CrossFire IX

Freely Programmable – Programming Manual

crosscontrol

www.crosscontrol.com

Contents

Revision history

1.0	First release	CMM	2018-06-18
1.1	Updates after review, added testing section,	CMM	2018-07-04
1.2	Updates in boot loader section	CMM	2018-07-04
1.3	Additions for data logger API	CMM	2018-07-05
1.4	Data logger API now has its own document. Added info about interrupt priorities. Clarification that OEM should only put code in OEM_Init(), OEM_Execute() and OEM_MSTick(). OEM_Init() is now called last in the init sequence.	CMM	2018-08-20
1.5	Rename from CrossFireIX CANopen Freely Programmable to CrossFireIX Freely Programmable as there is no requirement to use CANopen	CMM	2018-08-21
1.6	Added chapter about performance	CMM	2018-08-24
1.7	Review	CMM	2018-09-10
1.8	Updates about timers	CMM	2018-09-25
1.9	Added info about swv trace. Improved structure.	CMM	2018-10-01
1.10	Updates after name change of some functions	CMM	2018-10-19
1.11	Added architecture overview	CMM	2018-10-24
1.12	Split of pre-processor defines between OEM and CrossControl	CMM	2018-10-26
1.13	Clarification regarding USE_PWMI	CMM	2018-11-07
1.14	Added chapters about examples and project structure.	CMM	2018-11-12
1.15	Updates after adding the drivers/target subfolder. Now Atollic 9.1.0 is recommended. Updated architectural diagram.	CMM	2018-11-27
1.16	Added chapter about controlling the I/O	CMM	2018-11-28
1.17	Added info about setting defines in Atollic	CMM	2018-11-29
1.18	Updates for the SDK 0.9 release	CMM	2018-11-30
1.19	Review	CMM	2018-12-03
1.20	Updated API documentation for FRAM and CAN	CMM	2018-12-04
1.21	Review, improved structure, added summary of more API functions	CMM	2018-12-07
1.22	Review, improved structure, updates after changes in Util_ function API	CMM	2018-12-10
1.23	Review	CMM	2018-12-11

1.24	Added Basic I/O Example	CMM	2018-12-12
1.25	Added documentation for frequency and encoder inputs	CMM	2018-12-17
1.26	Watchdog is now a separate driver, not included in Core API	CMM	2018-12-19
1.27	Updated tools section	CMM	2018-12-21
1.28	Added more details of the I/O example. Updated info about CAN tests.	CMM	2019-01-02
1.29	Minor changes after review from ERJ	CMM	2019-01-17
1.30	Updates for version 1.21 of core api	CMM	2020-03-16

1. Introduction

CrossFire IX is the 2nd product on CrossControl's new I/O Controller platform. It is a compact 32-bit I/O module, designed for advanced hydraulics control in agricultural and construction equipment. It offers 22 I/O channels, versatile and configurable in software.

CrossFire IX Freely Programmable SDK is a package containing drivers and documentation to make it possible for OEMs to program the CrossFire IX in C/C++. To make the programming of the CrossFire IX as easy as possible, an extensive library (IX Core API) is available for the OEM. This library contains functions for controlling the inputs and the outputs of the CrossFire IX in an easy way. The library also contains a number of utility functions.

The CrossFire IX exists in two versions: "CANopen" and "Data Logger Edition". The CANopen name is a bit misleading as there is no requirement to run CANopen on this version. However, CrossControl has a ready-made CANopen slave software for this unit.

The "Data Logger Edition" contains some additional functionality: an RTC, an external FLASH memory and a Wi-Fi module. To use these functions, the Data Logger API is used. This API cannot be used with the CANopen version of the unit. Please read the document "CrossFire IX - Freely Programmable - Data Logger Edition - Programming Manual.docx" for more information.

2. Validity

This manual is valid for the 1.21 release of the CrossFire IX Freely Programmable SDK.

3. Licensing

The CrossFire IX Freely Programmable SDK is free to use and modify when running on CrossControl hardware. See the trademark section for details.

The CANopen example is using a CANopen stack from SYS TEC. If this stack will be used, it is necessary to buy a license for the CANopen stack separately from SYS TEC.

4. Development environment

There are two alternative development environments for the CrossFire IX tested by CrossControl:

- Atollic TrueSTUDIO for STM32 9.1.0 is a free IDE based on Eclipse/GCC.
- IAR Embedded Workbench (EWARM) 7.40. A licence for the IAR compiler must be bought separately from IAR.

CrossControl recommends Atollic TrueSTUDIO mainly due to that the CrossControl examples are made for this environment and that it is free.

5. Debugging

It is highly recommended use an IAR I-jet/Segger J-link/ST-link V2 or similar debug probe for program download and debugging. The CrossFire IX has a standard 10 pin JTAG/SWD debug connector. It is necessary to buy a CrossFire IX without casing to be able to reach the debug connector.

The processor used in CrossFire IX supports both JTAG and SWD. When using SWD a number of additional features compared to JTAG are available.

6. Boot Loader

CrossFire IX is running a boot loader that makes it possible to upgrade the CrossFire IX firmware over the CAN-bus (including the boot loader itself). The boot loader is only delivered as a binary. There are no pins in the connector to perform firmware upgrade not using the bootloader.

To make it possible to go to program upgrade mode it is necessary that the application has functionality for jumping to the boot loader at some trigger, for instance when receiving a certain CAN message or when activating a digital input. Otherwise there will be no way to trigger the execution of the boot loader. This is done by calling the function Util_JumpToBoot() when the trigger is received. All example projects have this functionality.

It is also necessary that the application flags to the boot loader that the application is working properly. Failing to do so will make the unit stay in the boot loader the next cold-boot after a software upgrade. This is done by calling the function Util_WriteApplicationOKtoFRAM(). This function is used to make sure that if upgrading to a non-working application, the bootloader will still be reachable. This call is normally done in the CrossControl init code.

The boot loader uses the CANopen protocol. This will be the case even if the main application is using another CAN protocol like J1939. The node-id and baud-rate used by the boot loader is read from the FRAM memory. There is also a mode where the node-id is read from the digital input pins of the CrossFire IX. This means that even if CANopen is not used at all in the main application, it is important that the FRAM address where node-id and baud rate is read is assigned appropriate values, otherwise the boot-loader might not work. If changing these values, use the access functions in the Core API because the data also contains a checksum that needs to be updated.

Definition	**FRAM address**	**Access Function**
FRAM_NODE_ID	0x0DA0	`void Set_V_0x2010_Node_ID_Value(unsigned char value)`
FRAM_BAUDRATE	0x0DA4	`void Set_V_0x2011_Baudrate_Index(unsigned char value)`

FRAM_NODE_ID_PIN_FILTER	0x0DA8	`void Set_NodeIdPinFilter(unsigned short value)`

The boot loader is located in the bottom of the FLASH memory. Therefore it is necessary that the user application start address is set to 0x08006000.

FLASH Memory Address	**Description**
0x08000000 - 0x08005FFF	Boot Loader Area (24KB)
0x08006000 - 0x0803FFFF	Application Area (256 – 24 = 232KB)

The supplied linker files for IAR/Atollic sets up the start address to 0x08006000.

Programming the CrossFire IX (using the boot loader) is done with the CrossFire IX Tool for Windows. Please read the tools section for more information.

7. Folder structure

The CrossFire IX CANopen Freely Programmable SDK will be delivered in a .zip file. The .zip contains the following folders:

Delivery – Contains already built binaries for the CANopen version. Used for reference and test.

Documentation – Programming documentation.

Doxygen – Configuration files for Doxywizard that can be used to generate API documentation. There is also HTML folders containing already generated HTML Doxygen documentation.

Reference Material – Reference manuals for the processor and the most important circuits on the CrossFire IX PCB.

Tools – Contains the IX Tool and the SakNfo PC tool.

Src – Contains the actual source code.

Src/Application – High level application code.

Src/CANopen-stack – Contains the SYS TEC CANopen stack and files configuring the stack and connecting the stack to the application.

Src/Drivers – Drivers for the CrossFire IX hardware. Drivers located in this folder are written by CrossControl.

Src/Drivers/Target/CMSIS – Contains CMSIS (Cortex Microcontroller Software Interface Standard) start-up and init code.

Src/Drivers/Target/STM32F37x_StdPeriph_Driver - Contains ST Std peripheral drivers.

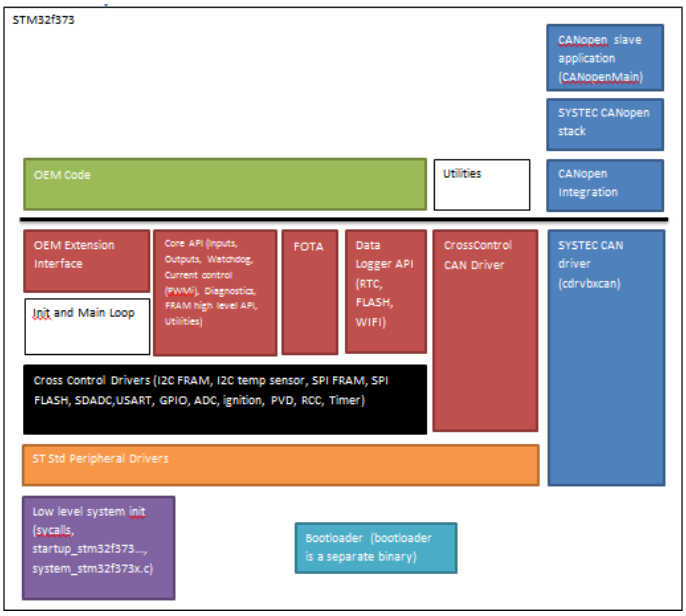
Src/Project – Contains project files for IAR Embedded Workbench 7.40 and Atollic TrueSTUDIO.

Src/Examples – Example applications.

Src/CoreAPI – Header files for the CrossFire IX Core API.

Src/Utilities – Additional utility files.

