MY_Dev555_VariableControl/MY_Dev555_Master_Include.h"
#endif
```


- #define a BUILD_STRATEGY label that pertains to your device. For an example, see "BB_BUILD_STRATEGY__Window_Controller_NetworkDevice".
- Set the BB_BUILD_STRATEGY to your new label, an example of which is shown below.

```
#define BB_BUILD_STRATEGY BB_BUILD_STRATEGY__Window_Motor_SubnetworkDevice
```

- Try compiling one file. If fails, you might need to adjust your compiler switches.
- Compile and create one new device, without making other changes. Try downloading into your microcontroller using DAVE. Get this working before making more changes.
- When working under DAVE, make sure you click the Generate Code button after changing Apps.
- When working under DAVE or Visual Studio, make sure you place a break point in LogErrorCode (). This is only called if a problem occurs. When that occurs, fix the problem before continuing.
- Search your MY files for the following text and update as needed:
 - SENSOR CHANNEL DECLARATION
 - PORT DATA TYPE DECLARATION
 - OUTPUT CHANNEL DECLARATION
 - :Setup_Hardware_ADC_Measurement_System
 - :Setup_Measurement_Scaling_System
 - :Measure_One_Analog_Input_Channel
 - :Update_One_Control_Output_Channel
- Compile and get this working before continuing.
- Update your immutable fields as needed. For details, search "DEFINE IMMUTABLE". Compile and get your device working before continuing.
- Continue to refine your code, focusing on unique aspects of your device.

Alternative to Copying Existing Files

Notice how Dev555_BUILD_... () functions in file MY_Dev555_BuildOneDevice.cpp creates a device with a specific ProductType and DeviceType. An alternative to copying existing files, described above, is to rework the Dev555 MY files by adding a routine that sets DeviceType/ProductType differently via a new BUILD function.

Downloading DAVE Software

To download free DAVE software, one must:

- [Register](#) at Infineon (try to download and it will push you into registering)
- Download 1.2GB file (Dave, Xmc libraries, examples)
- Unzip to 4GB total and place at C:\DaveSoftware\ directory (not nested deep into My Documents since that will exceed 256 maximum path length limitation).

Learning DAVE

To learn more, see [Introduction](#), see [Getting Started Guided](#), and see [Overview](#).

The [Working with DAVE Apps](#) document is helpful too (AP32295, 70pg pdf).

Also, after you run DAVE software, select HELP / HELP CONTENTS in the menubar and then see "XMC Lib Documentation" and "DAVE User's Manual". The "Getting Started" Chapter within the User's Manual is approximately 50 pages and is worth reading.

The DAVE Forum

One must register to view all posts within the DAVE [Forum](#). This registration is different from the Infineon Corporate registration (Dave Forum vs Infineon Customer).

Example Projects

For a list of high-quality example projects maintained by Infineon Corporation, click [here](#). If Google Chrome does not want to download .zip, try Internet Explorer.

Xmc4200 Reference Manual

To view the Xmc4200 Reference Manual (2100 page PDF), click [here](#).

Helpful YouTube Channels

- [Asright Asrain](#)
- [Infineon4Engineers](#)

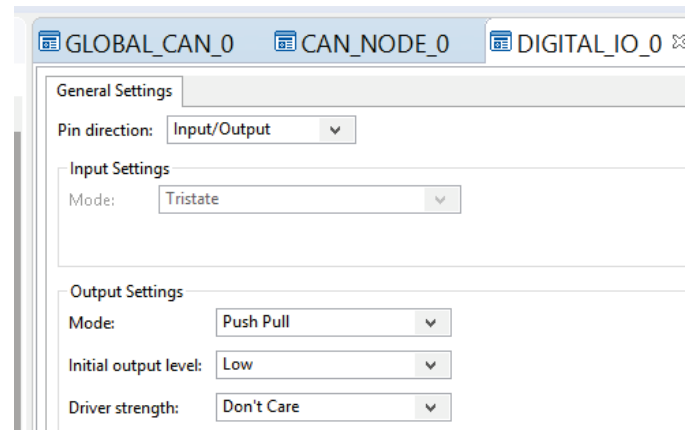
References to Applications Notes

Section "Xmc4200 Processor Resources" of the [GWeinreb Manhattan2 ResearchNotes.xlsx](#) spreadsheet lists a variety of helpful references.

Xmc4200 Processor Resources											
		Xmc42xx Processor									
			Product Page, XMC4200F64K256BAXQSA1								http
			Datasheet								http
			Reference Manual								http
		Xmc Power Application Notes									
			Introduction to Digital Power Conversion								http
			Synchronous buck converter with XMC™ Digital Power Explorer Kit (Xmc4200), #AP32319								http
			Application - Power Conversion - XMC™ in Power Conversion Applications								http
			Peak current control in XMC4								http
			Application - Power Factor Correction (PFC) with XMC								http
			Application - Power conversion - Power Management Bus (PMBus™)								http

One creates a digital I/O signal by adding a DIGITAL_IO App to the App panel. One attaches to a hardware pin via the PIN mapping panel or Manual Pin Allocator.

One can then read or write via several routines: DIGITAL_IO_SetOutputLow (), DIGITAL_IO_SetOutputHigh (), DIGITAL_IO_GetInput (). For an example, search for "DIGITAL_IO_LED" in the DavePrj_BB project.



See Also:

- Search "Digital_IO" in the DavePrj_BB project.
- In the DAVE Help menu, select Help Contents / Apps / DIGITAL_IO / Usage.
- In the DAVE Help menu, see Help Contents / DAVE User's Manual / Getting Started / "10. Pin assignment".
- In the DAVE Help menu, see Help Contents / DAVE Apps / DIGITAL_IO. Sections 'Overview', 'Usage' and 'Methods' are most helpful.
- For examples of going to direct to registers to set digital output bits, search DavePrj_BB for "BB_DioBit_SetOutputLow".
- Asrain Video [#1](#): Blinking LED (DIGITAL_IO App and Pin Assignment).
- Asrain Video [#4](#): DIGITAL_IO App input pin is connected to a physical button, it is read using DIGITAL_IO_GetInput (), and then the signal is output to another DIGITAL_IO App pin via DIGITAL_IO_ToggleOutput ().
- Asrain Video [#2](#) & [#5](#): Digital I/O pins are set up and maintained with XMC library routines, instead of Apps. One uses internal structs XMC_GPIO_PORT5 and XMC_GPIO_PORT15, programs them via XMC_GPIO_Init (), sets bits via XMC_GPIO_ToggleOutput (), and gets bits via XMC_GPIO_GetInput ().
- Asrain Video [#3.1](#) & [#3.2](#) & [#6](#): Program digital I/O pins by reading and writing directly to/from internal registers. This is confusing, yet does allow one to r/w multiple pins with one r/w command and only consumes 35nSec of processor time.
- Asrain Video [#24](#): DIGITAL_IO_0 pin input drives EVENT_DETECTOR signal_a, EVENT_DETECTOR trigger_out drives EVENT_GENERATOR trigger_in, EVENT_GENERATOR iout drives INTERRUPT_1 sr_irq, INTERRUPT_1 drives EXT_ISR () which toggles DIGITAL_IO_3 pin output. Also, TIMER App drives INTERRUPT_0, drives Timer_ISR (), toggles DIGITAL_IO_1 and 2 outputs.

For information on programming the internal XMC 12bit A/D, see:

- Search "Measure_One_ADC_Channel" within the DavePrj_BB project.
- Xmc4200 Reference [Manual](#) Chapter "Versatile Analog-to-Digital Converter (VADC)".
- In the DAVE Help menu, see Help Contents / DAVE Apps / ADC_MEASUREMENT, ADC_QUEUE, ADC_SCAN and ANALOG_IO (pin). Sections 'Overview', 'Usage' and 'Methods' are most helpful.
- Asrain Video [#7](#): ADC_MEASUREMENT and ANALOG_IO (input pin) Apps read A/D, and ADC_MEASUREMENT_GetResult () (600nSec) is called by ISR upon completion. StartConversion () (400nSec) is placed in the GetResult () ISR, causing this to run continuously as fast as possible (which consumes much of the processor). "Number of Measurements" is really "Number of channels". If one sets this to 2, for example, you can get "Channel_A" and "Channel_B", and you refer to these as ADC_MEASUREMENT_Channel_A (and _B) when you call GetResult (). You refer to the ADC_MEASUREMENT App when you call StartConversion ().
- Asrain Video [#8](#): Similar to above, yet works with lower level drivers.
- Asrain Video [#9](#): Similar to above, yet does r/w directly to registers, which is confusing, yet also fast.
- For examples of going to direct to a/d registers, search DavePrj_BB for "XMC_DirectToRegister_StartConversion" and "XMC_DirectToRegister_GetResult".
- GetResult/StartConversion Speeds: 600/400nSec with Apps, 300/200nSec with low level drivers, and 80/60nSec direct to registers.
- Asrain Video [#11](#): ADC Continuous Conversion with DAVE Apps + Micrium. This shows continuous conversion via the ADC_MEASUREMENT App, which drives its event_result_b output signal, which drives the INTERRUPT sr_irq input, which drives the ADC_Result () ISR. This is set up to read 2 channels. ADC_MEASUREMENT has "continuous measurement" enabled, which means it runs continuously.
- Asrain Video [#12](#): Same as above yet calls to XMCLibs functions and no Apps.
- Asrain Video [#13](#): Similar to above yet work directly with registers.
- Asrain Video [#14](#): ADC Timer Trigger + Micrium. ADC_MEASUREMENTS_0: 2 measurements (2 channels), external trigger input (starts conversion), ChB Result Event, no ISR upon completion. TIMER_0: 10uSec time interval, ccu4, time interval event enabled. INTERRUPT_0: drives ADC_Result () ISR. ADC_MEASUREMENT event_result_ChB drives INTERRUPT_0 sr_irq. TIMER_0 event_time_interval drives ADC_MEASUREMENT_0 trigger_input. ADC_MEASUREMENT_0 ChA/ChB connected to pins via Manual Pin Allocator. Timer runs continuously driving a/d measurements every 10uSec (100ks/sec/channel).

- Asrain Video [#15](#): Same as above yet XMCLibs instead of Apps.
- Asrain Video [#16](#): Same as above yet direct to register.
- Asrain Video [#27](#): PWM App drives 10ksample/sec A/D digitizing via ISR. Also, PWM set up for ccu8 drives LED and changing duty cycle adjust LED illumination 0 to 100%. PWM period match is turned on which enables its period match output. This event_period_match output is then used to drive the ADC_Measurement external trigger_input. ADC_MEASUREMENT connects to an IC pin w/o using an ANALOG_IO App.
- Asrain Video [#28](#): PWM + EXTI + LTC2500 A/D (32bit SPI a/d Mikroe Click board). MCLK: sampling clock (different from spi/mosi/miso data clk), connected to DAVE PWM App, gets 64 pulses for each sample. PWM_CCU8: output to MCLK pin. EVENT_DETECTOR: trigger_out drives EVENT_GENRATOR trigger_in. EVENT_GENERATOR: iout output drives INTERRUPT_0 sr_irq input. INTERRUPT_0: drives DRL_ISR () (a/d finished w/ data). DIGITAL_IO_0 pin drives EVENT_DETECTOR signal_a input; and also drives PWM ext_event0. DIGITAL_IO_1 pin routes to LED light. This drives a/d ic with 64pulses per sample and causes ISR to fire when complete.
- Asrain Video [#29](#): This builds on previous video and adds SPI + DMA interface.
- Asrain Video [#30](#): This builds on previous video and as sample rate clock via TIMER.
- Search "analog" among Infineon's [example programs](#).
- DAVE A/D Apps:

2.2.3 ADC APPs

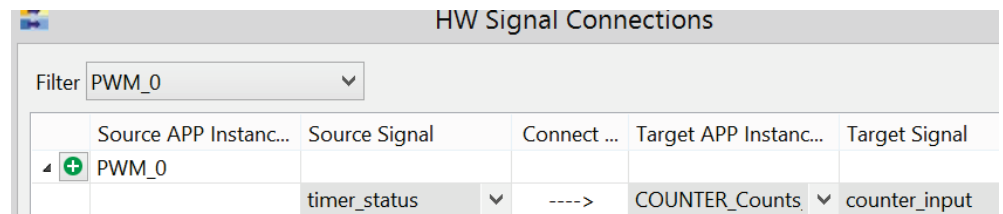
Table 7 DAVE v4: ADC APPs

S/N	APP Name	Description
1	ADC_MEASUREMENT	To incorporates analog to digital conversion for the required measurements.
2	ADC_QUEUE	To provide configurations for queue request source of VADC.
3	ADC_SCAN	To provide configurations for the scan request source of VADC.
4	GLOBAL_ADC	To configure the VADC global registers. It consumes CLOCK_XMC4 for XMC4x devices and CLOCK_XMC1 for XMC1x devices.

For information on programming timers, see:

- Search "timer" among Infineon's [example programs](#).
- In the DAVE Help menu, see Help Contents / DAVE Apps / PWM, TIMER, SYSTIMER, and COUNTER. Sections 'Overview', 'Usage' and 'Methods' are most helpful. In the DAVE Help menu, select Help Contents / Apps / Timer / Usage.
- If you do not work with Apps, make sure your ISR routine clears each event (e.g. calls `TIMER_ClearEvent()`).
- The TIMER App drives an Interrupt Service Routine (ISR). If you want a TIMER that outputs a signal, then consider the PWM App instead. If you want the PWM App to drive an ISR, then attach an INTERRUPT App to it.

- If you want a PWM output (clock) to drive a COUNTER App (input) then



route the PWM timer_status output (not clock out) to the counter_input. For details on status signal click [here](#).

- If you want a PWM App to drive an INTERRUPT App which calls an Interrupt Service Routine, then see the Dave Help User's Manual Getting Started "Chapter 5 -- Composing your first application using DAVE Apps".



- One can add an Interrupt Service Routine (ISR) via the SYSTIMER App. For an example of this, see `SETUP_SYSTIMER_10mSec_BbTick_InterruptServiceRoutine()` in the DavePrj_BB project. Make sure you don't have two ISR's working on the same memory, else they will conflict.
- See above A/D and Digital I/O Asrain videos that make use of TIMER and PWM.
- Search this document for "TIMER_" and "PWM".
- If you want to create a onetime hardware delay, search "CreateTimer" in [this](#) post.
- DAVE counter/timer Apps:

2.3.1 Generic APPs

Table 16 DAVE v4: Generic APPs

S/N	APP Name	Description
1	EVENT_DETECTOR	To provide the configuration for Event Request Source and Event Trigger Logic.
2	EVENT_GENERATION	To provide the configuration for Output Gating Unit.
3	RTC	To provide timing and alarm functions using RTC in the calendar time format.
4	SYSTIMER	To provide software timer interface for all non-RTOS applications.
5	WATCHDOG	To provide an interface to configure watchdog timer.

2.2.2 Timer/PWM related APPs

Code changes for PWM related APPs.

Table 4 DAVE v4: PWM-related APPs

S/N	APP Name	Description
1	COUNTER	To count the occurrence of external events using one timer slice of CCU4 or CCU4.
2	GLOBAL_CCU4	To initialize CCU4 global register set and to support synchronous start of CCU4 slices.
3	GLOBAL_CCU8	To initialize CCU8 global register set and to support synchronous start of CCU4 slices.
4	PWM	To generate a PWM using one timer slice of CCU4 or CCU8.
5	TIMER	To provide an accurate timer by using CCU timer. This can be used as trigger input to other peripherals or create an event.