8. Architectural Overview



The OEM should only modify the block marked as “OEM Code” in the diagram above. To make sure the OEM code is executed at the right time, the OEM should include the “OEMExtension.h” and implement the functions in the OEM Extension Interface. These functions are called by the CrossControl init and main loop code.

The OEM Extension Interface contains the following functions:

- OEM_Init() – Code to set up inputs and outputs according to the needs of the OEM as well as OEM specific init code.
- OEM_MsTick() – A callback called every millisecond where the OEM can add any timing related code. This code is called from interrupt so execution must be very fast.
- OEM_Execute() – A function called from the main loop where the OEM can add its own code that will be executed cyclically.

```
1  /*
2  Example application for CrossFire IX - Read I/O and send on CAN
3
4  (C)2018 CrossControl, only to be used on CrossControl hardware
5  */
6
7  #include "can.h"
8  #include "utility.h"
9
10 #include "OEMExtension.h"
11
12 #include "inputmanager.h"
13 #include "outputmanager.h"
14 #include "currentcontrol.h"
15
16 static int ledTickMS = 0;
17 static int ledFreq = 1000;
18
19 const static int SEND_INTERVAL_MS = 1000;
20
21 static unsigned long sendMessageTimer = 0;
22
23 // As this function is called from an interrupt, make sure to do no lengthy operations here!
24 void OEM_MSTick(void)
25 {
26   ledTickMS++;
27   if(ledTickMS > (ledFreq / 2))
28   {
29     Util_SetGreenLED(TRUE);
30   }
31
32   if(ledTickMS >= ledFreq)
33   {
34     Util_SetGreenLED(FALSE);
35     ledTickMS = 0;
36   }
37 }
38
39 // Initialize CAN and I/O
40 void OEM_Init(void)
41 {
42   CAN_Config(CAN_1000, MODE_STANDARD);
43
44   OutputManager_SetMode(OUTPUTCHANNEL_0, MODE_DIGITALOUT);
45   OutputManager_SetMode(OUTPUTCHANNEL_1, MODE_DIGITALOUT);
46
47   OutputManager_SetMode(OUTPUTCHANNEL_2, MODE_PWM);
48   OutputManager_SetPWMFrequency(OUTPUTCHANNEL_2, 50);
49
50   OutputManager_SetMode(OUTPUTCHANNEL_3, MODE_PWM);
51   OutputManager_SetPWMFrequency(OUTPUTCHANNEL_3, 400);
```

The OEM has access to all functions in the Core API, the Data Logger API (only for the Data Logger Edition) and the CrossControl CAN Driver. If needed the OEM can access the CrossControl Drivers layer directly (or even the ST Std Peripheral Drivers or the hardware), but this is not recommended as this might give problems if upgrading the a newer versions of the CrossFire IX Freely Programmable SDK. Please contact CrossControl in case you need to change any code outside the OEM Code box.

The parts regarding CANopen is optional and is only needed if CANopen support is needed (SYS TEC CAN driver, SYS TEC CANopen stack, CANopen integration and CANopen slave application). If using these parts a separate license from SYS TEC is needed.

It is of course possible to add CANopen support with another CANopen stack but it is possible to save a lot of time using the already available SYS TEC CANopen integration.

9. Getting started with Atollic TrueSTUDIO for STM32

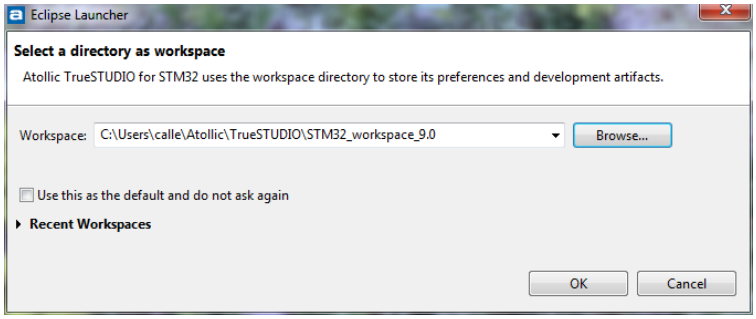
“Atollic TrueSTUDIO for STM32” is a commercially enhanced C/C++ IDE based on open source components”. This means that Atollic TrueSTUDIO for STM32 is basically Eclipse/GCC/CDT/GDB with enhanced support for STM32 processors.

9.1. Opening and building a project

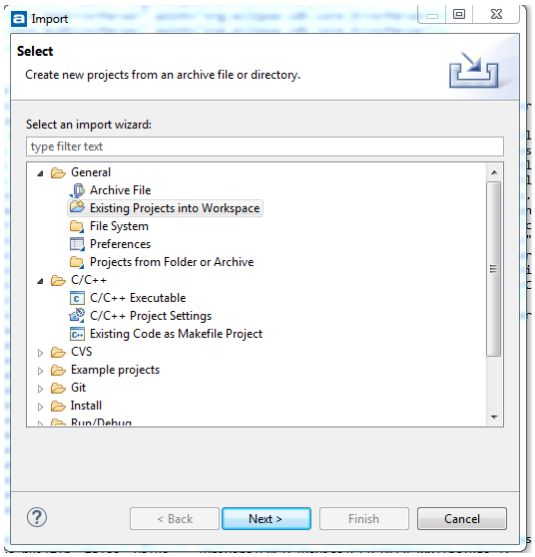
Open Atollic TrueSTUDIO for STM32 9.1.0 or later

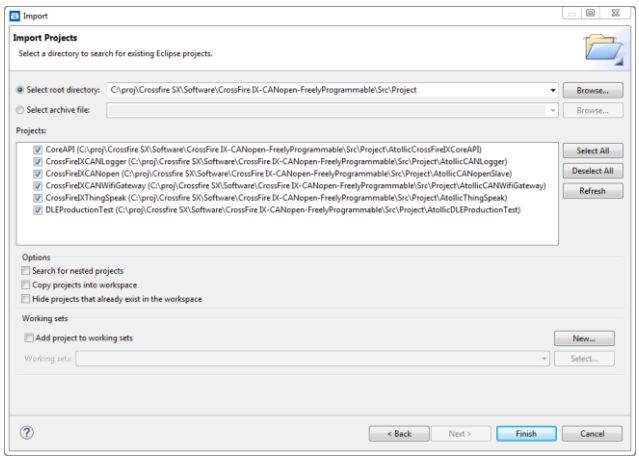
Select a workspace directory. A workspace in Eclipse is not the same as a workspace in IAR or Visual Studio. You never check out a workspace from version control system. A workspace in

Eclipse is a local folder on your computer where you keep your personal Eclipse settings. You can have several workspaces and switch between them with “File/Switch Workspace”. This can be handy if you are working with several languages like Java and C++ and want to have separate settings between the two. When switching workspace, Eclipse will shut down and restart using the new workspace.

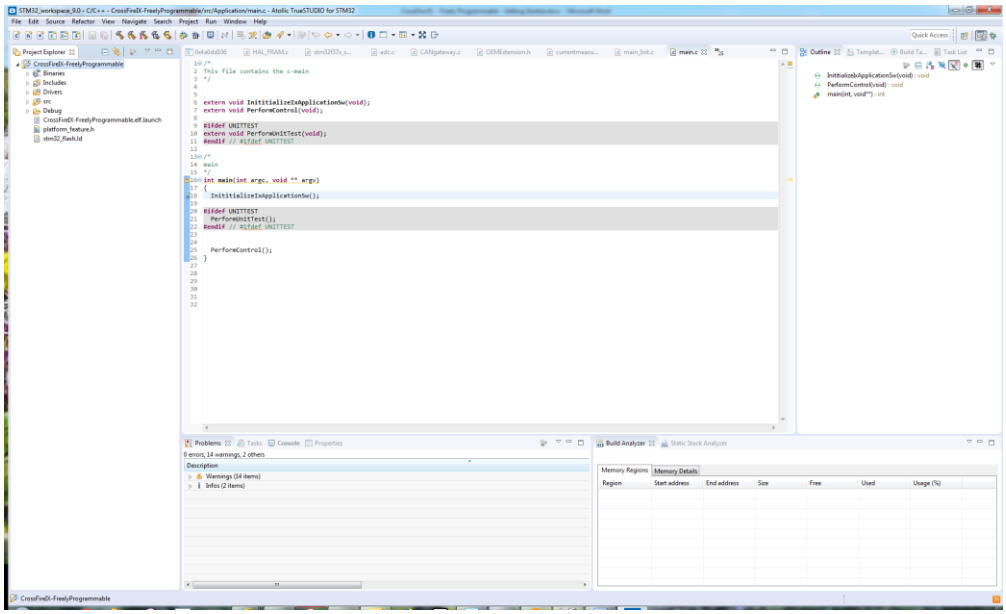


Now it is time to open the CrossFire IX project. There is no function called “open project” in Eclipse. Instead you should now import the project into your workspace. Select File/Import and then “Existing Projects into Workspace” from the import dialog. Browse for the folder containing the CrossFire IX projects.





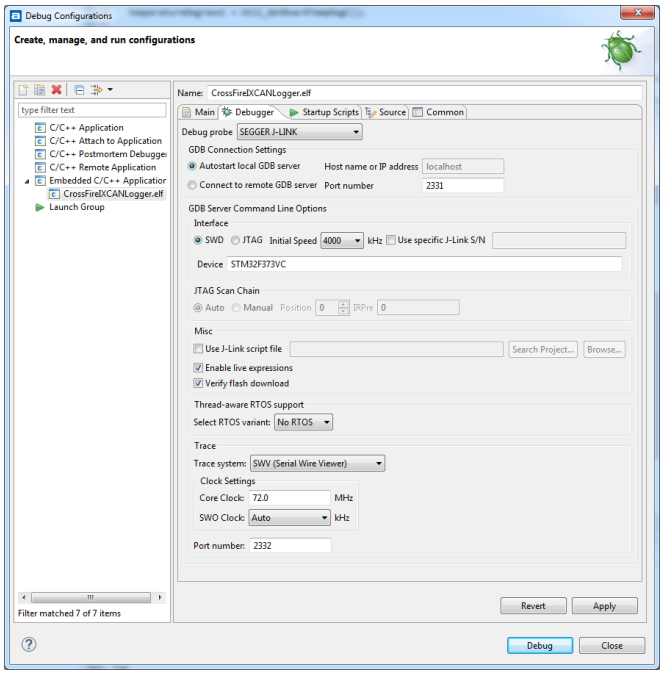
Make sure that the checkbox “Copy projects into workspace” is not checked. This means that the projects are not copied to your workspace, only links are created. You can now select which projects you want to import. Click Finish and your project/projects will be imported.



To build the project use Project/Build Project.

In case you want to rebuild everything select Project/Rebuild Project. In some cases you might also want to run Project/Clean project to make sure everything is rebuilt.

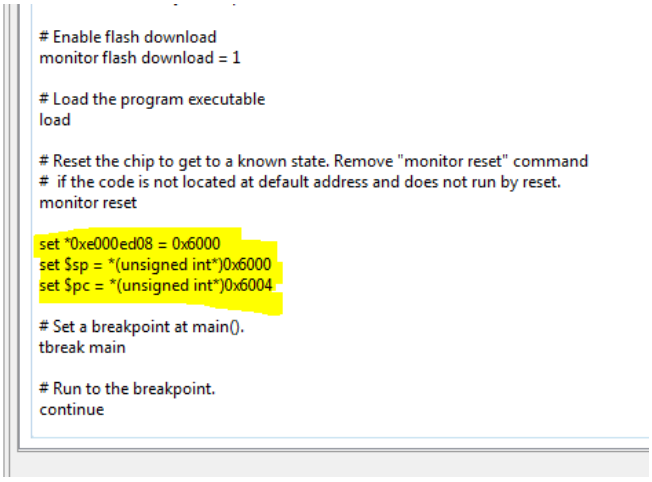
If you have a debug-probe connected you can now select Run/Debug to download the application to the CrossFire IX and start debugging. You have to set up a debug configuration to be able to debug the project.



A debug configuration basically sets up which type of debugger will be used an in which mode (JTAG/SWD).

To make it possible to download and debug code directly from the debugger, the following code has to be added to the “Target Software Startup Scripts” available at debug configurations. It will set up the stack pointer, the program counter and the interrupt vector register to the start of the application instead of to the start of the boot loader.

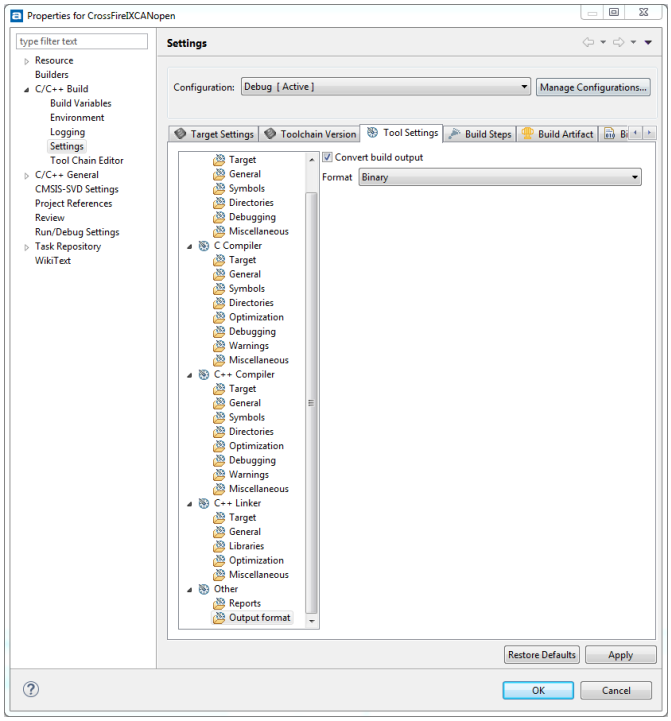
```
set *0xe000ed08 = 0x6000
set $sp = *(unsigned int*)0x6000
set $pc = *(unsigned int*)0x6004
```



The CrossFire IX example projects have two build configurations. There is a debug and a release configuration. You can switch configuration with Project/Manage Build Configurations...

You can reach the most important settings from Project/Build Settings...

To create a .binary file that can be downloaded with the bootloader, activate “Convert build output” in the Output format settings.



9.2. Atollic TrueSTUDIO for STM32 settings for CrossFire IX

The following settings have been made in the example projects:

The linker script stm32_flash.ld has been modified so that the first 24KB of the FLASH is reserved for the bootloader.

```
/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x08006000, LENGTH = 232K
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 32K
  MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
}
```

There is also a specific section to make the version number located in a specific address in flash memory:

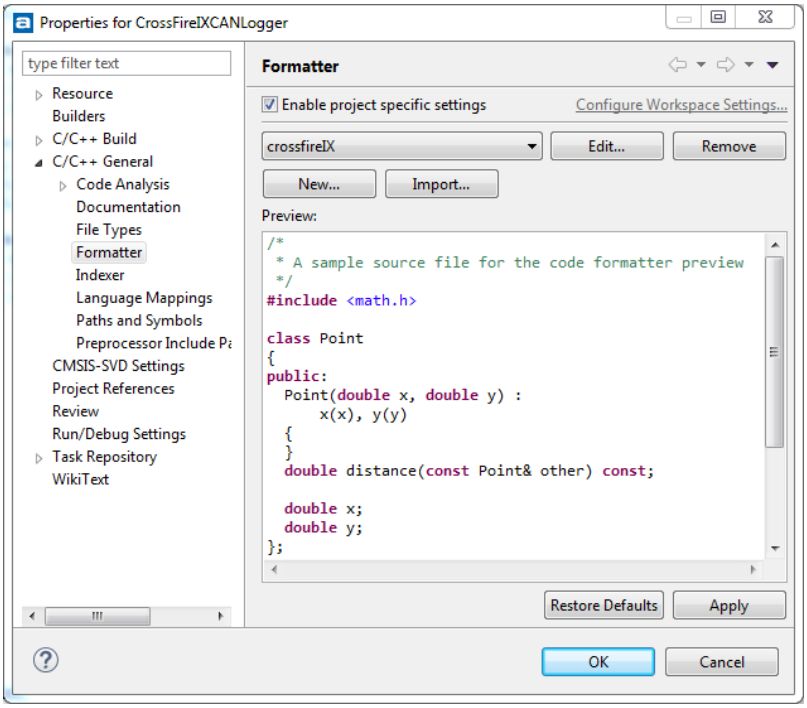
```
.VERSION_SECTION 0x08006200 :
{
  KEEP(*(.VERSION_SECTION)) /* keep my variable even if not referenced */
} > FLASH
```

Optimization for the debug build is set to (-O0) and for the release build is set to (–o2).

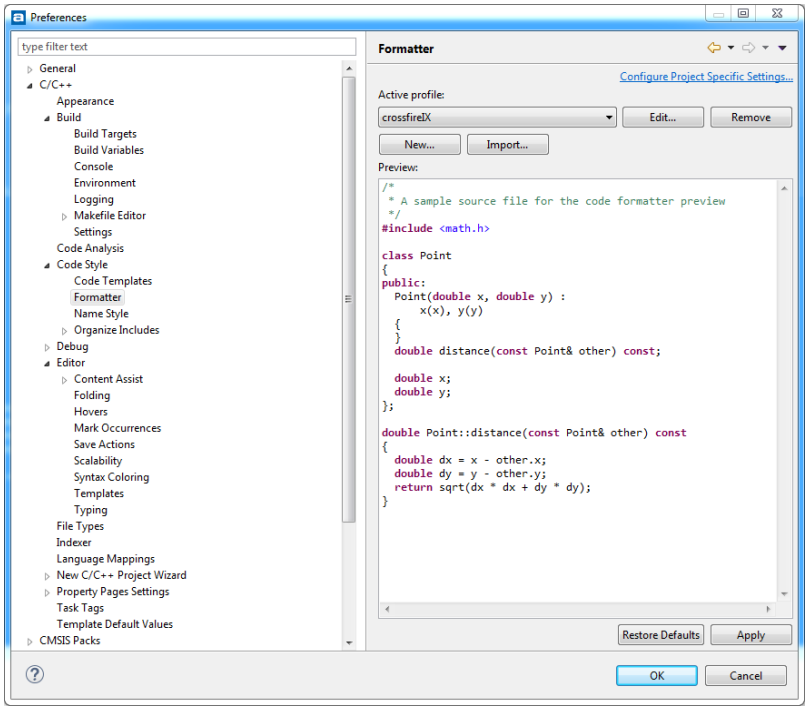
C/C++ Standard is set to C11/C++11.

9.3. Common operations in Atollic TrueSTUDIO

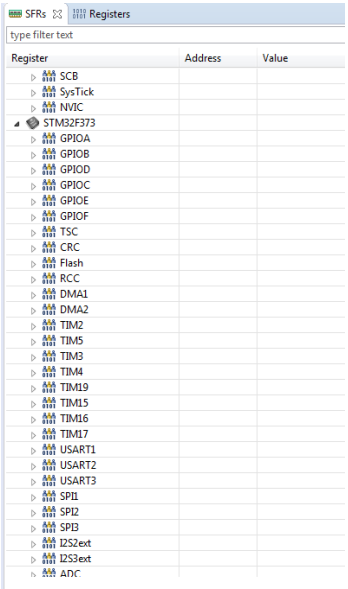
There is a formatting file in .xml format that can be imported to make sure to get the same formatting as is used previously in the project. The file is located under src/project.



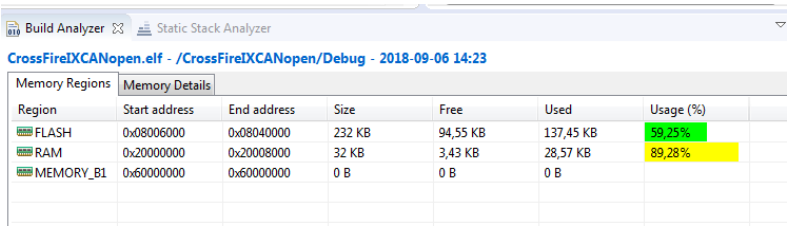
It is also possible to set the formatting globally for all projects.