For information on programming motor control, see:

- For an example XMC stepper motor project, click [here](#).
- For a partially working Xmc4200 motor project, see directory "DavePrj_StepperMotor_NotWorking_11-7-2020_Cephas".
- [BLDC motor control software using XMC](#)
- [Example Position Code](#) (file "Infineon-POSIF-XMC1000_XMC4000-AP32289_Example_Code-AN-v01_01-EN")
- POSIF (position) [Summary](#)
- [Summary](#) of XMC motor control applications
- Search "motor" among Infineon's [example programs](#).
- In the DAVE Help menu, see Help Contents / DAVE Apps / select motor Apps. Sections 'Overview', 'Usage' and 'Methods' are most helpful.
- DAVE motor control Apps:

2.4 Motor Control APPs

Table 25 DAVE v4: MOTOR CONTROL APPs

S/N	APP Name	Description
1	ACIM_FREQ_CTRL	To support Frequency Control- constant V/f Control for the AC Induction Motor.
2	AUTOMATION	Provide mechanism to connect two APPs using function block processor. This supports online parameter update and error logging. This consumes System Timer APP and provides task registration feature.
3	BLDC_SCALAR_CTRL	To support block commutation using 3Hall/2Hall feedback for Brushless DC motor.
4	GLOBAL_POSIF	To configure the POSIF module global settings.
5	MOTOR_LIB	This is a library APP and does not provide any UI configurations. This APP only provides common motor control algorithm APIs. This library is used by top level motor APPs.
6	HALL_POSIF	To get the motor position and speed using hall sensors separated at 120 degrees.
7	PMSM_SCALAR_CTRL	To support PMSM sinusoidal commutation using three or two hall sensors.
8	PWM_BC	To generate block commutation PWM waveforms (multichannel pattern) using CCU8 module.
9	PWM_SVM	To generate 3-phase Space Vector Pulse Width Modulated outputs using CCU8.

For information on programming CANbus, see:

- Search this file for "Working with CANbus"
- Search this file for "Testing Multiple Devices in a System"
- Search this file for "connect together two"
- CANbus Wikipedia [Article](#)
- Infineon CANbus [Summary](#) presentation
- Infineon CANbus Application Note [#AP32300](#).
- Xmc4200 Reference [Manual](#) Chapter "Controller Area Network Controller".
- Search "CAN" among Infineon's [example programs](#). Both "CAN_EXAMPLE_XMC45" and "MULTICAN_CONFIG_EXAMPLE_XMC47" are helpful. PDF files provide detailed documentation on each.
- For example Xmc4200 CANbus code, see directory "DavePrj_CANbus_12-20-2020_Frank".
- In the DAVE Help menu, see Help Contents / DAVE Apps / CAN_NODE and GLOBAL_CAN. Sections 'Overview', 'Usage' and 'Methods' are most helpful.
- DAVE communication Apps:

2.6 Communication APPs

Table 31 DAVE v4: Communication APPs

S/N	APP Name	Description
1	CAN_NODE	To configure a Node and MO registers of MultiCAN module. It provides a run-time APIs to change the node baud rate, to enable/disable Node and MO interrupts.
2	GLOBAL_CAN	To configure global resources of MultiCAN module.
3	I2C_MASTER	To implement I2C Protocol. It uses only master mode for communication. IIC uses two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock (SCL).
4	MANCHESTER_SW	To provide user configuration for Manchester CODEC.
5	SPI_MASTER	To implement SPI Master Protocol. It uses only master mode for communication.
6	UART	To configure a USIC channel to perform transfer and receive operations through UART protocol.

For information on programming SPI communication, see:

- Search "spi" among Infineon's [example programs](#).
- See Asrain's videos on LTC2500 connected to microcontroller via SPI
- Search this file for "SPI".
- In the DAVE Help menu, see Help Contents / DAVE Apps / SPI_MASTER. Sections 'Overview', 'Usage' and 'Methods' are most helpful.
- DAVE communication Apps:

2.6 Communication APPs

Table 31 DAVE v4: Communication APPs

S/N	APP Name	Description
1	CAN_NODE	To configure a Node and MO registers of MultiCAN module. It provides a run-time APIs to change the node baud rate, to enable/disable Node and MO interrupts.
2	GLOBAL_CAN	To configure global resources of MultiCAN module.
3	I2C_MASTER	To implement I2C Protocol. It uses only master mode for communication. IIC uses two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock (SCL).
4	MANCHESTER_SW	To provide user configuration for Manchester CODEC.
5	SPI_MASTER	To implement SPI Master Protocol. It uses only master mode for communication.
6	UART	To configure a USIC channel to perform transfer and receive operations through UART protocol.

For information on programming power conversion (e.g. DC-to-DC), see:

- Search "power" among Infineon's [example programs](#).
- See "Xmc Power Application Notes" in [GWeinreb_Manhattan2_ResearchNotes.xlsx](#).
- In the DAVE Help menu, see Help Contents / DAVE Apps / power conversion Apps. Sections 'Overview', 'Usage' and 'Methods' are most helpful.
- DAVE power conversion Apps:

2.5 Power Conversion APPs

Table 28 DAVE v4: POWER CONVERSION APPs

S/N	APP Name	Description
1	BUCK_VOLT_CTRL	To control different configurations of a buck converter in voltage mode.
2	GLOBAL_HRPWM	To initializes HRPWM global register set configuration.
3	HRPWM	To generate complementary PWM with optional high resolution positioning.
4	POWER_CON_LIB	Contains the PI & PID libraries for Digital Power Conversion.
5	PWM_CCU4	PWM generation using one timer slice of CCU4 to generate one PWM output.
6	PWM_CCU8	PWM generation using one timer slice of CCU8 to generate up to 4 PWM outputs.

For information on controlling LED illumination (e.g. 0 to 100% illumination of 10Watt LED bulb), see:

- Search "LED" among Infineon's [example programs](#).
- In the DAVE Help menu, see Help Contents / DAVE Apps / lighting Apps. Sections 'Overview', 'Usage' and 'Methods' are most helpful.
- DAVE lighting control Apps:

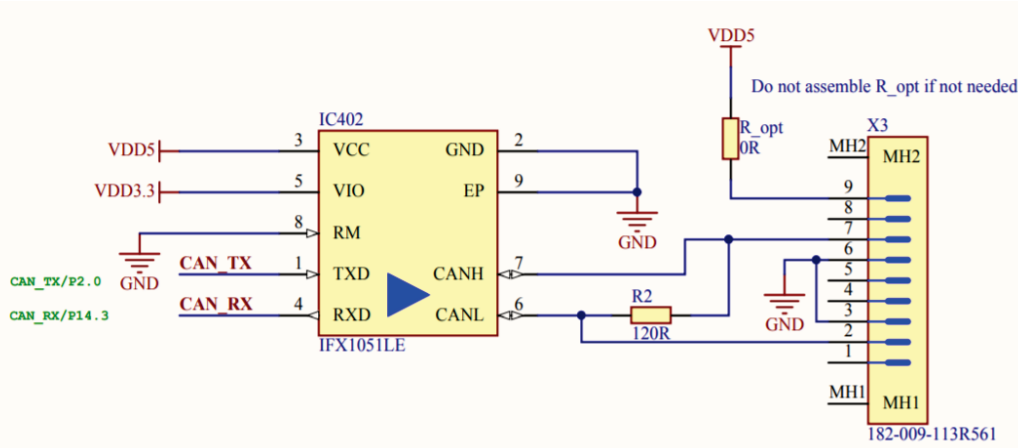
2.8 Lighting APPs

Table 37 DAVE v4: Lighting APPs

S/N	APP Name	Description
1	DIM_BCCU	To configure Brightness and Color Control Unit (BCCU) Dimming Engine registers.
2	DMX512_RD	To provide user configuration for DMX512 application Stack.
3	GLOBAL_BCCU	To configure the global registers of the Brightness and Color Control Unit (BCCU).
4	LED_LAMP	To create a virtual lamp with up to 9 channels, with up to 9 PDM_BCCU APPs.
5	PDM_BCCU	To configure Brightness and Color Control Unit (BCCU) channel registers.

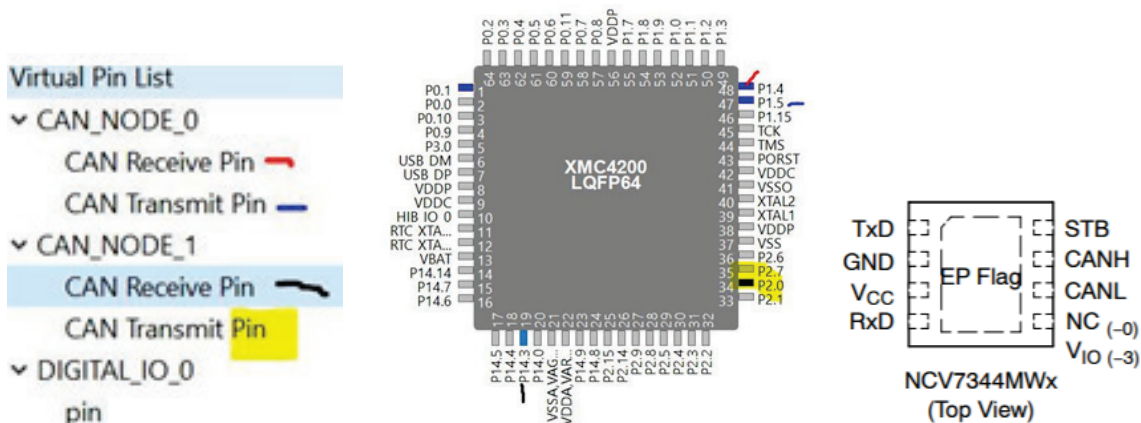
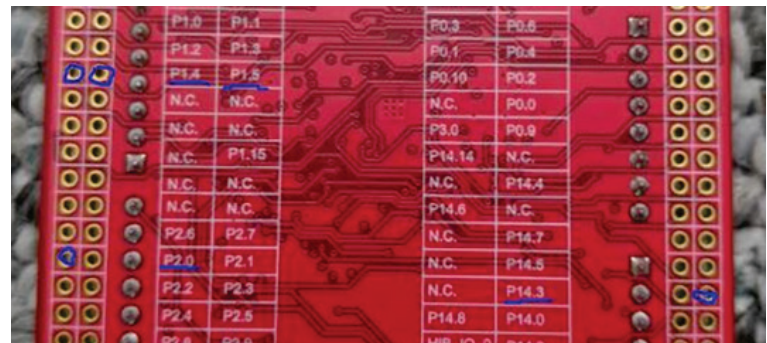
Cabling Together Two Xmc4200 Boards

To connect together two Xmc4200 boards: connect pin 7 to 7, pin 3 to 3 (or 6 to 6), and pin 2 to 2 between the two DB9 connectors ("Node #0"). The Xmc4200 Platform2Go board schematic is shown below. This board has a male Db9; therefore, a cable between two of these boards is Db9 female to Db9 female. For details on where to buy components, search this document for "Amazon".



Attaching 2nd CANbus Cable to Xmc4200 PlatformToGo

To attach a 2nd CANbus interface to an Xmc4200 board ("Node #1"), one needs to connect a transceiver IC (e.g. NCV7344) to specific CANbus pins on the microprocessor, as pictured below.

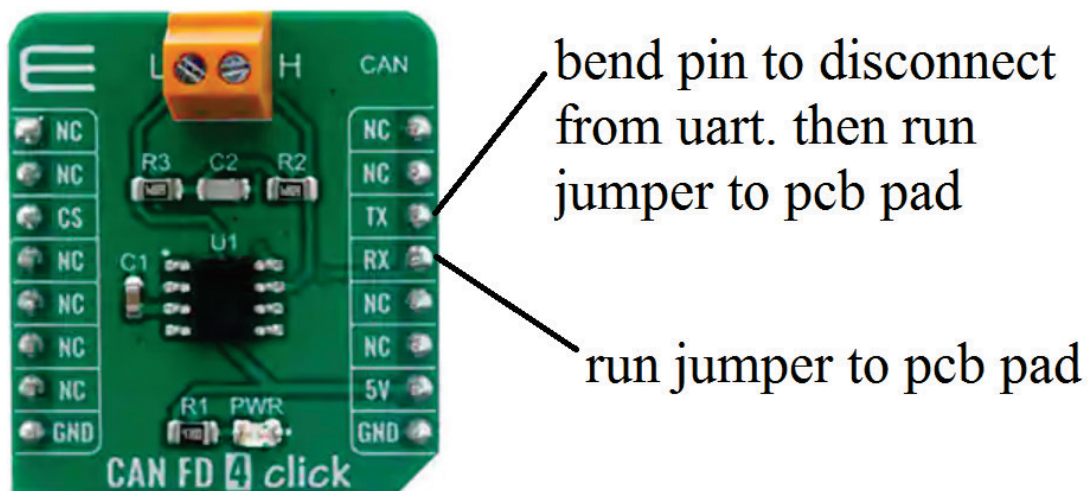


We specify which IC pins are utilized for CANbus using the DAVE Global Pin Allocator , pictured below. CAN_NODE_0 (Platform2Go DB9 Connector) is directed toward upstream devices (toward AMC master controller) and CAN_NODE_1 (Mikroe Daughterboard) is directed toward downstream devices (e.g. IC pins #35 and #48).

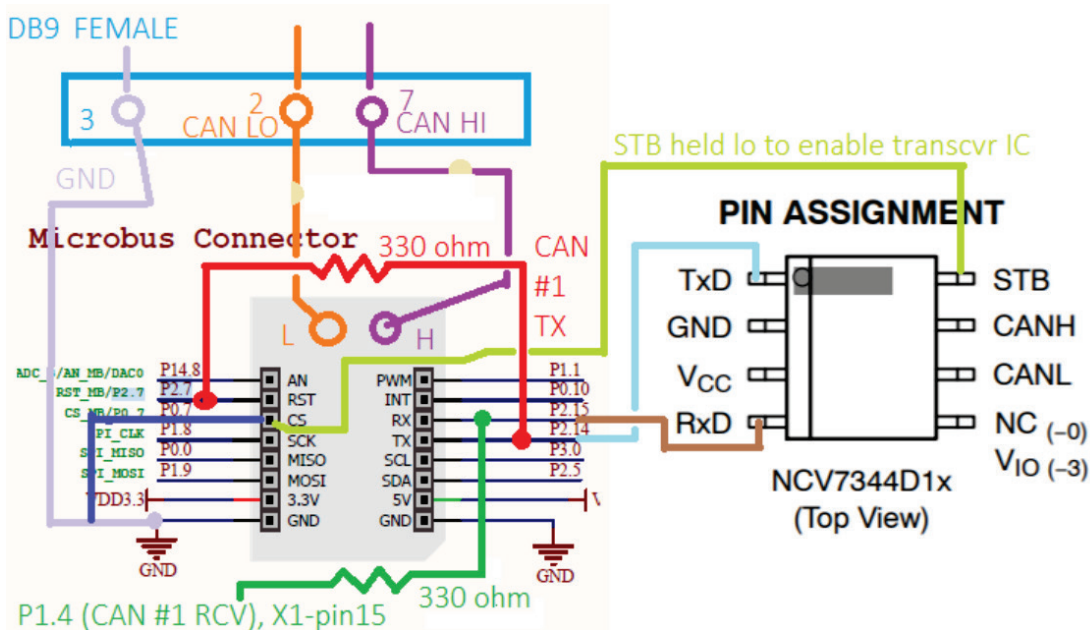
▴ CAN_NODE_0		
	CAN Receive Pin	#19 (P14.3)
	CAN Transmit Pin	#34 (P2.0)
▴ CAN_NODE_1		
	CAN Receive Pin	#48 (P1.4)
	CAN Transmit Pin	#35 (P2.7)

One can get a Transceiver IC mounted on Mikroe Click board ([CAN FD Click 4, #MIKROE-4107](#)) and then run jumpers from this board's Tx/Rx pins to Platform2Go PCB pads. This Mikroe board uses an [NCV7344](#) IC to translate TX/RX to CANbus Hi/L0.