For advanced debugging there is access to the processor SFRs and registers.

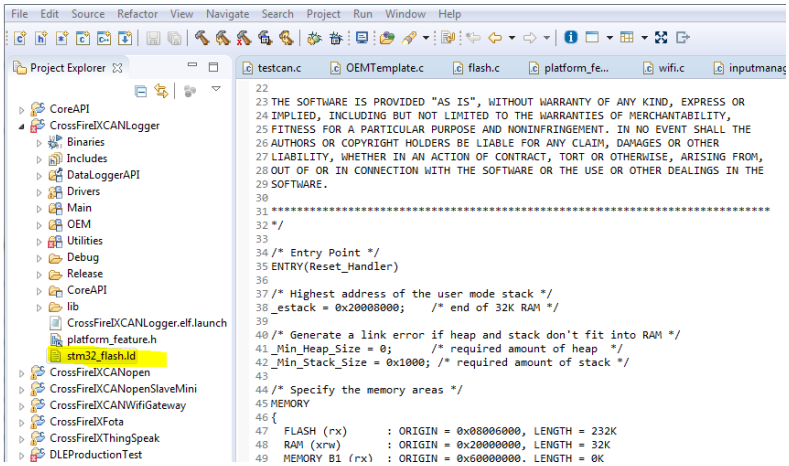


The amount of RAM/FLASH used can be seen in the “Build Analyzer”

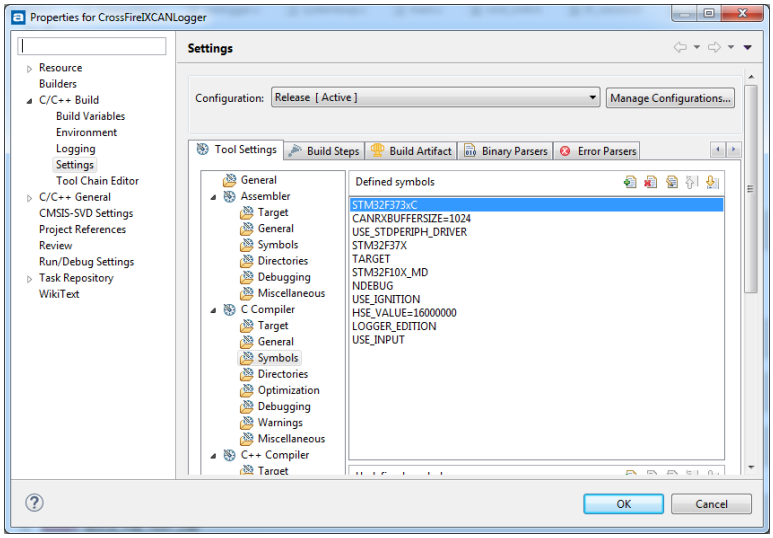


Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH	0x08006000	0x08040000	232 KB	94,55 KB	137,45 KB	59,25%
RAM	0x20000000	0x20008000	32 KB	3,43 KB	28,57 KB	89,28%
MEMORY_B1	0x60000000	0x60000000	0 B	0 B	0 B	

Memory settings is done directly in the linker file stm32_flash.ld



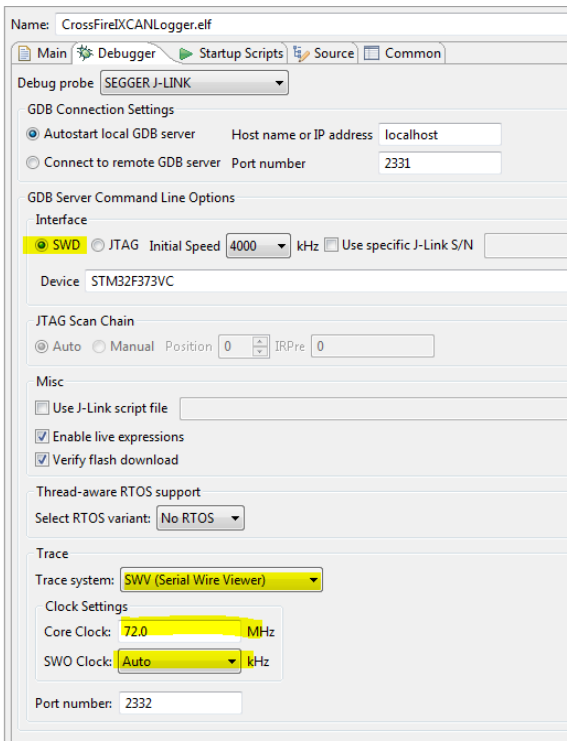
Defines can be set in project properties/C/C++ Build/Settings/C Compiler/Symbols. If C++ is used, make sure to go to C++ Compiler instead of C Compiler.



9.4. Using SWV Trace

To be able to perform advanced debugging you have to activate SWD mode and also set up Trace over SWV.

Select Run/Debug configurations. Select the Debugger tab.

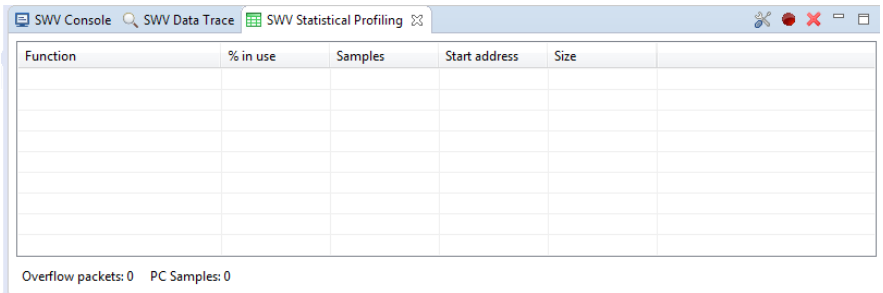


Make sure that “SWD” is selected as Interface and Trace system is set up to “SWV”. Check that Core Clock is set up to 72 Mhz.

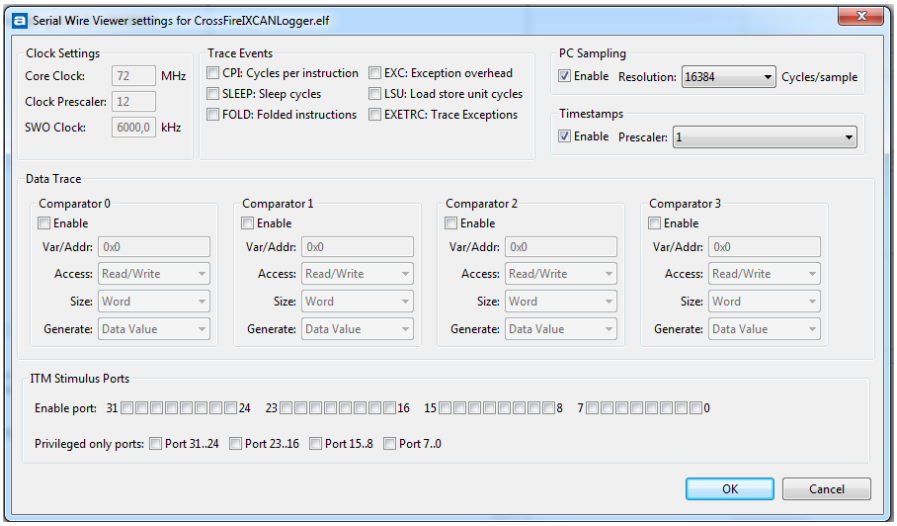
9.4.1. SWV Statistical Profiling

Statistical profiling is very useful to find performance bottlenecks in the code.

Activate the “SWV Statistical Profiling” window and press the configure button.

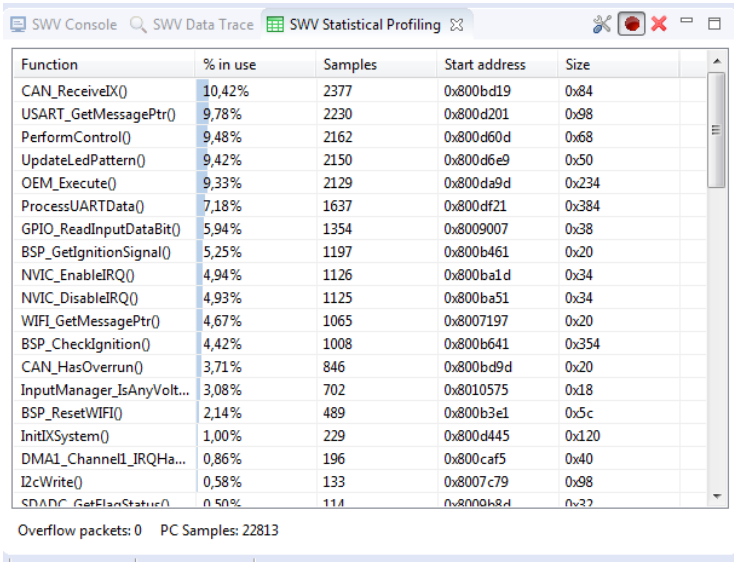


Activate “PC Sampling”.



Now press the “record” button in the “SWV Statistical Profiling” window.

After running the project for a while and after selecting “run/suspend “you will see something like the following:

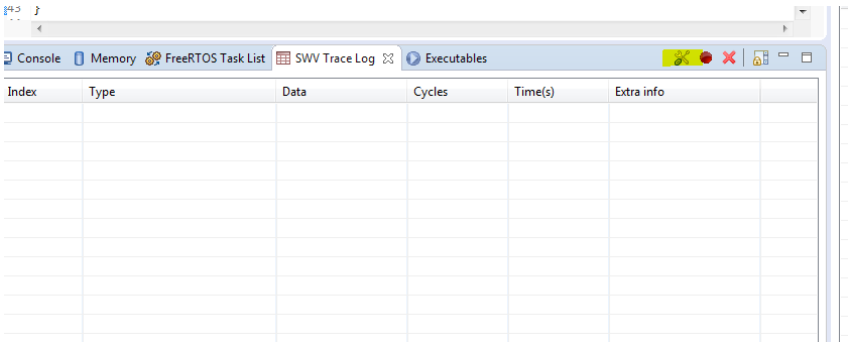


Function	% in use	Samples	Start address	Size
CAN_ReceiveIX()	10,42%	2377	0x800bd19	0x84
USART_GetMessagePtr()	9,78%	2230	0x800d201	0x98
PerformControl()	9,48%	2162	0x800d60d	0x68
UpdateLedPattern()	9,42%	2150	0x800d6e9	0x50
OEM_Execute()	9,33%	2129	0x800da9d	0x234
ProcessUARTData()	7,18%	1637	0x800df21	0x384
GPIO_ReadInputDataBit()	5,94%	1354	0x8009007	0x38
BSP_GetIgnitionSignal()	5,25%	1197	0x800b461	0x20
NVIC_EnableIRQ()	4,94%	1126	0x800ba1d	0x34
NVIC_DisableIRQ()	4,93%	1125	0x800ba51	0x34
WIFI_GetMessagePtr()	4,67%	1065	0x8007197	0x20
BSP_CheckIgnition()	4,42%	1008	0x800b641	0x354
CAN_HasOverrun()	3,71%	846	0x800bd9d	0x20
InputManager_IsAnyVolt...	3,08%	702	0x8010575	0x18
BSP_ResetWIFI()	2,14%	489	0x800b3e1	0x5c
InitIXSystem()	1,00%	229	0x800d445	0x120
DMA1_Channel1_IRQHa...	0,86%	196	0x800caf5	0x40
I2cWrite()	0,58%	133	0x8007c79	0x98
SDADC_GetFlagStatus()	0,50%	114	0x8009b8d	0x32

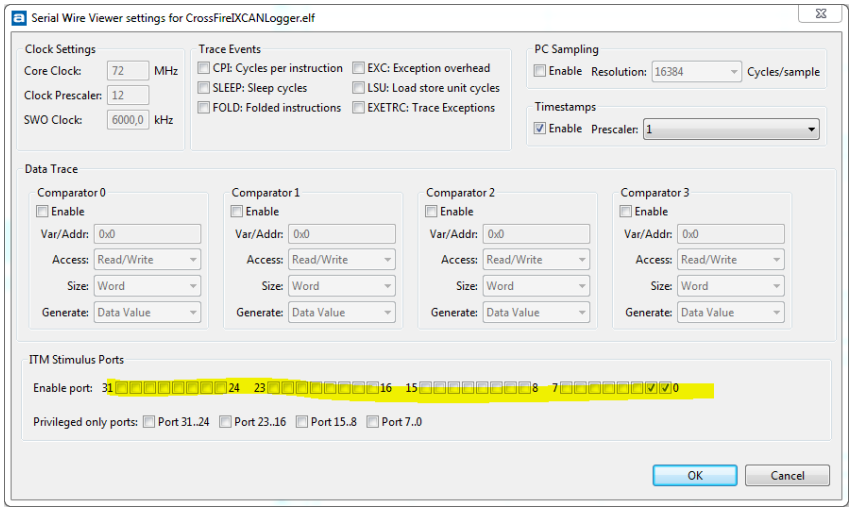
9.4.2. SWV Trace Log

Another useful debugging technique is the SWV trace log. By adding calls to ITM_Port32() macro defined in defines.h you can get close to realtime debug information in the SWV trace log.

There are 32 trace channels that can be used. To make this work you need to enable the channel(s) you like to use.



Select the configure button in the SWV Trace Log window and enable the desired channel in the settings window. Also activate “Timestamps”. Close the settings window. Enable tracing by pressing the ”Record” button.



Now you can add calls to your code

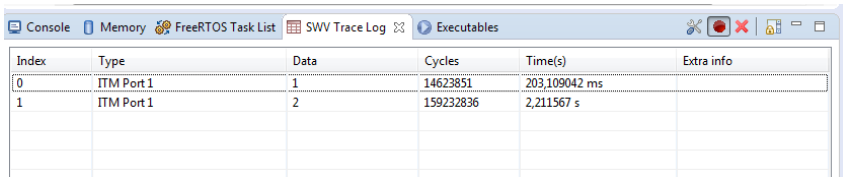
```
#ifdef __GNUC__
int main(int argc, char ** argv)
#else
int main(int argc, void ** argv)
#endif
{
  InititializeIxApplicationSw();

  ITM_Port32(1) = 1;

  I2C_WriteTest();

  ITM_Port32(1) = 2;
```

That will result in the information below in the SWV Trace Log.

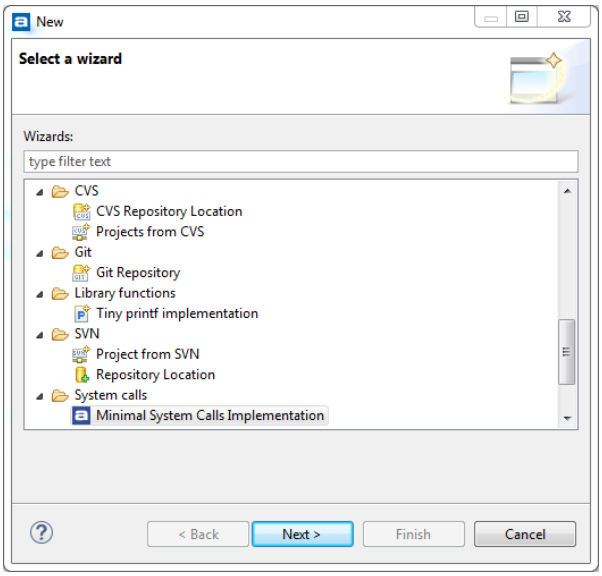


9.4.3. Redirecting printf to SWV Console

It is possible to redirect all printf calls to the SWV console. This can be very handy while debugging.

Make sure SWD debugging is enabled (see above) and that SWV is selected as trace system.

Add a syscalls.c file to the project (select Minimal System Calls Implementation) from the File/New/Other wizard.



Modify the _write function to write to write to ITM

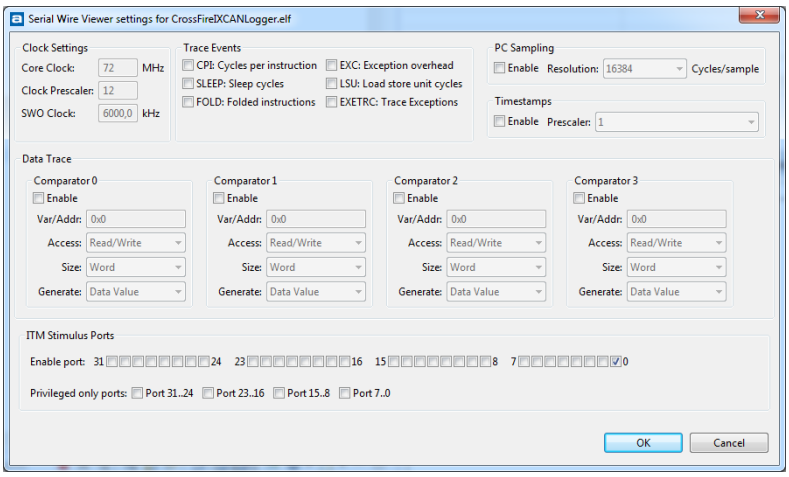
```
int _write(int32_t file, uint8_t *ptr, int32_t len)
{
  int i=0;
  for(i=0 ; i<len ; i++)
    ITM_SendChar(*ptr++);
  return len;
}
```

To make it possible to use ITM_SendChar add

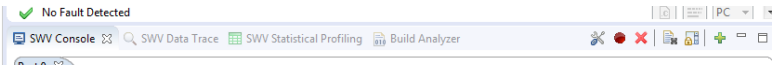
```
#include "stm32f37x.h"
```

In the beginning of the syscalls.c

Make sure that ITM port 0 is enabled



When program is in pause mode, press the “record” button in the SWV Console.



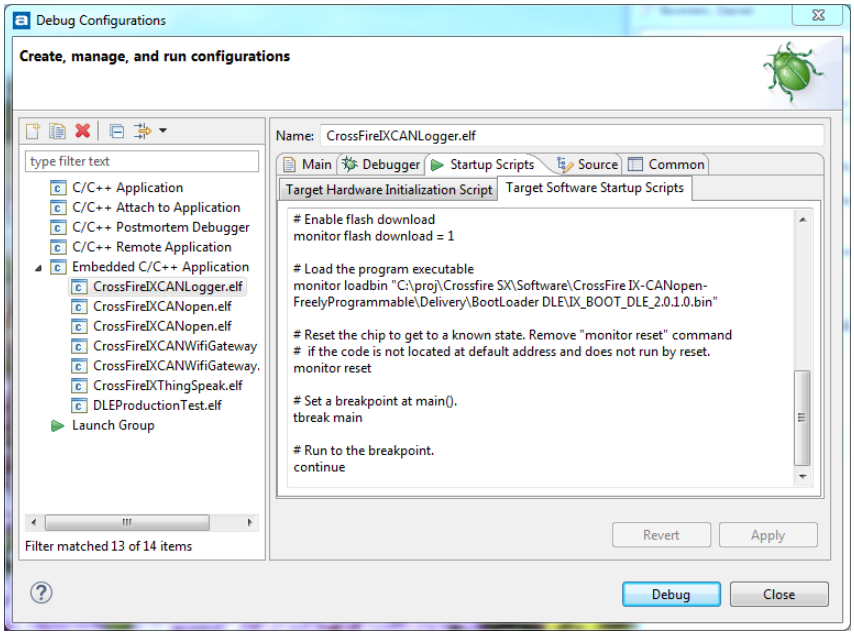
When resuming program, printf outputs will be seen in the SWV Console.