We bend the RX, TX and Cs pins on the Mikroe board 90 degrees so that they do not enter the socket, to gain control over their use. Subsequently, we do not need to be concerned with a UART interacting with this circuit.

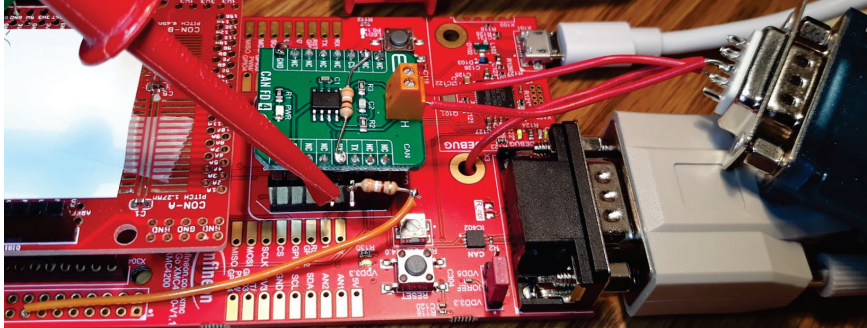


The below illustration shows how we set up CANbus Node #1.



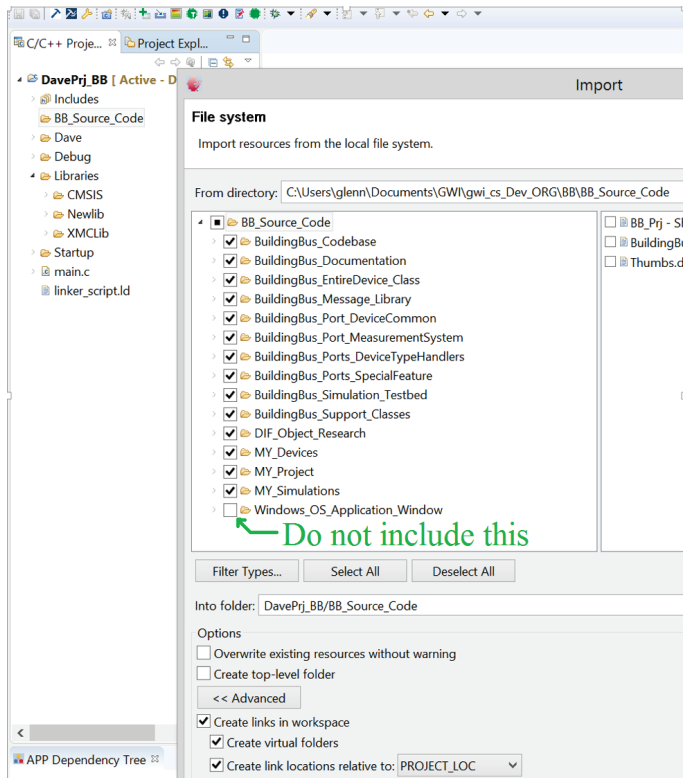
- We route CANbus TX signal from transceiver TX pin to microcontroller IC pin P2.7 via 330 ohm resistor (red in above dwg). This is labeled "RX" on actual Mikroe PCB, which is misleading. One can ignore this. They do this since UART Tx is attached to UART Rx, yet we are not working with UART. The Mikroe PCB P2.7 pin might be labeled "RST" on the Mikroe PCB, yet we do not use this as a reset signal. It is routed to microcontroller IC pin P2.7.
- We route CANbus RX signal from transceiver RX pin to microcontroller IC pin P1.4, via X1 connector pin 15, via 330 ohm protection resistor (green in above dwg). This is labeled "TX" on actual PCB, yet we ignore this.
- We do not insert the Mikroe PCB RX and TX pins into the socket and instead bend them 90 degrees and attach directly to them, to gain 100% control over them (e.g. so that someone does not try to attach a UART to them).
- Transceiver IC "STB" pin needs to be attached to GND to enable this IC (e.g. NCV7344). We bend this Mikroe pin 90 degrees to isolate it and then short it to GND, so that we are not dependent on the microcontroller to set it low via Mikroe "CS".
- Mikroe CANbus Hi is routed to pin 7 of DB9-female (purple in above dwg).
- Mikroe CANbus Lo is routed to pin 2 of DB9-female (orange).
- We route PCB GND to pin 3 of DB9-female.
- The Mikroe board and the Platform2Go boards already have a 120 ohm termination resistors built-in. One can see this with an ohm meter.

Below are several photos that show a MikroE PCB on a Platform2Go board. For more photos, search this file for "Multiple Devices in a System".



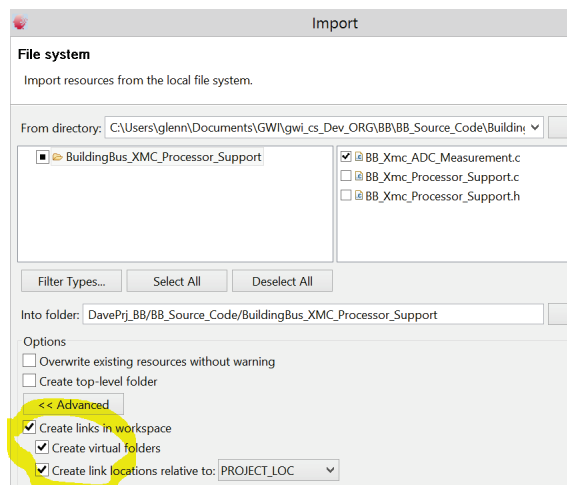
Importing Many Files into a Dave Project

One can import 100's of files/folders into a DAVE project with one operation using the Import feature, shown below.

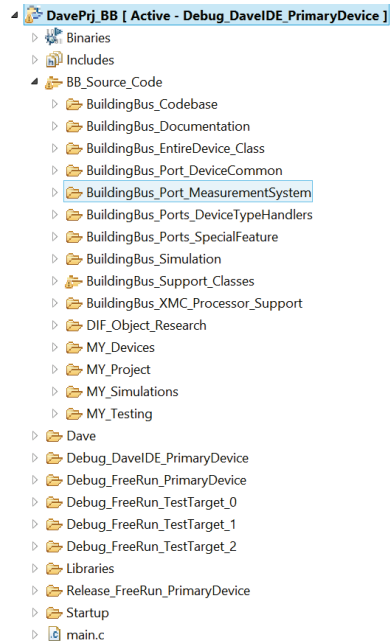


Create Links

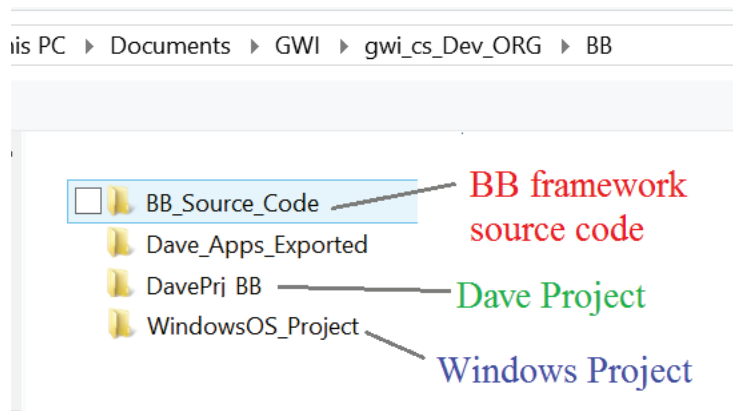
When Adding Files to a Project, Make sure you "Create Links" (not copy files). Click the ADVANCED button to access this checkbox.



The resulting DavePrj_BB project should look like the following within the DAVE development system:






We keep the BuildingBus framework source code in one folder, as shown below. This is accessed by both the DavePrj_BB project and the Windows OS project.



Reset DAVE Workspace

If you cannot launch DAVE (e.g. it crashes), find your Workspace folder, rename it to "SameNameAsBefore - damaged", create a new empty folder w/ the same name as the previous workspace, re-run DAVE, select the new empty folder, and then import your project into the new DAVE workspace system. DAVE will set up a new workspace in your new empty folder. If you want your Debug Configurations in the new workspace, then copy them from the damaged folder and place them into the new folder, as shown in the below example.

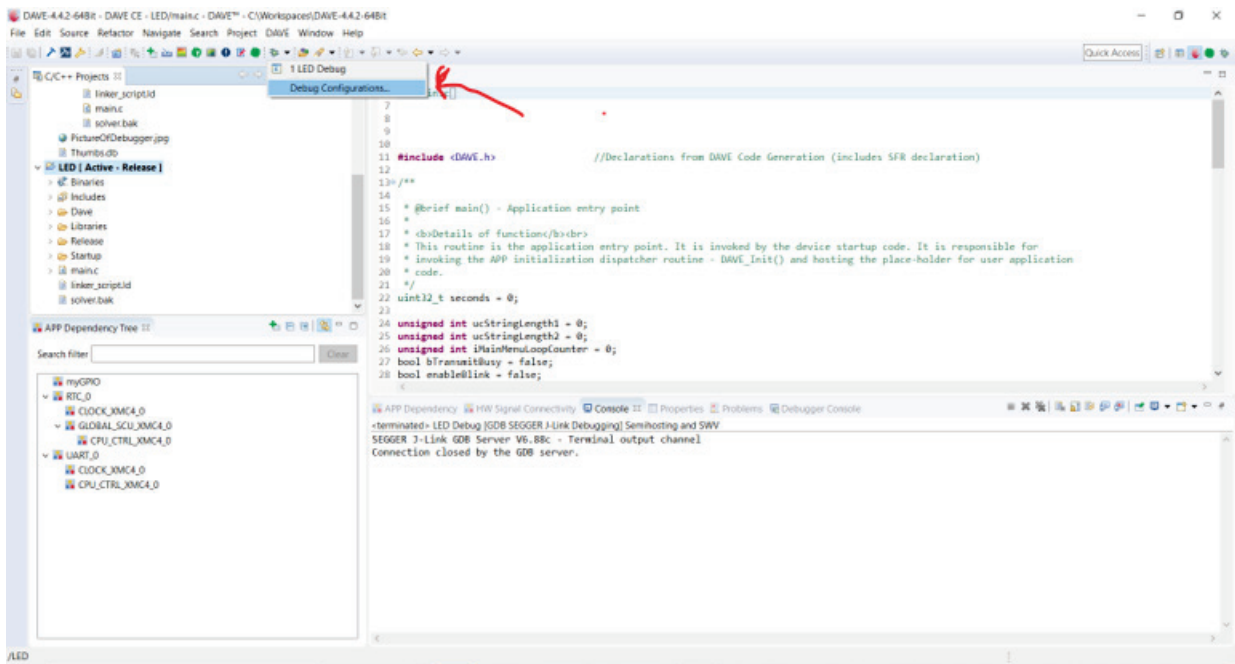
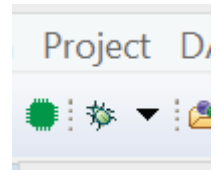
▶ Local Disk (C:) ▶ Workspaces ▶ DAVE-4.4.2-64Bit ▶ .metadata ▶ .plugins ▶ org.eclipse.debug.core ▶ .launches

Name	Date modified	Type	Size
 DavePrj_BB Debug_DaveIDE.launch	1/12/2021 2:22 PM	LAUNCH File	8 KB
 DavePrj_BB Debug_FreeRun.launch	1/11/2021 5:52 PM	LAUNCH File	8 KB
 DavePrj_BB Release_FreeRun.launch	1/10/2021 3:29 PM	LAUNCH File	8 KB

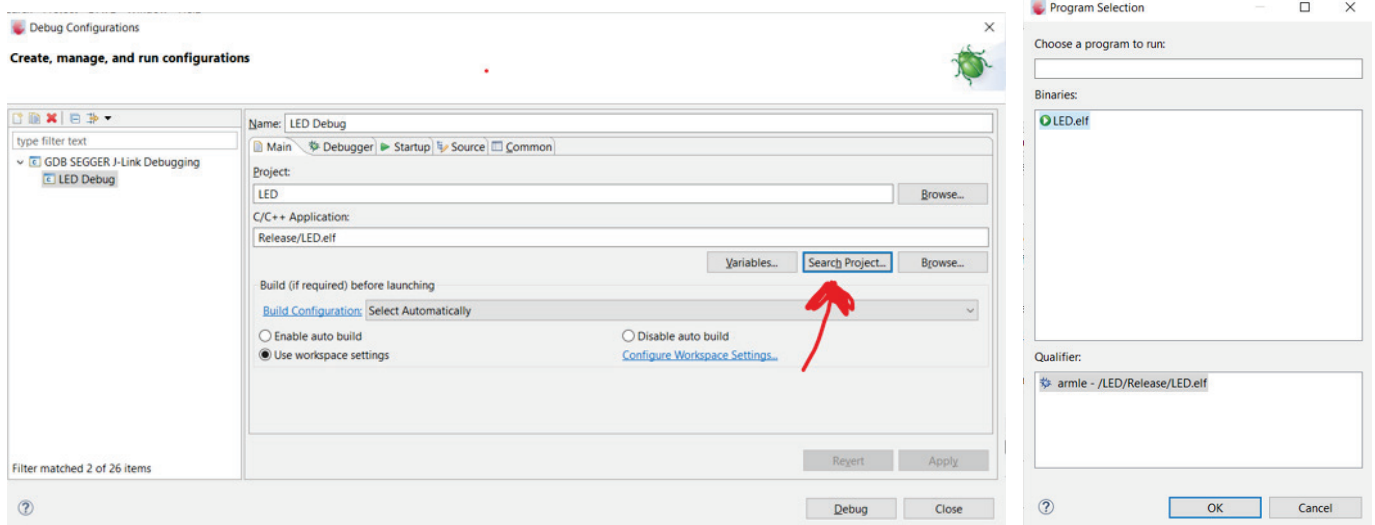
Chapter 27) Working with the DAVE Debugger

Setting up the DAVE Debugger with a .elf Debug File

If you run the debugger and get a “Program File Not Found” error, then perhaps the selected program file in the debugger settings window is not correct. To fix, press the debug icon, and click ‘Debug Configurations’ as shown in the screenshot below.



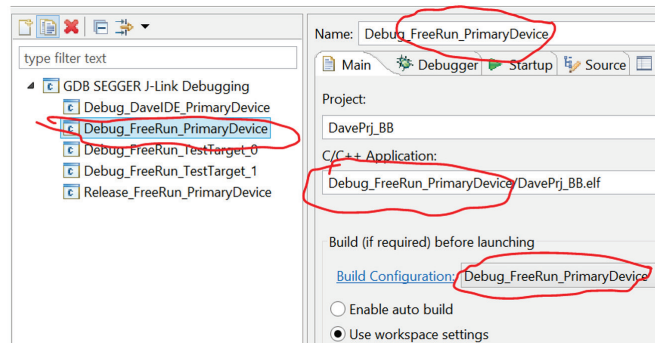
Then, in the resulting window, press the ‘Search Project’ button



Then select the .elf file that pops up in the menu (e.g. "Debug/DavePrj_BB.elf"). Press OK and then click the Apply button.