Note that printf is buffered so it is needed to add an printf("\r\n") to actually send data to the output window.

Note that all example projects already have this functionality added.

9.5. Downloading binaries

Downloading a binary (like the bootloader) can be performed by adding the following to the "target software startup scripts" - monitor loadbin "C:\proj\Crossfire IX\Software\CrossFire IX-CANopen-FreelyProgrammable\Delivery\BootLoader DLE\IX_BOOT_DLE_2.0.4.0.bin" (of course you need to adjust the path according to your needs).



9.6. Additional tips for using Atollic TrueSTUDIO for STM32

- Projects can be located in your local workspace folder but does not need to. Generally it is better to not keep your projects into your workspace folder.
- A project in Eclipse will automatically include all subfolders created under the project folder. However, the folders do not need to be included into the build process. Removing a folder from a project in Eclipse (excluding virtual folders) means that the folder will also be removed physically from the disk. It is also possible to add links to files and folders. If removing a link, the physical file will not be removed.
- Make sure you understand the difference between folders and virtual folders. A virtual folder does not exist on your disk but only in Eclipse.
- There is no explicit way to save project settings in Eclipse. Changes in project settings are automatically saved. Project file must be writable to make it possible for Eclipse to save your settings. There is no warning if settings cannot be saved.

- Eclipse uses the concept of “perspectives”. A perspective is a set of windows suitable for a certain task. There is a C++ perspective and a debug perspective but more perspectives can be defined. A handy feature is the “reset perspective” function which can be found under Window/Perspective/Reset perspective. This is very good if closing a window by mistake.
- Adjusting formatting settings is done in Project properties/C/C++ General/Formatter/Configure Workspace Settings. Note that you need to create a new profile to be able to adjust the settings as the built in profiles cannot be modified. After correct tab settings has been adjusted, the command “source/correct indentation” can be used (ctrl+i).

9.7. Common problems and solutions

- When working with a project, make sure that the project files are writeable; otherwise changes in project settings might not work and will not be saved even if making files writable at a later stage.
- Run/Debug does only work if there is already a debug configuration available. A debug configuration can be set up with “Run/Debug configurations…”.
- To decrease build time, make sure that “Enable parallel build” is enabled in C/C++ build/Behaviour.
- If application does not start as expected, check that the bootloader has not been overwritten or that FRAM settings for the bootloader has been corrupted.
- If debugging does not work properly, try disconnecting the debugger from USB and also the CrossFire IX from power. Connect back again. Also make sure that the correct binary is selected under “Debug configurations”.
- In some cases a “clean all, rebuild all” sequence is needed.
- If files within a folder does not build, check the “Exclude resource from build” setting in Properties C/C++ build.
- Note that many settings are separate for .c and .cpp files and also for different configurations (debug/release).
- If installing a new version of Atollic TrueSTUDIO, make sure to create a new workspace for that version. Incompability problems are common if using the same workspace folder for several versions of TrueSTUDIO.

10. Getting started with IAR Embedded Workbench

IAR Embedded Workbench is a commercial IDE from IAR systems. A licence for the IAR IDE must be bought separately from IAR.

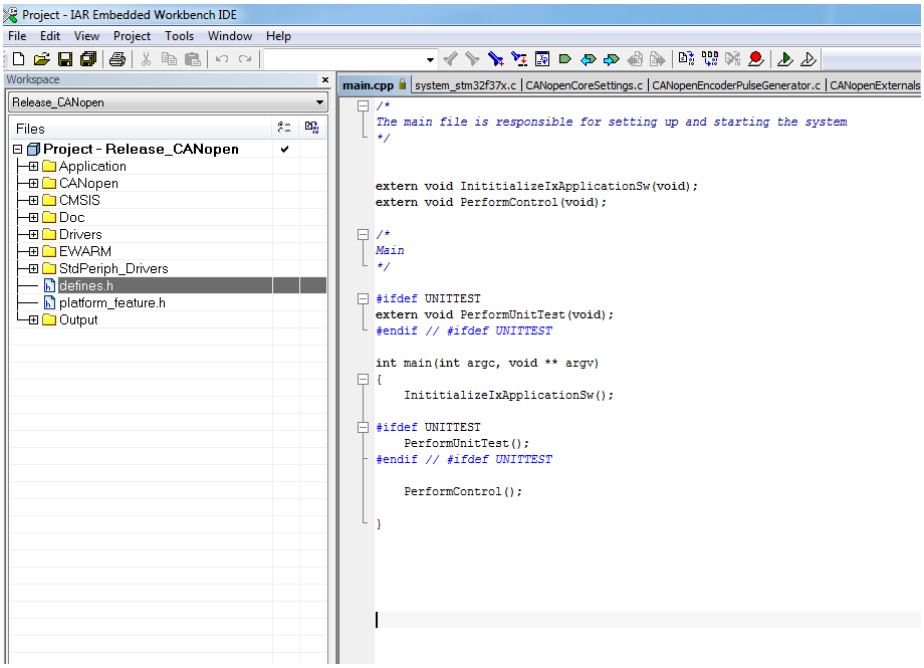
10.1. Opening and building a project

Open IAR Embedded Workbench 7.40 or later

Select File/Open Workspace

The workspace for CrossFireIX is located into the Project/IAR folder

After opening the workspace it will look like the following:



To build the project select Project/Make.

If you have a debug probe connected you can select Project/Download and debug. The executable will then be downloaded to the unit and started. You might need to change the debugger properties in Project/Options/Debugger.

In case you want to rebuild everything select Project/Rebuild all.

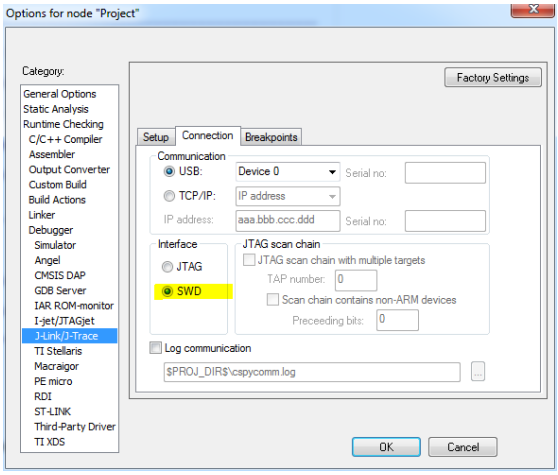
To change project options go to Project/Options.

For the CANopen example there are two project configurations: Release_CANopen and Debug_CANopen. The release config is used for release builds. The debug configuration can be used while debugging to get additional debug information. The project configuration to use is selected in the drop-down box above the project files list.

10.2. Function profiling

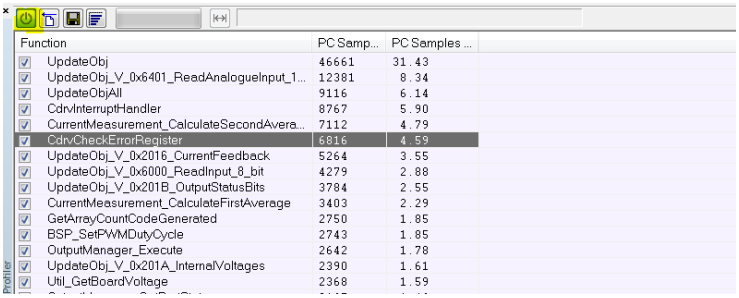
It is highly recommended to use the IAR “Function profiler” to check which code that takes most processor resources. Profiling can be done over SWD using “sampled trace”. This can be done using a normal I-Jet debugger; a real trace debugger is not needed for sample mode.

Activate SWD



Select J-Link/Function Profiler from the menu

Press the On button in the Function Profiling window. This must be done in break mode (Select Debug/Break) if system is running



Function	PC Samp...	PC Samples ...
UpdateObj	46661	31.43
UpdateObj_V_0x6401_ReadAnalogueInput_1...	12381	8.34
UpdateObjAll	9116	6.14
CdrvInterruptHandler	8767	5.90
CurrentMeasurement_CalculateSecondAvera...	7112	4.79
CdrvCheckErrorRegister	6816	4.59
UpdateObj_V_0x2016_CurrentFeedback	5264	3.55
UpdateObj_V_0x6000_ReadInput_8_bit	4279	2.88
UpdateObj_V_0x201B_OutputStatusBits	3784	2.55
CurrentMeasurement_CalculateFirstAverage	3403	2.29
GetArrayCountCodeGenerated	2750	1.85
BSP_SetPWMDutyCycle	2743	1.85
OutputManager_Execute	2642	1.78
UpdateObj_V_0x201A_InternalVoltages	2390	1.61
Util_GetBoardVoltage	2368	1.59

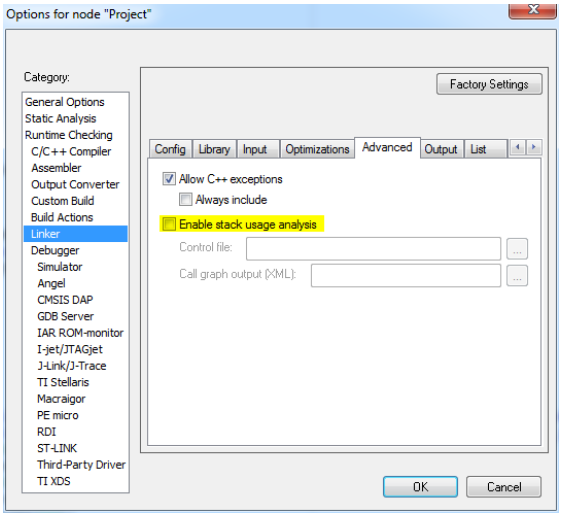
Select “sampling mode” by right clicking in the Function Profiling Window and select (source: sampled).