Setting up a Debug Configuration

Set up one configuration (Debug, not Release), in a manner similar to the dialog shown to the right (else the debugger might not work). If you have a Release configuration, then create a Debug configuration.



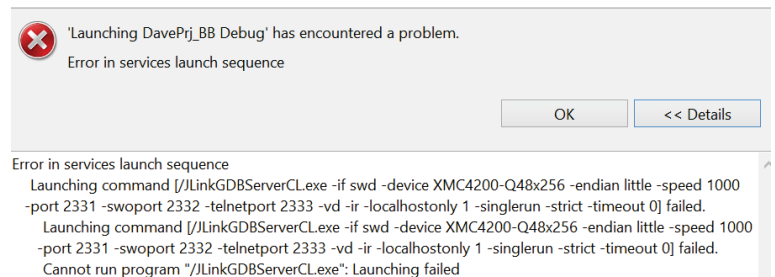
Compile, Download, Run, Debug



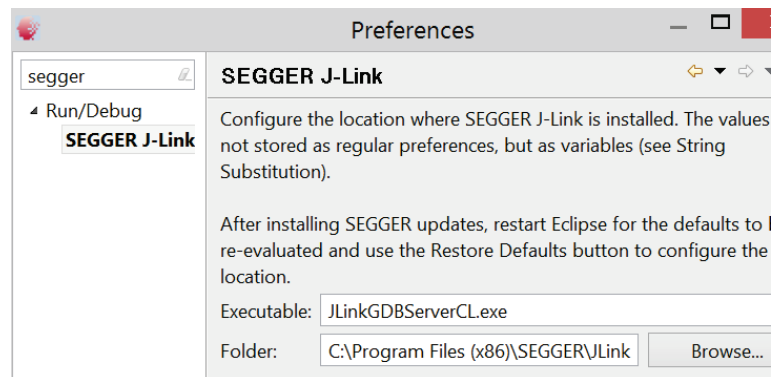
To download your code into the processor and run the program under the Debugger, click the Debug button, pictured to the left.

Connecting DAVE to the Segger JLinkGDBserverCL.exe file

If you get a "Launching Project-Name has encountered a problem" alert with details "Cannot run program JLinkGDBserverCL.exe", then you probably need to install the Segger debugger.



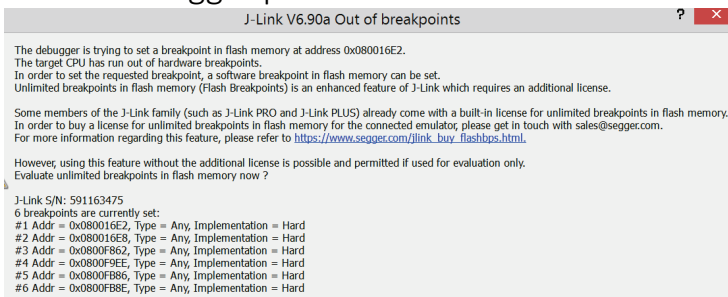
To do this, [download](#) and install. File "JLink_Windows_V690a.exe" will place debugger at "C:\Program Files (x86)\SEGGER\JLink".



Do not have the xmc42xx USB connector cabled to the computer USB connector when you install this software. When finished installing, physically attach the Xmc42xx via a USB cable. In the Windows Device manager you should see "J-Link Driver" appear under USB devices. Then, run the DAVE software. For details, click [here](#).

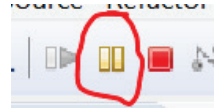
Free Debugger Has Limitations

The DAVE debugger provides a limited number of breakpoints with the free license.

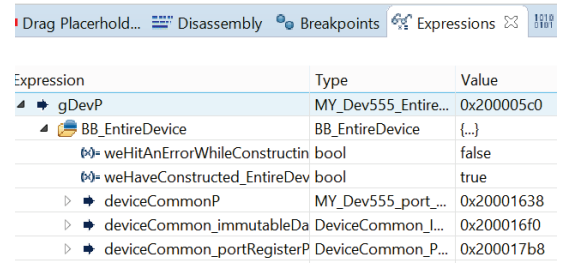


Suspend Wants Breakpoint at Start

If one wants to be able to suspend a program while running (Suspend button pictured at far right) to stop the program and examine variables, they need to first place a breakpoint at the start of main (), pictured below.



```
34     #if defined( ENABLE_PRINTING_TO_XMC_CONSOLE_WINDOW
35         Initialize_Xmc_Console_Printing();
36     #endif
```



Expression	Type	Value
gDevP	MY_Dev555_Entire...	0x200005c0
BB_EntireDevice	BB_EntireDevice	{...}
weHitAnErrorWhileConstructin	bool	false
weHaveConstructed_EntireDev	bool	true
deviceCommonP	MY_Dev555_port...	0x20001638
deviceCommon_immutableDa	DeviceCommon_I...	0x200016f0
deviceCommon_portRegisterP	DeviceCommon_P...	0x200017b8

Viewing a Device

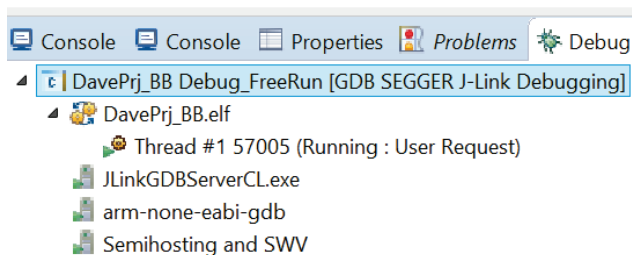
The device class is contained in the gDevP pointer; therefore, one can add this to Expressions pane to see internal variables.

Terminating the Debugger Session

If you get a debugger error similar to that shown here:

```
'Launching DavePrj_BB Debug_FreeRun' has encountered a
problem.
```

Then right-click the .elf file, shown below, and select "Terminate".



Dave IDE Controlled Image vs Free Running Image

There are two ways of controlling a downloaded DAVE image. One is to have the DAVE IDE software control the image via USB, which supports printing to the DAVE console. And the other method is to have an image that does not print to the DAVE console window, and the USB debug cable to computer is optional. The first method is used to develop code and requires a Windows computer be attached to the microcontroller via USB. The second method is for a board that may or may not be attached to Windows via USB (e.g. Platform2Go reference board or a dedicated PCB w/ a microcontroller). If you build an image that uses the first method, it will not run without the USB cable since it will not receive the begin execution command from the DAVE IDE software.

Build Configuration and Debug Configuration

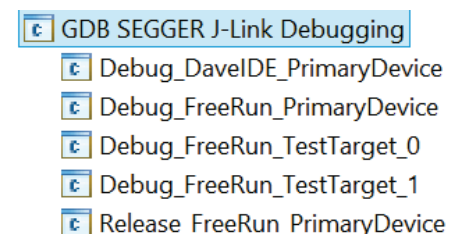
There are two different groups of settings within the DAVE IDE. One is the "Build Configuration", which controls how a project is compiled and linked; and the other is the "Debug Configuration", which determines how the program is downloaded and debugged.

Primary Device, Test Device #0, Test Device #1, etc

The project can build an image for multiple devices in a system, as described previously. The Primary device is the device in the system that runs the tests. The test devices respond to the primary, as described previously. The Build/Debug Configurations with "PrimaryDevice" in the name are used to make an image for the Primary Device, and Build/Debug Configurations with "TestTarget_N" in the name are used to make an image for Test Device #N.

Debug DaveIDE, Debug FreeRun and Release FreeRun

The DavePrj_BB project has multiple Debug/Build Configurations, as shown to the right. The "Debug_FreeRun_PrimaryDevice" Build Configuration is associated with the Debug Configuration with the same name. The same is true with the other configurations.



The Debug_DaveIDE configuration will not run if the debug USB cable and Windows DAVE IDE host are not attached. If you want a board that runs on its own, you need to download a configuration that has "FreeRun" in the name.

Configurations with "Debug" in the name enable debugging code and do not optimize; whereas configurations with "Release" in the name do the opposite.

The below table summarizes the various configurations.

	Debug_DaveIDE...	Debug_FreeRun...	Release_FreeRun
Build Configuration name	Debug_DaveIDE...	Debug_FreeRun...	Release_FreeRun
Debug Configuration name	Debug_DaveIDE...	Debug_FreeRun...	Release_FreeRun
Requires Computer USB Debugger Connection to Microcontroller PCB	Yes	no	no
Supports print to computer console pane via XMC_DEBUG()	Yes	no	no
XMC_DEBUG_ENABLE defined in C, C++ and Assembly preprocessor panel	Yes	no	no
BB_DEBUG_ENABLE defined in C, C++ and Assembly preprocessor panel	Yes	Yes	no
Call initialise_monitor_handles() in main()	Yes	no	no
C and C++ Compiler Optimization	no	no	Yes
Linker / Miscellaneous / Other flags set to " --specs=rdimon.specs"	Yes	Yes	no

Preprocessor symbols (e.g. XMC_DEBUG_ENABLE, BB_DEBUG_ENABLE)

Preprocessor symbol XMC_DEBUG_ENABLE enables the DAVE console printing and internal DAVE error checking.

Preprocessor symbol BB_DEBUG_ENABLE enables BuildingBus testing and error checking. In the code, one checks for "BB_DEBUG", which is defined in a .h file if BB_DEBUG_ENABLE has been defined (two step processes gives you control over this in .h file).

Preprocessor symbol WIN32_ is defined if running under Windows Visual Studio.

Preprocessor symbol ENABLE_BuildImage_TEST_TARGET_DEVICE_N is defined if building an image for Test Device #N.

Preprocessor symbols are defined for C compiler, C++ compiler, and assembler.

Debug_DaveIDE...

Defined symbols (-D)
XMC4200_F64x256
BB_DEBUG_ENABLE
XMC_DEBUG_ENABLE

Debug_FreeFun...

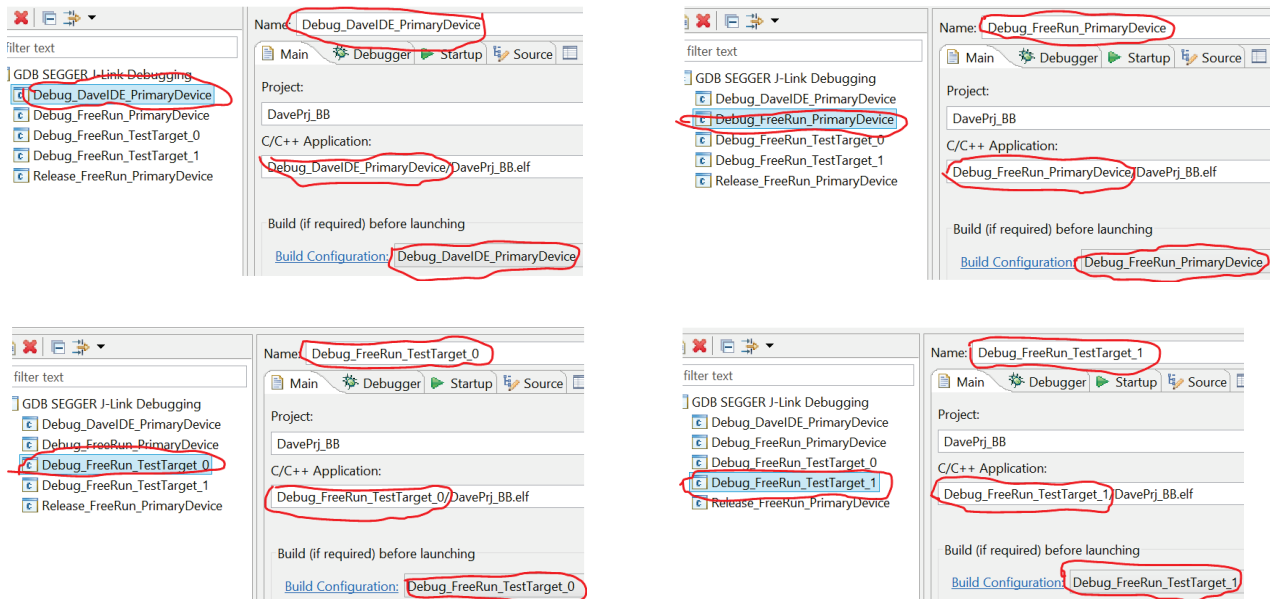
Defined symbols (-D)
XMC4200_F64x256
BB_DEBUG_ENABLE

Release_FreeRun...

Defined symbols (-D)
XMC4200_F64x256

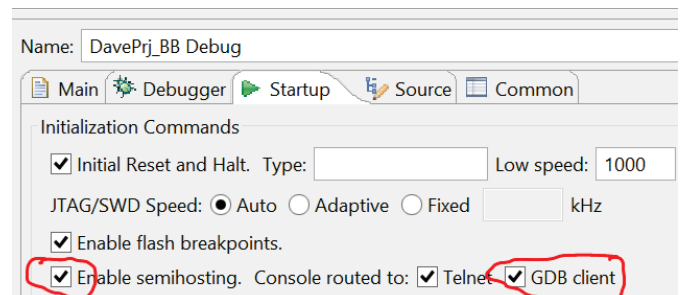
Putting it all Together

As shown below, the Debug_DaveIDE... debug configuration references the Debug_DaveIDE... resources, the Debug_FreeRun... debug configuration references the Debug_FreeRun... resources, etc. One needs to be careful to keep track of these.



Printing to the DAVE Console Panel

To print text to the DAVE Console panel, call `Print_Line_Char8 ()` within the BuildingBus system. For details on how this works, search "How to Enable XMC Debug Printing" in the DavePrj_BB project.



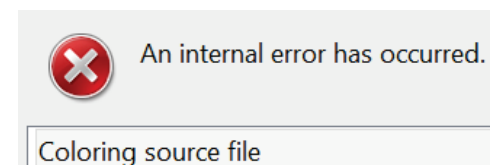
To set up console printing one must set up the following. For details, click [here](#).

- Enable "Semihosting" and "GDB Client" (among other things), as shown above.
- Set Linker / Miscellaneous / Other flags to " --specs=rdimon.specs", as shown to the right.
- Define XMC_DEBUG_ENABLE preprocessor symbol for C, C++, and Assembler.

Other flags `--specs=rdimon.specs`

Debug_DaveIDE... sets up printing as described above; however, Debug_FreeRun... and Release_FreeRun... do not print.

If the above linker options (i.e. " --specs=rdimon.specs") are not set up you might get the follow error when you mouse over Disassembly text (DAVE does not have access to resource it needs to show you information).





Remember to click "Generate Code" after updating your Apps, with each configuration.

LogErrorCode

In all cases, place a breakpoint at `BbError_int16 LogErrorCode(BbError_int16 errorCode)` `LogErrorCode ()` to identify problems as quickly as possible. You should never enter this routine with an error. If you do, it is a bug and needs attention.

Blinking LED



















In all configurations, we blink the Platform2Go LED once a second. For details, search "TOGGLE_LED_EVERY_1_SECOND".

Identifying Processor Position after a Crash

If your processor crashes and you are not sure what it was doing before the crash, then place 'gCpuMonitor' into the Expressions pane.

Parameters 'lastExecutionPosition' and 'lastForegroundThreadPosition' indicate last processor position. Note their values, identify their corresponding BB_ProcessorPosition enum label, and then search the code for those labels.

The 'weAreInside' struct, shown to the right, contains bits that are set if the processor is in a specific area. Look for cases where these are set and then search for their name to learn more.

 weAreInside	{...}
 receive_One_CANbus_MsgPacket	0
 transmit_One_CANbus_MsgPacket	0
 ISR_CAN_x_Tmit	0
 ISR_CAN_x_Rcv	0
 tmit_TryToPopltemOutOfFifo	0
 ISR_Rcv_IsPushingMsgIntoFifo	0
 fifo_push	0
 fifo_pop	0
 MasterIdleChore	0
 PrintLine	1
 PrintAddress	1
 Foreground_Port_Process_One_CANbusM	0
 Foreground_Processes_Multiple_INCOMIN	0
 Foreground_PushesOutgoingCANbusMsg	0
 Foreground_PushResponseMsg	0
 lastExecutionPosition	0 '\0'
 lastForegroundThreadPosition	0 '\0'

Detecting Stack Collision

If your processor crashed and you click on the Suspend button and see something similar to the below text, then you may have run out of memory in a stack collision.

```
336     .type Default_Handler, %function
337 Default_Handler:
338     b .
339     .size Default_Handler, . - Default_Handler
340
341     Insert_ExceptionHandler NMI_Handler
342     Insert_ExceptionHandler HardFault_Handler
343     Insert_ExceptionHandler MemManage_Handler
```

This means your local variables in your nested functions are using up too much memory. To see this, add a 4KB array to one of your functions, as pictured to the right, and try stepping through this code. You will probably end up w/ the IRQ handler response, shown above, with no stack crawl telling you how you got here.

```
void PrintLine_CallerAppendsCR_IN
{
    volatile int8_t data[4000];
    data[3999]=33;
```

Below is a list of other causes of a crash that do not produce a stack crawl:

- Non-existent Interrupt Service routine (ISR), or ISR with incorrect name, or ISR with a C++ prototype instead of C.
- One thread enters an XMC library routine, another thread interrupts, and the 2nd thread enters the XMC library as well (they are sometimes not re-entrant).

Stop Display Scaling with High Resolution 4K Monitors

Some versions of the DAVE.exe application program do not display fonts and icons of correct size, and do not fully show DAVE App dialog boxes, possibly when working with high resolution 4K monitors. For details, see [ref1](#) and [ref2](#).

One remedy is to place the following at the end of the DAVE.ini file, which resides next to the Dave.exe file.

```
-Dswt.enable.autoScale=true  
-Dswt.autoScale=150  
-Dswt.autoScale.method=nearest
```

The autoscale value that works that works the best might depending on your monitor. 4K monitors with 3840 x 2160 resolution seems to work well with autoScale=150. If you open the MULTICAN_CONFIG App, you might notice some of the text is clipped when autoscale is too small, or too large. 150 works just right.

The location of the DAVE.ini files is determined by where you placed your installation, after unzipping.

Local Disk (C:) > DAVE Sftwr 4GB > DAVE-IDE-4.4.2-64Bit > eclipse

Name	Date modified	Type
 DAVE.exe	9/25/2020 8:26 PM	Ap
 DAVE.ini	1/2/2021 1:25 PM	Co

In many cases, one can remedy this problem by selecting the application .exe file, right-click Properties, and then select "Disable display scaling on high dpi monitors". However, in order for that to work, one needs a manifest file and standard windows installation program, which is not the case when one installs by unzipping and placing the .exe at any location. In other words, Properties is not likely to help you.

Compiler Options

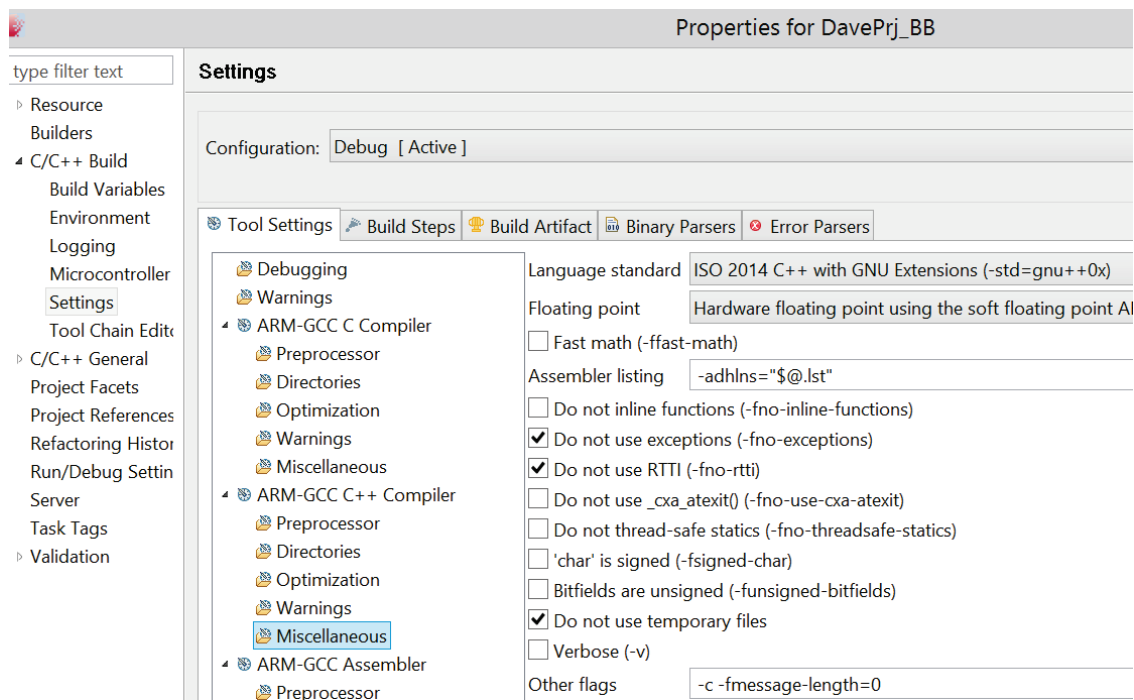
Compiler options enable one to optimize code and reduce total code size. The debugger will not work as well when code is optimized, yet optimization does reduce code size. For details, select the DAVE Help menu, select Help Contents / User's Manual / Getting Started / Chapter 14 - Tweaking the build toolchain settings.

If you want to update your DAVE development environment, then see Help Contents / User's Manual / Getting Started / 15. Keeping your DAVE installation updated.

If you want to move your DAVE project to a different microcontroller or update your Apps to a newer version, then see Help Contents / User's Manual / Getting Started / 16. The great migration.

To program the way your microcontroller boots up, see Help Contents / User's Manual / Getting Started / 18. BMI (Boot Mode Index).

C++ Compiler Version set to "2014 C++ w/ Gnu Extensions" (yet -std is much lower?)



Overview of Apps

One can program using icons, each of which are referred to as an "App".



After you place your Apps, you need to click the "Generate Code" icon, pictured to the left. DAVE then auto-generates multiple files and places them into the Dave folder. You can click on "Dave / Generated / DAVE.c" and "Dave / Generated / DAVE.h" to see what it did.

DAVE.c contains the initialization routine `DAVE_Init ()`, which is called from `main ()`. This initializes your Apps. The DAVE.h file includes the .h files that you will be using. To open a .h file, select it, right-click, and select "Open Declaration". For example, after placing the RTC (real-time clock) App, one can call `RTC_GetTime ()` to get the current date and time. To learn more about an App, select Help Contents in the Help menu, DAVE Apps, and then the App. If you see security alerts, click Cancel and Unblock as described later in this document. In the Help system, one can click on a list of topics to learn more. "Overview" is typically a good place to begin. "Usage" and "Methods" are helpful too.

RTC Documentation

Index

License Terms and Copyright Information
Abbreviations and Definitions
Overview
Architecture Description
APP Configuration Parameters
Enumerations
Data structures
Methods
Usage
Release History

Interrupt Service Routines

If an interrupt service routine (ISR) is referenced in an App and does not exist in the source code, you will not get a compile/link error. Instead, your program will crash and the cause will not be obvious. Also, your ISR needs to be a C function, which means you need to wrap the prototype and declaration in extern "C" {...}. If this is not done, the same will occur -- crash with non-obvious cause.

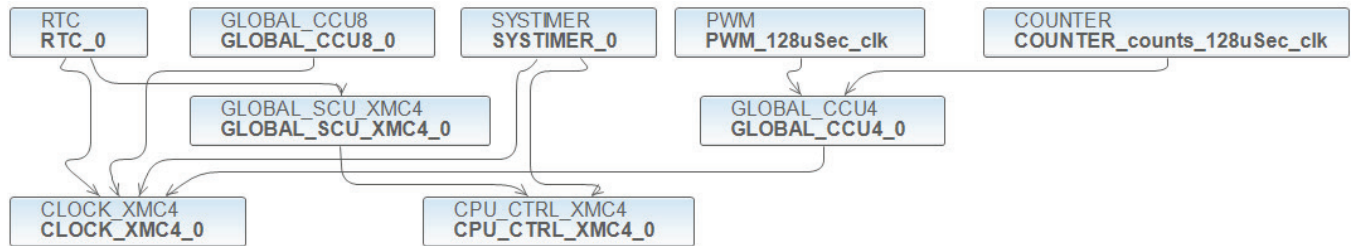
Notes on Apps

- To export/import one Apps, select it, right-click, select Export or Import.
- Select Report or Global Interrupts in DAVE menu for more info on your Apps.
- The E_EEPROM App helps to set up an area of memory to be used as FLASH
- For a summary of each App, see page 15 of [this](#) document.

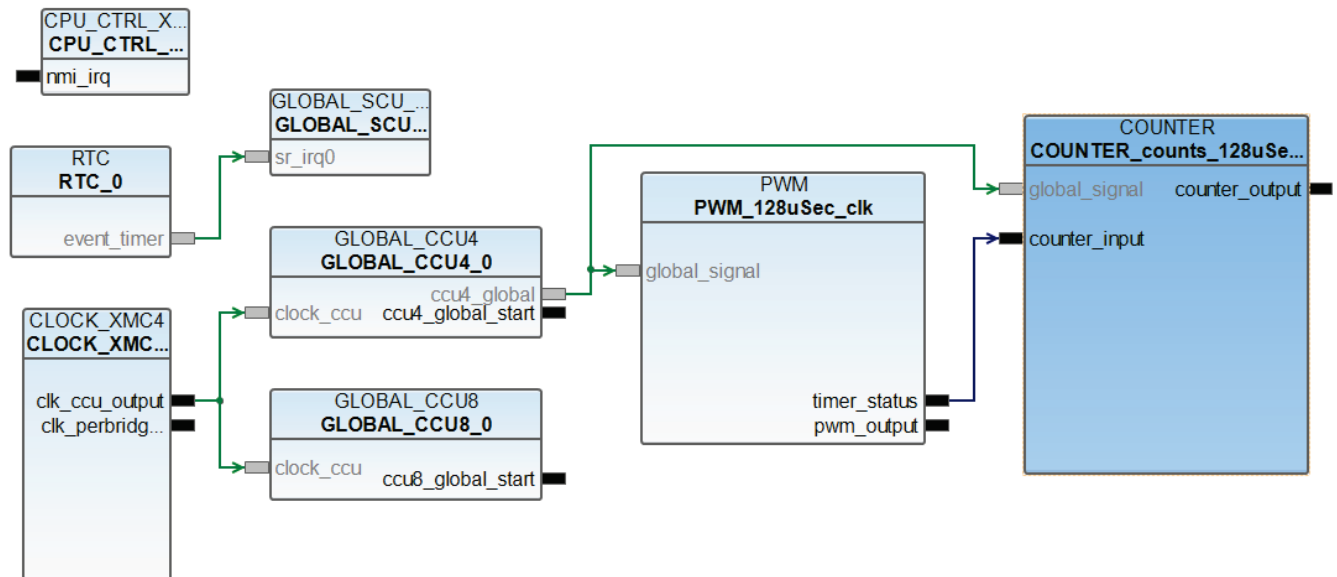
Chapter 33) Setting up 128uSec 64bit Counter and 10mSec ISR Apps

The DavePrj_BB project uses several DAVE Apps to create a Real-Time Clock (RTC App), 64bit hardware counter that counts a 128uSec clock (PWM and COUNTER App), and an Interrupt Service Routine that runs once every 10mSec (SYSTIMER).

The resulting Apps diagram is shown below:



The hardware connectivity is shown as follows:

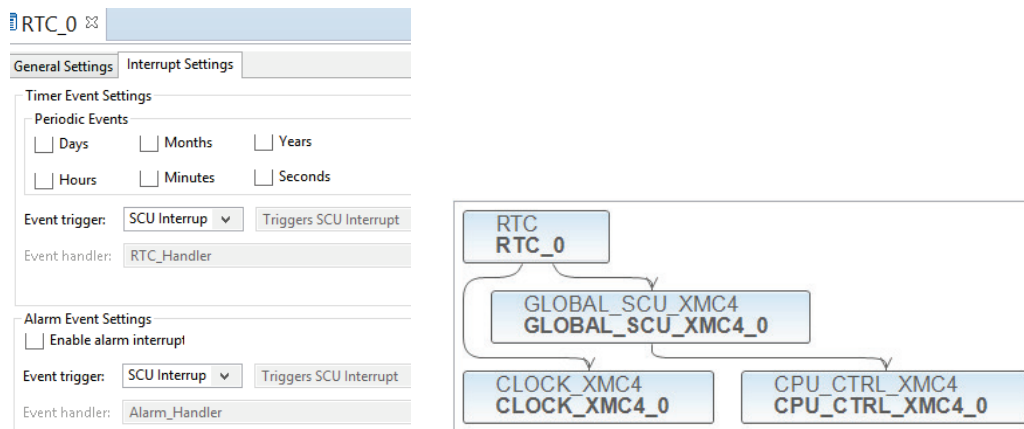


To pull these Apps into your project, import file "BB\\Dave_Apps_Exported\\DavePrj_BB\\DaveApps_Export_10mSecIsr_RTC_128uSec64bitCtr_12-21-2020.xml".

RTC Real-Time Clock

The RTC App is placed on the App panel. One can then call routines that get date & time. The RTC App does not call an Interrupt Service Routine (e.g. "seconds" is not selected in below dialog) since we already have one primary ISR that runs every 10mSec and one can

divide down to make an ISR that runs once a second (or more) that does not trample memory being worked on by the primary thread.

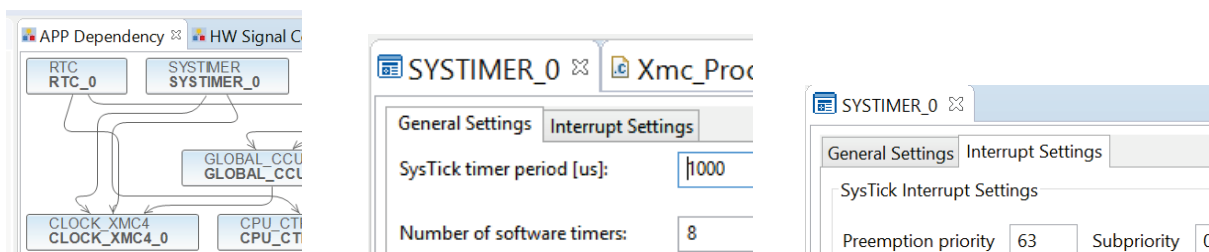


SYSTIMER Calls ISR Once Every 10mSec

The SYSTIMER App is placed on the App panel. Then, the

SETUP_SYSTIMER_10mSec_BbTick_InterruptServiceRoutine () routine sets up an interrupt service routine named HARDWARE_10mSec_BbTick_InterruptServiceRoutine () that is called once every 10mSec (search these names in DavePrj_BB for details). This has a low priority of 63 (others can easily interrupt) since we do not want to block important processes of more priority.