Start the system again by selecting Debug/Go (F5)

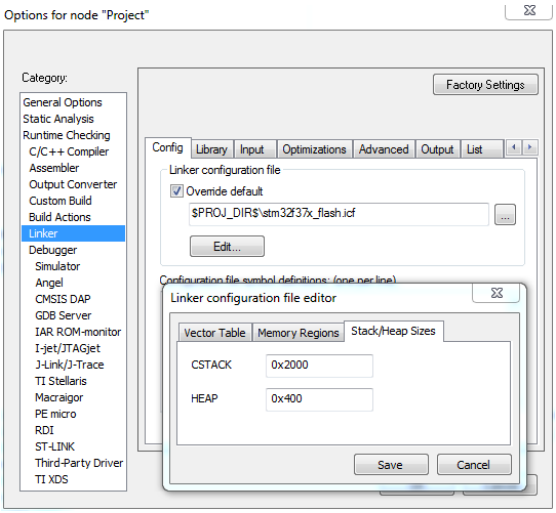
You can now sort the list by clicking the PC Samples column

10.3. Stack

As the CrossFire IX stack size is limited, make sure to check that the stack is not overwritten. There is a good function in the IAR compiler to generate a stack analysis log.



The stack and heap size can be changed in the Linker configuration file editor if needed



11. Programming the CrossFire IX

11.1. General CrossFire IX programming recommendations

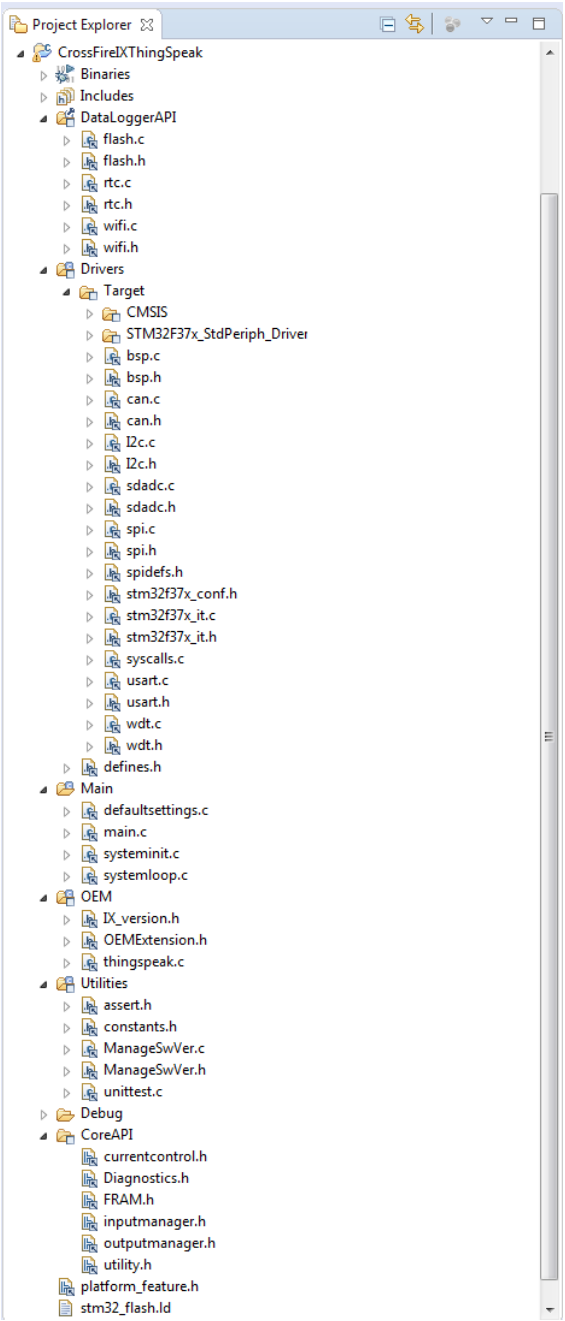
Do not use dynamic memory management (malloc, new etc). This is to avoid memory problems in runtime.

Do not put large objects on the stack; the stack size is limited due to limited RAM.

Do not write functions that takes a long time to execute (> 1ms) and call from the OEM_Execute(). Doing so will starve out other important things done in the main loop. Code that takes a long time to execute should be divided into smaller parts.

11.2. Project structure

The standard CrossFire IX Atollic TrueSTUDIO project structure looks like the following:



- Binaries and Includes is folders generated by Atollic TrueSTUDIO.
- DataLoggerAPI contains source and header files for the data logger functions. Only applicable for the Data Logger Edition.
- Drivers contain all CrossControl drivers, ST Std Peripheral Drivers, interrupt handlers and CMSIS (Cortex Microcontroller Software Interface Standard) drivers.
- Main contains main, systeminit, systemloop and default settings.
- OEM Contains the files the OEM is responsible for.
- Utilities contain various utility files.
- The Debug folder is generated by Atollic TrueSTUDIO.

- CoreAPI contains header files for the Core API. Core API is delivered as a library so no source files are available.
- platform_feature.h contains defines for which features are available on which PCB version.
- stm32_flash.ld is the linker file defining how the linker will use the FLASH and RAM memories.

11.3. CrossFire IX Core API

CrossFire IX Core API is the API that is used by the OEM when building an application on the CrossFire IX. The Core API is hardware independent and built itself upon the drivers layer. The OEM should not call functions directly in the drivers layer (except for the CAN driver and WDT) but only through the Core API. Code for the drivers layer is supplied for reference.

The CrossFire IX Core API contains the following modules:

Module	**Header File**
Input control	inputmanager.h
Output control	outputmanager.h
CurrentControl for outputs (PWMi)	currentcontrol.h
Diagnostics	Diagnostics.h
FRAM (Persistent Storage)	FRAM.h
Utility functions (LEDs, Boot, Ignition, board temp)	utility.h

Most parts of the CrossFire IX firmware are delivered in source code form. However, the CrossFire IX Core API is delivered in library form. There are two versions of the libraries, one release and one debug version. The debug version ends with a D.

There is a complete description of the CrossFire IX Core API available in Doxygen format.

11.4. Pre-processor defines

There are a number of pre-processor defines used in the code. The following table describes their use.

The following defines should NOT be changed by the OEM.

Define	**Description**
USE_STDPERIPH_DRIVER	Enable ST Std peripheral drivers
STM32F37X	Define processor family
TARGET	Compile for target hardware
STM32F10X_MD	Define processor type as a medium density processor
HSE_VALUE=16000000	Define crystal frequency

STM32F373xC	Define processor family

The following defines can be changed by the OEM.

CCIX/LOGGER_EDITION	Should be set according to board type. CCIX for the CANopen version of the board and LOGGER_EDITION for the logger edition of the board.
DEBUG /NDEBUG	Compile for DEBUG or RELEASE (NDEBUG = Not Debug)
CANopen_BUILD	Build with SYS TEC CANopen support
USE_OUTPUT	Activate support for outputs
USE_INPUT	Activate support for inputs
USE_DIAG	Activate support for diagnostics
USE_IGNITION	Activate support for ignition signal
FRAM_RANGE_CHECK	Test that the FRAM is used within range. Normally only active for debug.
USE_PWMI	Activate support for output 4-8 (PWMi outputs)
CANRXBUFFERSIZE	Set up the size of the CAN rx buffer used in can.c

There are also some defines used for debugging by CrossControl as well as a number of defines used by the SYS TEC CANopen stack. Please read the SYS TEC documentation for details about the SYS TEC defines.

11.5. System Init

System initialization is made in the systeminit.c file. The main function for initialization is the InititializeIxApplicationSw() function. This function calls a number of different helper functions to initialize different parts of the hardware. In the end of the init phase the function OEM_Init() is called. Here, the OEM can add its own initialization code.

Before the OEM_Init is called, the function PerformControlInit() is called. This function enables the watchdog and activates inputs and outputs. The idea is that the outputs and inputs should not be active before the watchdog is activated.

11.6. Main Loop

CrossFire IX does not use an operating system; instead the execution is driven by a main loop and interrupts.

The main loop is called PerformControl() and is located in the systemloop.c. PerformControl will never exit but will run until the unit is reset or shut down. PerformControl is intended to run cyclic tasks with low priority. Note that PerformControl can be interrupted by an interrupt at any time as long as interrupts are not disabled.

A very simple scheduler is used to drive tasks that do need to run at a regular time interval. The scheduler is run in the function TickScheduleTimerMS(). TickScheduleTimerMS is called every ms from the SYSTICK interrupt setting flags that tell the PerformControl loop that it is time to execute a certain function.

Every loop turn the function OEM_Execute() will be called. Here the OEM can add code that should be executed cyclically. It is not recommended to change directly in the function PerformControl.

Every millisecond also the function OEM_MSTick() is called. This makes it possible for the OEM to keep track of the elapsed time to be able to perform cyclic tasks.

11.7. Interrupts

Interrupts are handled in the stm32f37x_it.c file located in the drivers folder. It is possible for OEM to change the interrupt handlers but this must be done with great care.

CurrentControl for PWMi and input filtering is driven by interrupts. These interrupts take relatively long time to execute. It might be necessary to create interrupts with higher priority than these if quick response time is needed. However, such interrupts must be very fast to not disturb the PWMi regulation and the input filtering.

Do not access shared resources from interrupts as this might cause re-entrance problems!