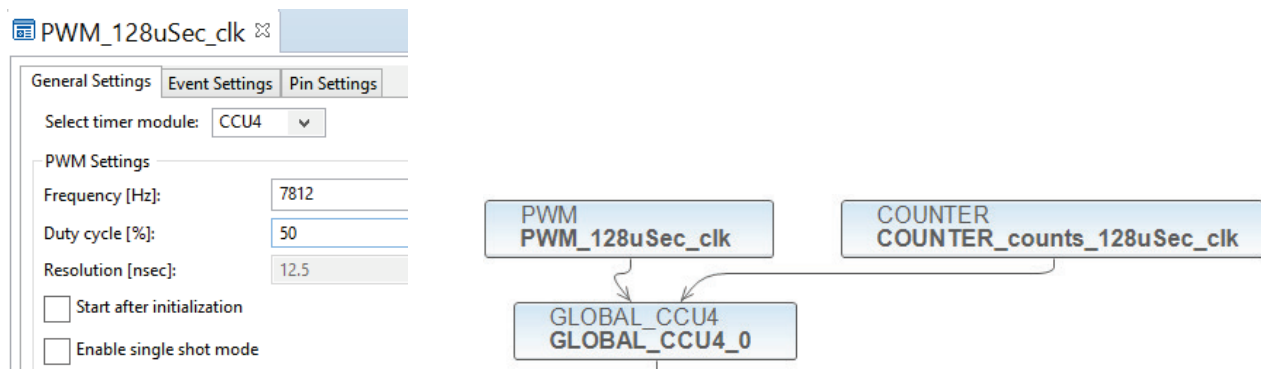
We set the SysTick timer period to 1000uSec. Subsequently it internally executes once every 1mSec, and when it does, it calls other ISR's at a multiple of that rate. The 10mSec ISR that we set up will be called once every ten 1mSec internal interrupts. The time accuracy on this ISR is not accurate since code can defer this interrupt (it might only run 50 times a second). If you want accurate time, then call GET_DateTime_1uSec_Units_1Jan2020_int64 ().



We interact w/ SYSTIMER via C code instead of an App since we want more control. Alternatively, one could use a PWM App drive to an INTERRUPT App. For details on this 2nd method, see the Dave Help User's Manual Getting Started "Chapter 5 -- Composing your first application using DAVE Apps".

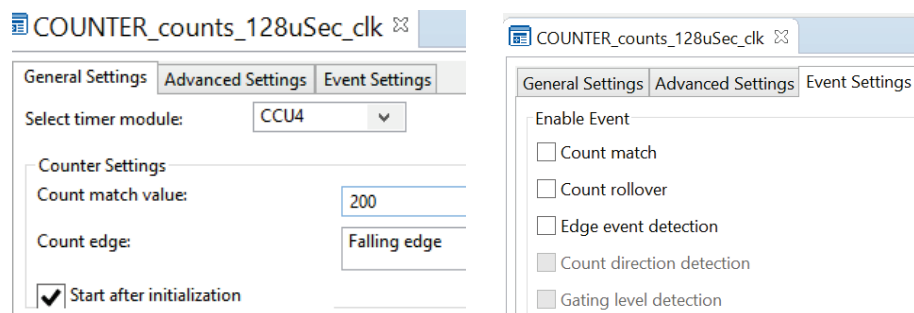
128uSec Clock Produced by PWM App

A PWM App named "PWM_128uSec_clk" outputs a 128uSec clock (7.8 KHz). SETUP_128uSec_Counter_Timer () sets this up when your program begins.



Hardware Counter Counts 128uSec

A COUNTER App named "COUNTER_counts_128uSec_clk" counts the 128uSec clock (7.8 KHz) from the above PWM. "Start after initialization" is selected.



The PWM "timer_status" (i.e. not clock output) is connected to the COUNTER counter_input via the HW Connections dialog. Subsequently, one can read the counter to establish accurate time.

Filter PWM_128uSec_clk					
	Source APP Instance Name	Source Signal	Connect To	Target APP Instance Name	Target Signal
+	PWM_128uSec_clk	timer_status	----	COUNTER_counts_128uSec_cl	counter_input
		Not Selected	----	Not Selected	Not Selected

GET DateTime 1uSecUnits 1Jan2020 int64()

One can call GET_DateTime_1uSecUnits_1Jan2020_int64 () to receive the number of int64 microseconds since midnight Jan 1, 2020.

Alternatively, one can call GET_DateTime_10SecUnits_1Jan2020_int32 () to get the number of int32 10second quanta since Jan 1, 2020.

These routines in turn call XMC_HARDWARE_GET_DateTime_1uSecUnits_1Jan2020_int64 (), which maintains a 64bit 1uSec counter that is based on the 16bit 128uSec hardware COUNTER/PWM Apps, described above. XMC_HARDWARE_GET_DateTime...() sees the 16bit counter roll-over every 10 seconds and increments the high 48bits component using

software. When one wants a 64bit 1uSec count, the system adds the high 48bits to the low 16bits to get an accurate hardware based count For details, see `XMC_HARDWARE_GET_DateTime_1uSecUnits_1Jan2020_int64 ()` in the DavePrj_BB project.

In order to get date/time with respect to Jan 1, 2020; devices send messages with the actual # of seconds and we adjust the base (`g_1uSec_Counter_BASE_int64`) as needed.

The difference between Jan 1, 2020 and Jan 1, 1970 is 1,577,836,800 seconds, as noted [here](#).

For more information, see file "BB_Xmc_Cpu_Support.c" in the DavePrj_BB project.



Remember to click "Generate Code" after updating your Apps.

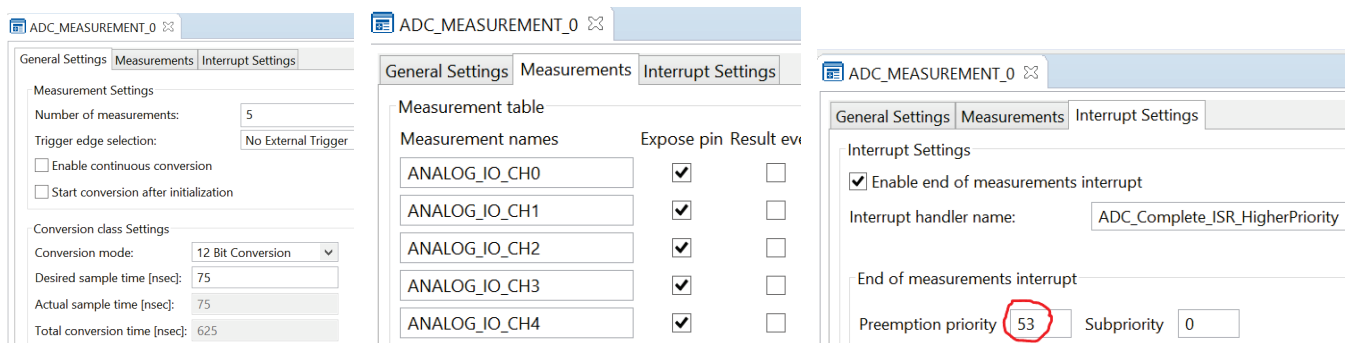
Testing

To test the above, we call `REALTIME_CLOCK_RTC_1_second_InterruptServiceRoutine ()` and `TEST_Counter_Timer_Hardware ()` once a second. We found the 128uSec 64bit hardware counter to be very accurate yet the 10mSec ISR counter was off time-wise by ~1%. It is ok, perhaps even good, if ISR's are off by 1%, or more, since it means the processor is able to focus on other things that are of more priority. The 10mSec ISR drives the main thread master idle chore, and is not considered to be of priority. An example of something with priority is we receive a CANbus message, which calls an ISR, and we need to pull it out of CANbus controller hardware before another message comes in and is ignored due to a full controller. In this case, the CANbus Receive ISR needs priority over the main thread.

Chapter 34) Setting up the Analog Input Channels Apps

The DavePrj_BB project sets up 5 analog input channels with an ADC_MEASUREMENT App. The 5 channels are named "ANALOG_IO_CH0", "ANALOG_IO_CH1", etc. These names are referenced in the source code.

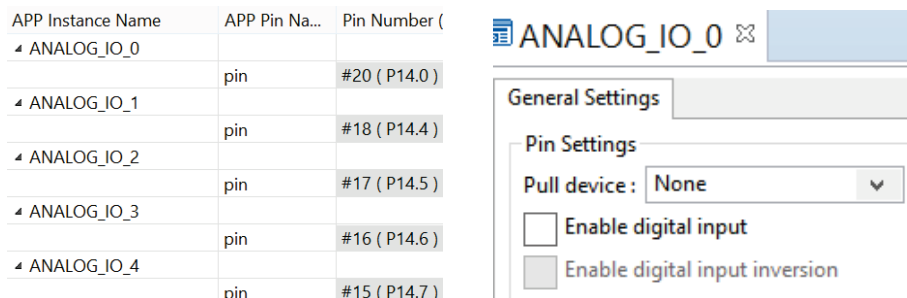
When the conversion of one A/D sample ends (0 to 4095 value), then ADC_Complete_ISR_HigherPriority () ISR routine is executed. This has priority 53 (not 63) since it needs a higher priority than the 10mSec interrupt system, otherwise it is blocked.



This project defines 5 analog input pins with ANALOG_IO Apps named "ANALOG_IO_0", "ANALOG_IO_1", etc; and these are mapped to pins 14.0, 14.4, etc.

One needs to make sure the # of channels set up here is the same or more than the # of sensors in the device. For example, if your device has 6 sensors connected to A/D channels, you need to specify 6 A/D channels in the A/D App. For an example of A/D channels being set up in a device, search "enum MY_Dev555_Sensor_ChannelIndex".

The HW Signal Connections dialog connects the channels to the pins (e.g. ANALOG_IO_0 to ANALOG_IO_CH0). One does this for all analog channels. Remember to click the SAVE button after you set up a signal connection.



HW Signal Connections				
er	ANALOG_IO_0			
	Source APP Instanc...	Source Signal	Connect ...	Target APP Instance Name
	ANALOG_IO_0	pin	----	ADC_MEASUREMENT_0
				ANALOG_IO_CH0

If you call `Measure_One_ADC_Channel ()` and ask for 100uSec of integration (i.e. averaging of multiple samples), and it takes 4uSec to do one A/D conversion, then `Measure_... ()` will return the average of 25 samples. This means the `ADC_Complete... ()` ISR will be called 25 times before the `Measure_... ()` routine returns the average of 25 samples. The `Measure_... ()` routine counts the number of samples, and is not controlled by time. Subsequently, if a 200uSec interrupt occurs during A/D conversion, it will still return the average of 25 samples. If you average 25 samples, for example, you reduce noise by the square root of 25 (i.e. 5-fold).

Measuring the Time it takes to Convert One Sample

A TIMER App named "TIMER_adc", and an INTERRUPT App named "INTERRUPT_adc" are set up to execute an ISR routine after a programmed number of microseconds have elapsed. This is used to measure the time that it takes to convert one A/D sample. We measure this time once, when one first starts up (before enabling CANbus interrupts). Then, we use this to calculate the number of samples to average given a microsecond integration time. Parameter "weAreMeasuringAdcConversionTime" is set true while doing this startup calibration procedure.

The TIMER_adc App is initially set up with a 10uSec time interval, yet this is re-programmed later. This timer drives an interrupt, which drives an ISR named `Timer_adc_ISR ()`, which tells us the time has elapsed. The interrupt priority is set to 60, since it needs to be higher than the 10mSec ISR at priority 63, else it is blocked (low number is high priority).

TIMER_adc

General Settings

Event Settings

Select timer module: CCU4

Timer Settings

Time interval [uSec]: 10

☐ Start after initialization

TIMER_adc

General Settings

Event Settings

☒ Time interval event

INTERRUPT_adc

Interrupt Settings

☒ Enable interrupt at initialization

Interrupt Priority

Preemption priority 60

Interrupt handler: TIMER_adc_ISR

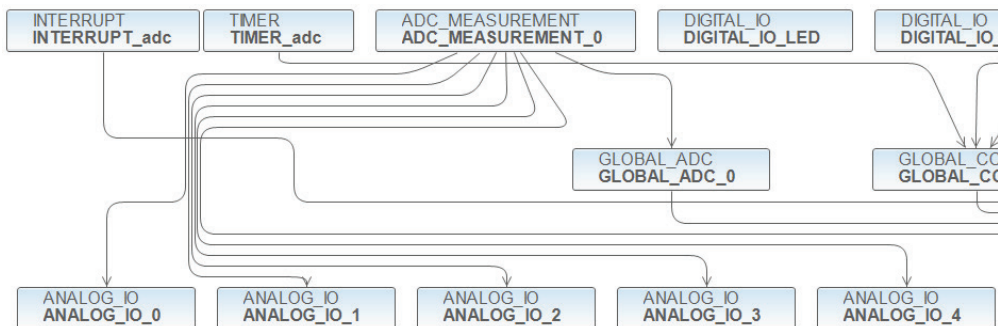
Iter	TIMER_adc				
	Source APP Instance Name	Source Signal	Connect ...	Target APP Instance Name	Target
	TIMER_adc				
		event_time_interval	---->	INTERRUPT_adc	sr_irq
		Not Selected	---->	Not Selected	Not Se

The Platform2Go board physically connects processor IC pin 14.0 to its R8 potentiometer, which means one needs to rotate the POT to see the voltage change. One cannot drive this pin w/ an external signal since it conflicts with this POT.



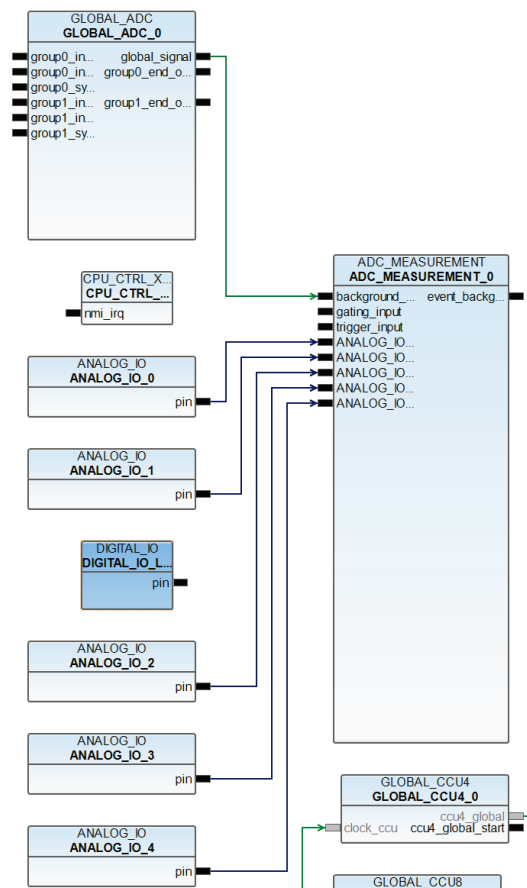
Remember to click "Generate Code" after updating your Apps.

Hardware Dependencies are set up as follows:



Hardware Connections are shown below. The Green lines are automatically installed by the DAVE system and the Blue lines are manually set up in the HW Signal Connections dialog (e.g. 5 analog input channels connected to ADC_MEASUREMENT system).

For more information, search the DavePrj_BB project for ":Measure_One_ADC_Channel", and search this document for "Programming the A/D".



Chapter 35) Setting up Digital I/O pin Apps

In the DavePrj_BB project, a digital i/o signal named "DIGITAL_IO_LED" is connected to the LED light at Pin0.1 via the DIGITAL_IO App; and a digital i/o signal named "DIGITAL_IO_ChZ" is connected to IC Pin #36 (P2.6).

DIGITAL_IO_LED

General Settings

Pin direction: Input/Output

Input Settings

Mode: Tristate

Output Settings

Mode: Push Pull

Initial output level: Low

Driver strength: Medium Driver

DIGITAL_IO_ChZ

General Settings

Pin direction: Input/Output

Input Settings

Mode: Tristate

Output Settings

Mode: Push Pull

Initial output level: Low

Driver strength: Weak Driver

One can connect these signals to physical IC pins via the Manual Pin Allocator or the Pin Mapping panel.

DIGITAL_IO_ChZ		
	pin	#36 (P2.6)
DIGITAL_IO_LED		
	pin	#1 (P0.1)

A logic 1 turns the LED on via Push-Pull mode ($3V / 680\Omega = 5mA$). This digital I/O is also routed to Platform2Go X2 connector #13 (IC pin #1 routes to connector X2 pin #13).

The LED signal is controlled with 3 routines: BB_SetOutputLow_Dio_LED (), BB_SetOutputHigh_Dio_LED (), and BB_ToggleOutput_Dio_LED ().

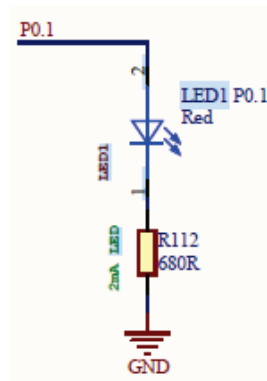
The ChZ digital output is controlled with 3 similar routines: BB_SetOutputLow_Dio_ChZ (), BB_SetOutputHigh_Dio_ChZ (), and BB_ToggleOutput_Dio_ChZ ().

Each of these takes about 125nSec on an Xmc4200.

For more information, search the DavePrj_BB project for "Dio_LED"; and search this document for "Programming Digital I/O".



Remember to click "Generate Code" after updating your Apps.

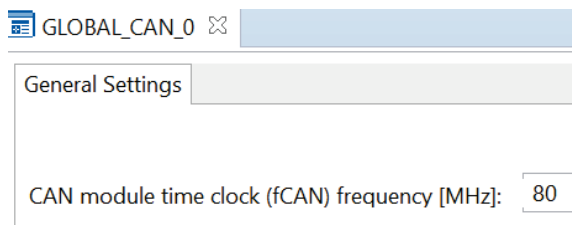


CAN_NODE and CAN_GLOBAL Apps

In the DavePrj_BB project we utilize a CAN_NODE and CAN_GLOBAL Apps to interface with CANbus. One CAN_NODE instance is used for each CANbus cable. CAN_NODE_0 connects to upstream devices and is required (toward AMC). And the optional CAN_NODE_1 connects to downstream devices (i.e. toward subnetwork). Some devices and subnetwork devices only have one node, and are therefore never a bridge between two CANbus cables.

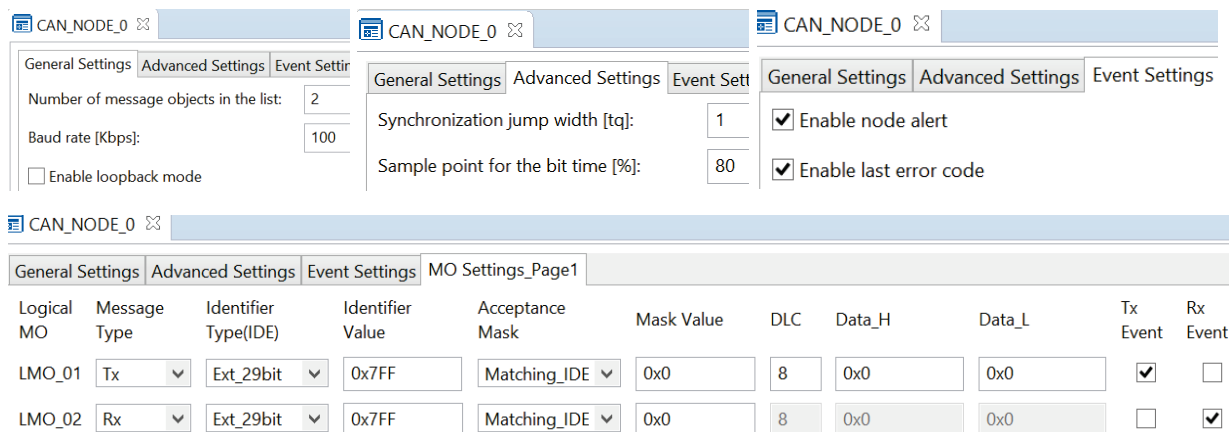
GLOBAL CAN App

We set up the Global CAN App as follows. This is used by the CAN_NODE Apps.



CANbus Node Apps

We set up two CAN_NODE Apps, one for each CANbus cable. The below pictures show CAN_NODE_0. We set up CAN_NODE_1 in the same way.



Interrupt Apps

We set up four INTERRUPT Apps for each CANbus node (4 for node #0, 4 for node #1, 8 total):

INTERRUPT_CAN_0_Rcv

Interrupt Settings

☐ Enable interrupt at initialization

Interrupt Priority
 Preemption priority Sub

Interrupt handler: ISR_CAN_0_Rcv

INTERRUPT_CAN_0_Tmit

Interrupt Settings

☐ Enable interrupt at initialization

Interrupt Priority
 Preemption priority Sub

Interrupt handler: ISR_CAN_0_Tmit

INTERRUPT_CAN_0_NodeAlert

Interrupt Settings

☐ Enable interrupt at initialization

Interrupt Priority
 Preemption priority Subpriority

Interrupt handler: ISR_CAN_0_NodeAlert

INTERRUPT_CAN_0_LastErrorCode

Interrupt Settings

☐ Enable interrupt at initialization

Interrupt Priority
 Preemption priority Subpriority

Interrupt handler: ISR_CAN_0_LastErrorCode

We connect NODE sources to the Interrupts as shown below. We do this for both Node #0 and Node #1.

CAN_NODE_0					
Source APP Insta...	Source Signal	Connect To	Target APP Instance Name	Target	
CAN_NODE_0	event_lmo_01_txinp	---->	INTERRUPT_CAN_0_Tmit	sr_irq	
	event_lmo_02_rxinp	---->	INTERRUPT_CAN_0_Rcv	sr_irq	
	event_node_alert	---->	INTERRUPT_CAN_0_NodeAlert	sr_irq	
	event_node_lec_error	---->	INTERRUPT_CAN_0_LastErrorCode	sr_irq	

CAN_NODE_1					
Source APP Insta...	Source Signal	Connect To	Target APP Instance Name	Target	
CAN_NODE_1	event_lmo_01_txinp	---->	INTERRUPT_CAN_1_Tmit	sr_irq	
	event_lmo_02_rxinp	---->	INTERRUPT_CAN_1_Rcv	sr_irq	
	event_node_alert	---->	INTERRUPT_CAN_1_NodeAlert	sr_irq	
	event_node_lec_error	---->	INTERRUPT_CAN_1_LastErrorCode	sr_irq	

Interrupt Priorities

We set up interrupt priorities as follows. Receive priority is highest since one needs to move that data before it is overwritten.

filter	ALL	
APP Instance Name	Interrupt Name	Priority
ADC_MEASUREMENT_0		
	ginterruptprio_backgnd_rs_intr	53
CPU_CTRL_XMC4_0		
	ginterruptprio_bus	0
	ginterruptprio_memmanage	0
	ginterruptprio_usage	0
INTERRUPT_adc		
	ginterruptprio_priority	60
INTERRUPT_CAN_0_LastErrorCode		
	ginterruptprio_priority	46
INTERRUPT_CAN_0_NodeAlert		
	ginterruptprio_priority	44
INTERRUPT_CAN_0_Rcv		
	ginterruptprio_priority	32
INTERRUPT_CAN_0_Tmit		
	ginterruptprio_priority	40
INTERRUPT_CAN_1_LastErrorCode		
	ginterruptprio_priority	45
INTERRUPT_CAN_1_NodeAlert		
	ginterruptprio_priority	43
INTERRUPT_CAN_1_Rcv		
	ginterruptprio_priority	31
INTERRUPT_CAN_1_Tmit		
	ginterruptprio_priority	39
SYSTIMER_0		
	ginterruptprio_systimer	63

Mapping IC Pins to CANbus Driver Node #0

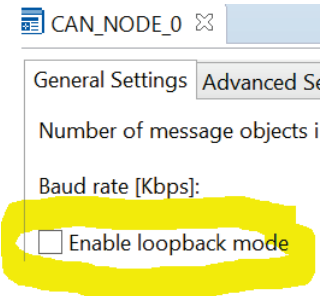
The Platform2Go board routes Xmc4200 IC pins P2.0 and P14.3 to CANbus transceiver CAN_TX and CAN_RX respectively; therefore, this needs to be set up in the Manual Pin Allocator, as shown below for Node #0. Also, as shown here, we map pins P1.4 and P2.7 for Node #1.

Resistor	XMC Pin	Signal
R3	P2.0	CAN_TX
R4	P14.3	CAN_RX

CAN_NODE_0		
	CAN Receive Pin	#19 (P14.3)
	CAN Transmit Pin	#34 (P2.0)
CAN_NODE_1		
	CAN Receive Pin	#48 (P1.4)
	CAN Transmit Pin	#35 (P2.7)

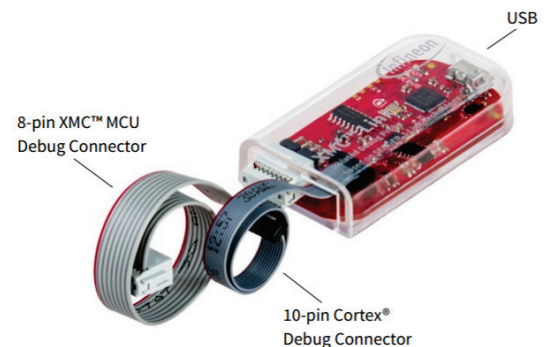
Loop Back

If you select "Enable Loop Back Mode" in the CAN_NODE_x App (route tmit to rcv internally for testing purposes), then the physical tmit/rcv pins will not be shown in the Pin Allocation dialog (since they are internally routed). After enabling loop back you can test transmit to receive without two hardware devices; in theory. Yet we don't see this and instead we see lack of ACK leading to Bus-Off, as described above (receive sees nothing). In summary, loopback does not seem to be helpful.



XMC Link Debugger

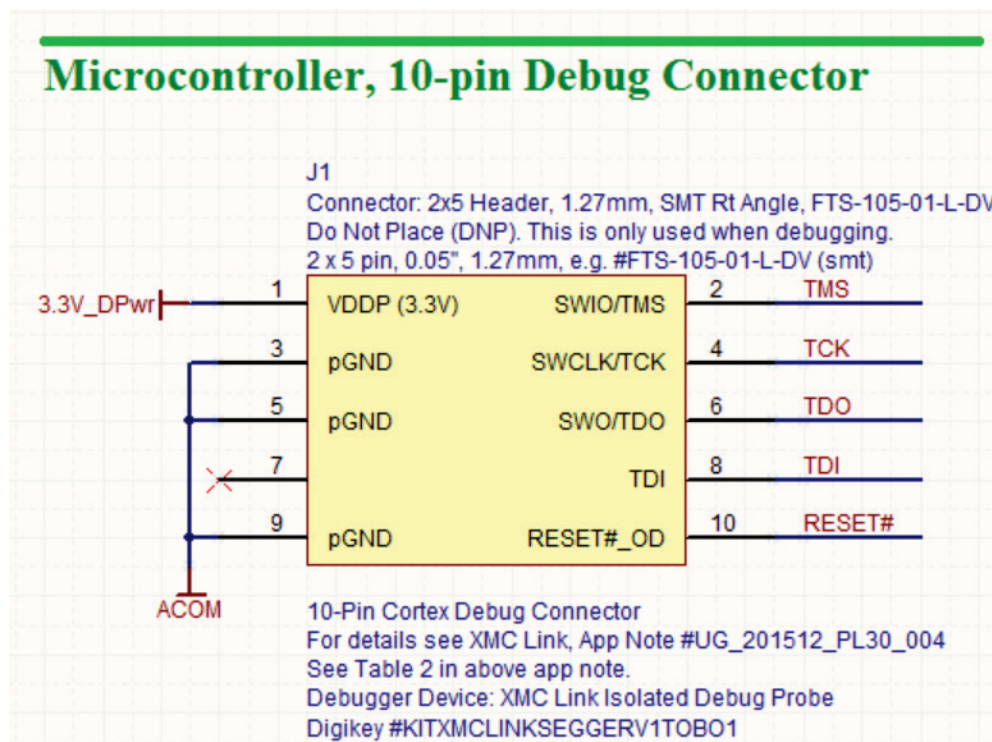
If one wants to download/debug to a PCB other than an existing reference board, then consider making use of a downloader/debugger. An example is the Infineon XMC Link (#KITXMCLINKSEGGERV1TOBO1) which supports debugging from the DAVE environment. This is very similar to the secondary Xmc4200 processor on the Platform2Go board that manages debugging of the primary Xmc4200. For details, see [Datasheet](#), [User's Manual](#) and [Product](#) page.



Alternatively, one might work with something like the Segger [Edu Mini](#) Debugger. For details, see their [User's Manual](#).

10pin Cortex Connector

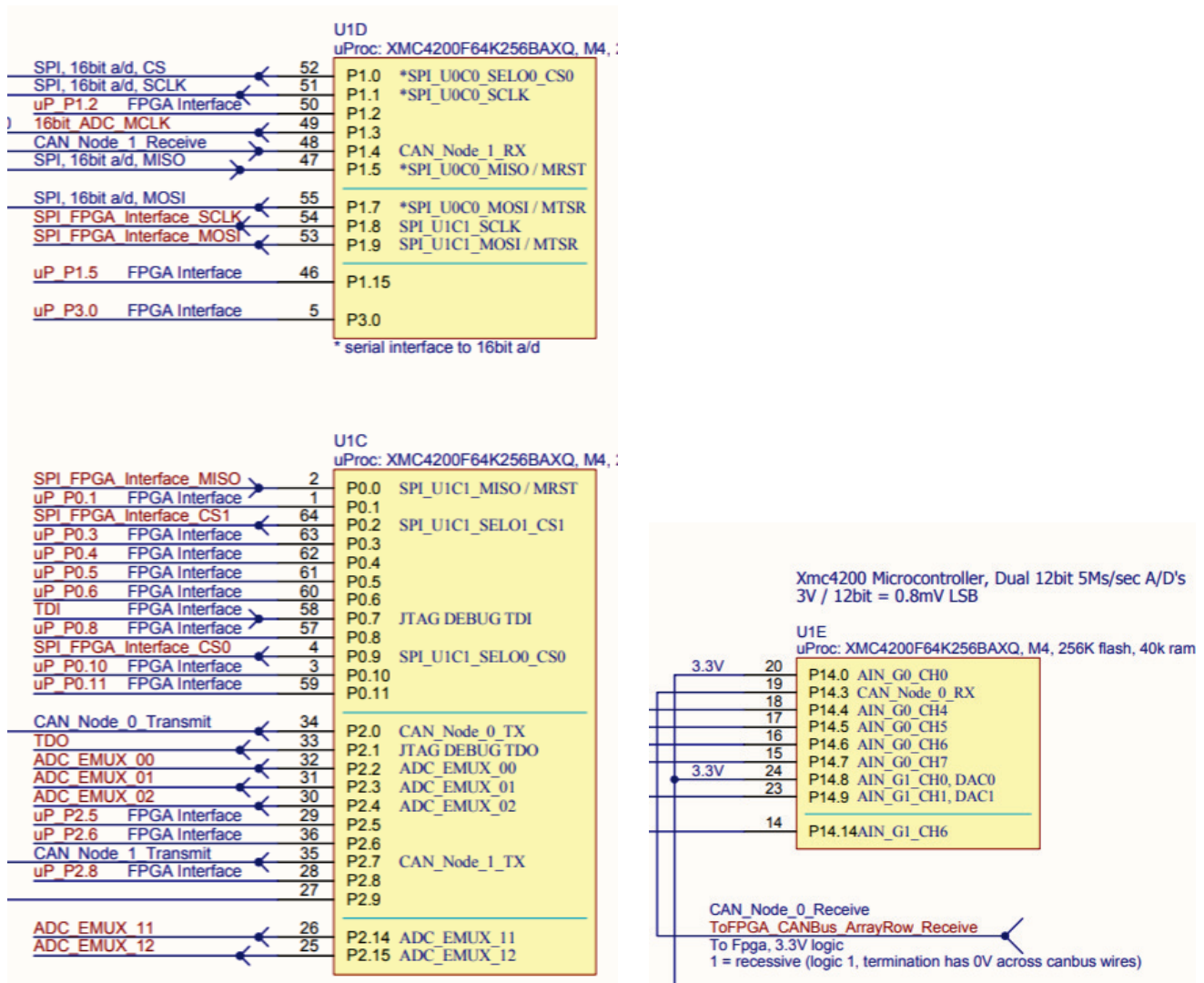
The above devices attaches to a small 10-pin connector on one's PCB, an example of which is shown below.

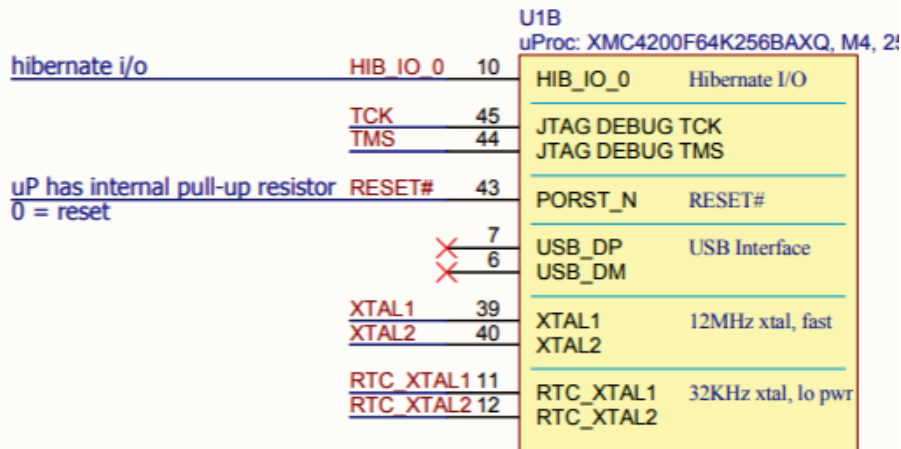
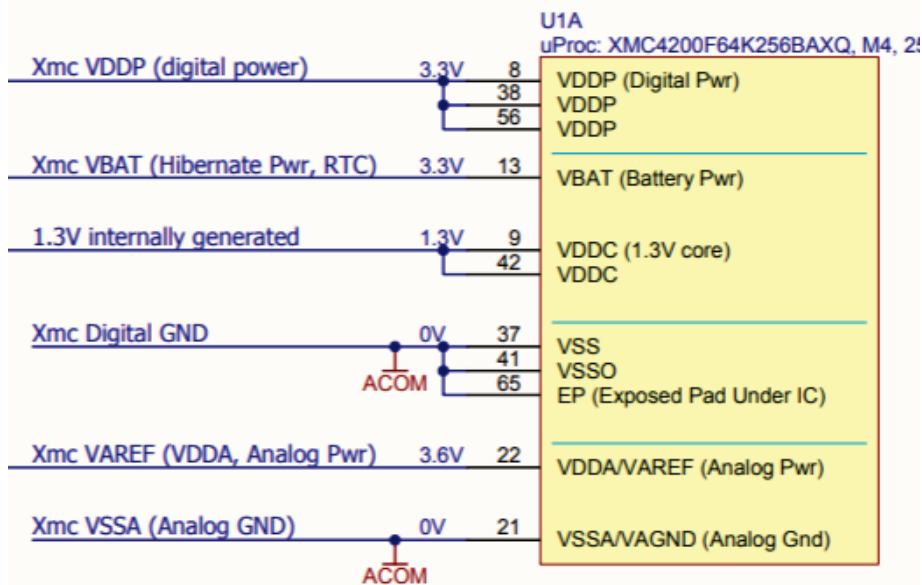


Schematics with Processor

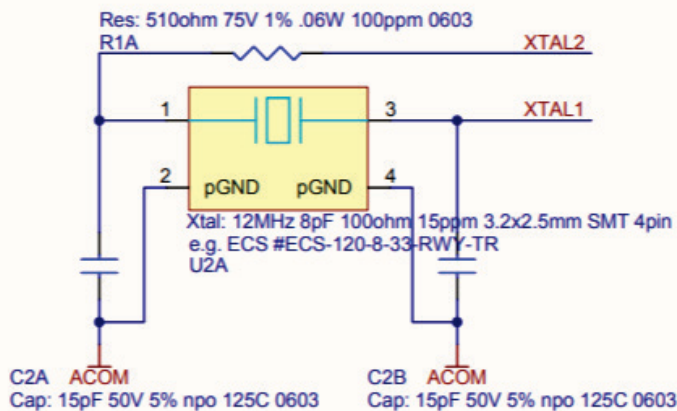
There are many schematics that include microcontrollers. For example, the Platform2Go [manual](#) contains a schematic that shows an Xmc4200 processor.

Another example is a [300W solar power DC-to-DC converter](#), shown below. For a list of components in this schematic, search "XMC4200F" in [ResearchNotes.xls](#).

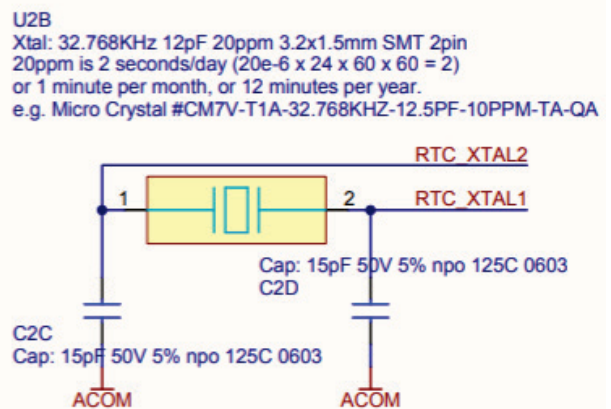




Clock (12MHz, fast, more power)



RTC (32KHz, slow, low power)



Increasing Font Size





If the font is too small in the DAVE Help Viewer (which displays .chm files) click "CTRL" and "+", as noted [here](#).

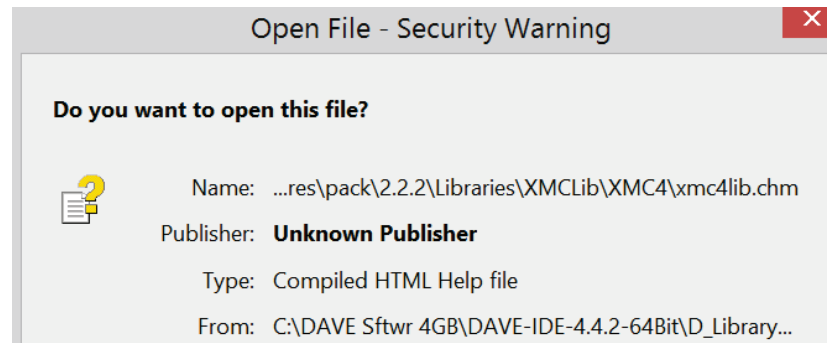
Bypassing Security with DAVE Help XMC Lib Documentation

If an alert appears that says "File Download Security Warning, ieiframe.dll is trying to save http_404_webOC to your computer and is blocked by security " while in the DAVE Help system, it might be due to DAVE software not being properly installed into the Windows ProgramFiles folder, and not having a secure area to load .chm files. For details, click [here](#).

One work-around is to open the .chm file directly (w/o security) by double-clicking on the file from the Windows Desktop (and not going through DAVE's Help system). Note the pathname referenced in the alert shown to the right. This is the one you need to click from the desktop.

XMC Lib Documentation

-  XMC1 [2.2.10]
-  XMC1 [2.2.2]
-  XMC4 [2.2.10]
-  XMC4 [2.2.2]



This PC > Local Disk (C:) > DAVE Sftwr 4GB > DAVE-IDE-4.4.2-64Bit > D_LibraryStore_4.4 > DeviceFeatures > pack > 2.2.10 > Libraries > XMCLib > XMC4



<input type="checkbox"/>	Name	Date modified	Type	Size
	inc	11/16/2020 9:55 AM	File folder	
	src	11/16/2020 9:56 AM	File folder	
	xmc4lib.chm	11/16/2020 9:56 AM	Compiled HTML Hel...	4,281 KB

Bypassing Security with Help for DAVE Apps

Another work-around is to locate the .chm file, select it, right-click Properties and click the "Unblock" button, as noted [here](#).

Notice there are version numbers in the listed DAVE Apps (e.g. "4.0.12", shown to the right). To unblock, one needs to locate these directories, drill down to the .chm file, right-click properties, and then click Unblock. If you know you are going to use 5 to 10 of these, you might want to unblock all at one time so you only need to figure this out once. To get to the directories w/ these versions, refer to the below path:

DAVE APPs

-  ACIM_FREQ_CTRL [4.0.12]
-  ACIM_FREQ_CTRL [4.0.8]

Local Disk (C:) ▸ DAVE Sftwr 4GB ▸ DAVE-IDE-4.4.2-64Bit ▸ D_LibraryStore_4.4 ▸ resources			
Name	Date modified	Type	Size
0.0.0	2/21/2018 10:56 AM	File folder	
0.1.0	2/21/2018 10:56 AM	File folder	
0.1.19	2/21/2018 10:56 AM	File folder	
0.1.20	2/21/2018 10:56 AM	File folder	

The below picture shows drilling down to .chm file:

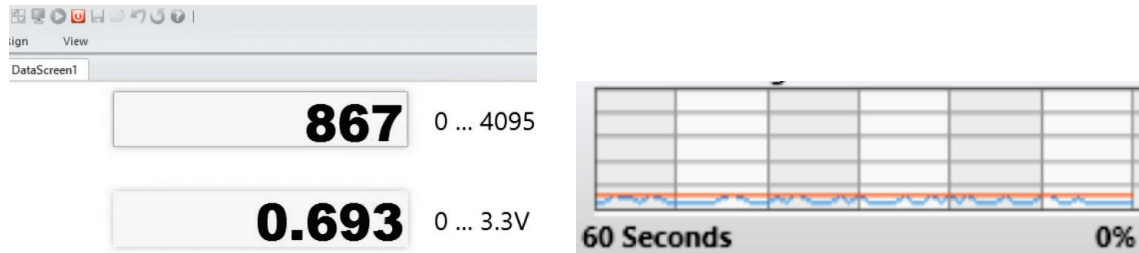
« DAVE-IDE-4.4.2-64Bit ▸ D_LibraryStore_4.4 ▸ resources ▸ 4.0.18 ▸ app ▸ ADC_SCAN ▸ 0 ▸ doc			
Name	Date modified	Type	Size
index.chm	9/25/2020 8:24 PM	Compiled HTML Hel...	

[Bypassing Security when downloading example .zip files from Infineon Website](#)

One might incur security when downloading .zip files from the Infineon website. Adding Infineon.com as a trusted URL might help. Alternatively, consider closing all your open windows and trying a different browser (e.g. try Internet Explorer instead of Google Chrome).

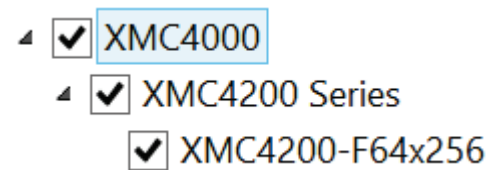
Chapter 39) Setting up a Real-Time Micrium Control Panel

Micrium software enables one to easily monitor changing real-time parameters within a DAVE program. For example, your program might read the A/D converter once a second, and Micrium can be set up to display, and plot, this value on your Windows computer. For details on how this works, see Asrain Video [#10](#).



Targeting Xmc4200 Platform2Go

The DavePrj_BB project targets (is built for) the XMC4200-F64-256 processor (not F48). This IC has 64pins and 256kb of flash memory, and is the processor on the Xmc4200 [Platform2Go](#) Board.

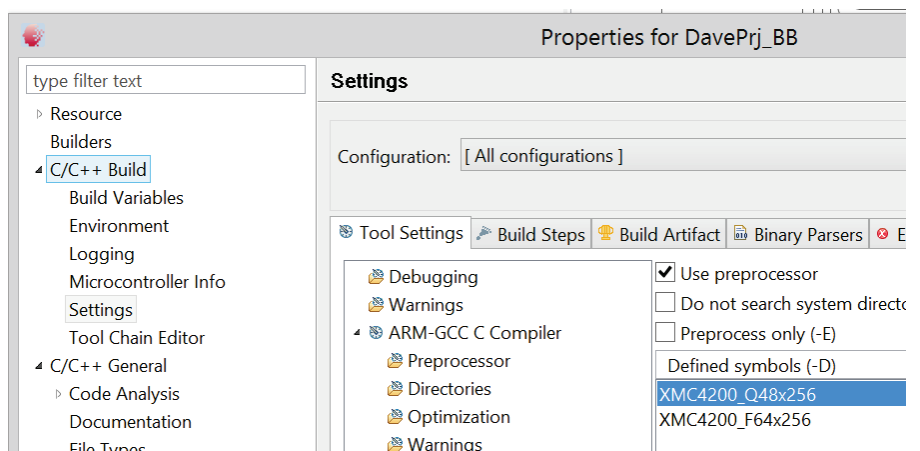


Migrating to a Different Processor

One can change the project to a different Xmc4200 processors by right-clicking on the active project and then select: Dave Project Upgrade > Device Migration > new processor. It will then make a copy of your project and placed the new project in the workspaces directory (e.g. "..BB\DavePrj_BB\DavePrj_BB_Workspace\DavePrj_BB_XMC4200-F64x256_0"). After it does this, you can exit DAVE and move this folder to a location of your choosing. The files contained in the BB_Source_Code/ folder might be gone, since those links have been broken. To place the BB files back into your project (i.e. update the links), see the above "Importing Many Files" discussion. This shows how one can import many directories and files with one operation, link to external files, and not copy source code files.

After migration, your pin assignments will probably need to be reworked, since the system has no way of knowing which pin you want to connect to.

One should also delete pre-processor symbols that no longer apply for the C compiler, C++ compiler, and Assembler.



Remember to click "Generate Code" after updating your Apps.