STM32F373 uses 4 bits for interrupt priority and sub-priority. It is possible to decide how many bits will be used for priority and how many bits will be used for sub-priority according to the table below. CrossFire IX uses group 2 which means that 4 levels of interrupt priority and 4 levels of sub-priority is available.

```
The table below gives the allowed values of the pre-emption priority and subpriority according
 to the Priority Grouping configuration performed by NVIC_PriorityGroupConfig function
  ==================================================================================================================
    NVIC_PriorityGroup   | NVIC_IRQChannelPreemptionPriority | NVIC_IRQChannelSubPriority  | Description
  ==================================================================================================================
   NVIC_PriorityGroup_0  |               0                  |            0-15             |   0 bits for pre-emption priority
                         |                                  |                             |   4 bits for subpriority
  ------------------------------------------------------------------------------------------------------------------
   NVIC_PriorityGroup_1  |               0-1                |            0-7              |   1 bits for pre-emption priority
                         |                                  |                             |   3 bits for subpriority
  ------------------------------------------------------------------------------------------------------------------
   NVIC_PriorityGroup_2  |               0-3                |            0-3              |   2 bits for pre-emption priority
                         |                                  |                             |   2 bits for subpriority
  ------------------------------------------------------------------------------------------------------------------
   NVIC_PriorityGroup_3  |               0-7                |            0-1              |   3 bits for pre-emption priority
                         |                                  |                             |   1 bits for subpriority
  ------------------------------------------------------------------------------------------------------------------
   NVIC_PriorityGroup_4  |               0-15               |            0                |   4 bits for pre-emption priority
                         |                                  |                             |   0 bits for subpriority
  ==================================================================================================================
```

Interrupt	Description	Priority (priority/sub priority). Lower number means higher priority.
DMA from ADC	Used for current measurement for current control (PWMi).	1/0
SPI2	A3942 Gate Driver	1/0

SPI3	Shift registers for port config	1/0
PVD	Programmable Voltage Detector interrupts to detect low supply voltage	0/0
EXTI4	External Interrupt to handle ignition	0/0
CAN1RX/TX/SCE	Interrupts for CAN communication	0/0
SDADC	Interrupts for handling the sigma delta ADC converter (SDADC) used for analog inputs	3/0
Timer 5	Interrupt to drive input sampling/filtering	3/0
SYSTICK	Interrupt is used for timing purposes	

11.8. Using the Watchdog

To make sure that the unit does not lock up in a dangerous state in case of a software error, the hardware watchdog is used. If doing a lengthy operation, calling reload on the watchdog might be necessary. However, in most cases it is better to rewrite the code to avoid the lengthy operations. Lengthy operations do not only trigger the watchdog, they also reduce the responsiveness of the device.

Name	**Description**
`void WDT_Reload(void);`	Reload the Watchdog.

11.9. Performance considerations

In case inputs or outputs are not used at all, it is possible to optimize performance by removing the pre-processor defines USE_INPUT and/or USE_OUTPUT. By removing these defines, code that handles inputs and/or outputs are not executed resulting in a faster execution. If outputs are used but not the PWMi outputs (4-8) the define USE_PWMI can be removed.

For solving performance problems, using function profiling using SWD is highly recommended.

11.10. Version number

It is highly recommended to keep an updated version number. The version number is manually entered into IX_version.h. The version entered here will automatically be placed at a fixed location in the FLASH memory, which makes it possible also for the bootloader to read the version.

For the CANopen slave version the version number is also mapped into the Object Dictionary so it is possible to read the version number over CAN.

11.11. Diagnostics

The CrossFire IX Core API contains a diagnostics module. This module is responsible for writing diagnostics data to FRAM. The diagnostics data can help CrossControl in case of a problem. To make diagnostics work, the function DIAG_Cyclic() must be called cyclically from the main loop. The function DIAG_SetTick() must be called every ms. The function DIAG_Init() must be called at start up. The calls to these functions are already added to systeminit/systemloop so the OEM does not need to do anything. It is possible to disable to use of diagnostics by removing the pre-processor define USE_DIAG. However, it is highly recommended to keep the diagnostics functionality.

12. Examples

Note that all examples need additional testing and error management to be used as products.

12.1. The Basic I/O example

The Basic I/O example shows how to configure input and output I/O. Measure values are sent as user defined CAN messages. By setting the #define TEST_ENCODER inputs 13, 14 are used in combined encoder mode instead of in frequency mode. This is the most basic example available and a good starting point for writing CrossFire IX applications.

This example sets up I/O according to the following table:

Output I/O

0	Digital
1	Digital
2	PWM 50Hz
3	PWM 400Hz
4	PWMi
5	PWMi
6	PWM
7	Digital

Input I/O

0	0-32V
1	0-32V
2	0-32V
3	0-32V
4	Current (4-20mA)
5	Current (4-20mA)
6	Current (4-20mA)

7	Current (4-20mA)
8	Digital/Pull Up
9	Digital/Pull up
10	Digital/Pull down
11	Digital/Pull down
12	Freq/Pull Up
13	Freq/Pull Up

NOTE! As inputs 8-11 is always digital there is no need to explicitly set the mode.

The example also sets up input filtering to the following:

Sampling frequency 50Hz.

0	Average of 10 measurements
1	Average of 50 measurements
2	25% forgetting filter

12.2. The CANopen slave example

NOTE! This example is only available for customers that have signed an NDA.

The CANopen slave example is actually a complete CANopen slave, basically the same code as is included in the CANopen slave version of the CrossFire IX. This means that the example is quite complex, however it shows all functionality that is needed to build a CANopen slave and is also possible to extend to get a customer version of a complete CANopen slave.

The CANopen slave example is based on the SYS TEC CANopen stack. The SYS TEC CANopen stack normally uses a file called target.c to set up the processor (gpio, sysclk etc). As the CrossFire IX can run completely without the SYS TEC CANopen stack, CrossFire IX does not use this file to set up the processor. Processor setup is done in CrossControl driver and bsp files. However, there are some functions for memory management and CAN interrupts that are used from target.c. The CrossFire IX specific version used is called target_ix.c.

The main file for init and execution of CANopen functions is the CANopenMAIN.c file. This file is derived from the slave example from SYS TEC. This file contains the function CANopenMAIN_Init() that initializes the stack and the function CANopenMAIN_Execute() that is called from the main loop to execute the stack. It also contains the function AppCbNmtEvent() that is called by the stack in case of an NMT (NodeManagementT) event.

For persistent storage of CANopen parameters the file tgtcav.c is used. This file is delivered as a generic template from SYS TEC and has been adapted to the CrossFire IX by CrossControl

The CAN driver cdrvbxcan.c from SYS TEC is used as CAN driver. The driver is slightly modified by CrossControl to work for the CrossFire IX.

The connection between the CANopen stack and the CrossFire IX Core API is done in the file CANopenIntegration.c. Even if the SYS TEC CANopen stack is not used, CANopenIntegration.c can be seen as an example of how the Core API is used.

The CANopen slave example uses a specific define “CANopen_BUILD”. This define will activate some additional code mainly in systeminit.c and systemloop.c.

Details about the CANopen-OD/ObjectDictionary-build-process can be read in the CrossFire IX CANopen - SADD.docx.

13. Testing

There are a number of test functions included in the freely programmable SDK. By setting the define UNITTEST the function PerformUnitTest will be executed after system init. In this function a number of test functions are called. The test functions will use printf to give user feedback so it is important that printf calls are redirected to a console window to see the output. It is possible to comment out the test functions that you do not want to run.

The available test functions are:

Name	**Description**
I2C_FRAM_SpeedTest	Test read/write performance of the I2c FRAM
CAN_SendTest	Test to send 100000 CAN messages at 1000 kbit/s as fast as possible.
CAN_ReceiveTest	Test to receive 10000 CAN messages at 1000 kbit/s.

It is highly recommended to run these test functions before a release and the OEM is encouraged to add their own test functions that can be run in the same way.

It is recommended to use static code analysis. Cppcheck is a free tool (http://cppcheck.sourceforge.net/) that has been used on the CrossFire IX by CrossControl. Cppcheck can be run stand alone or as a plugin in Atollic TrueSTUDIO/Eclipse.

14. Using the CrossFire IX Hardware

14.1. Hardware resources

The following hardware resources are used in CrossFire IX.

The OEM should normally not use these HW resources directly, although it is possible to use for instance the PWM generation timers for other purposes if PWM outputs are not used.

Resource	**Usage**
Timer 17 Timer 19 Timer 16 Timer 12 Timer 15 Timer 13 Timer 4	PWM generation for outputs

Timer 14	
Timer 5	Input filtering
SDADC1 SDADC2	Analogue inputs
ADC1	Analogue measurements (excluding inputs)
DMA1 CH1	ADC measurements used for PWMi
PVD	Low supply voltage detection
SPI2	SPI to gate driver A3942 driving the PWMi capable outputs
SPI3	Shift registers for input configuration
I2C1	Temp sensor/FRAM
CAN1	CAN
EXTI4	Ignition
SYSTICK	General timing
DMA1CH5 DMA1CH7 Timer 2	Frequency /Encoder Inputs
DMA1CH2 DMA1CH3	SPI-FLASH RX SPI-FLASH TX

Timer 3 is reserved for future use in the Core API.

Timer 6, 7 and 18 is free to use for the OEM.

14.2. Using the CrossFire IX I/O

The I/O of the CrossFire IX is available through the Core API. Generally, the I/O configuration should be set up in OEM_Init(). However, it is also possible to change the configuration "on the fly" if needed. I/O is generally controlled from the OEM_Execute() function. Make sure to call the function `OutputManager_ApplySettings(void);` after changing output mode or pwm frequency.

Most I/O functions use very little CPU power. For instance PWM is completely generated in hardware. The most processor intensive I/O task is PWMi control. PWMi control is done in software so do not activate PWMi if not needed. Also inputs will require some CPU power if using input filtering. It is highly recommended to use function profiling to see how much CPU power is used for different functions.

This is a list of the most common functions for controlling the I/O. A complete list of functions is available in the Doxygen documentation for Core API.

Outputs	
`BOOL OutputManager_SetMode(outputchannel channel, controlModes mode);`	Set the mode for an output channel
`void OutputManager_SetPWMDutyCycle(outputchannel channel, unsigned short dutyCycle);`	Set the PWM duty cycle for an output in PWM mode
`void OutputManager_SetPWMFrequency(outputchannel channel, unsigned short frequency);`	Set the PWM frequency in Hz for an output in PWM mode
`void OutputManager_SetOn(outputchannel channel, BOOL value);`	Set on/off for an output in digital mode
`void OutputManager_ApplySettings(void);`	Apply setting changes. This function must be called when changing port mode or PWM frequency
`portStatus OutputManager_GetPortStatus(outputchannel channel);`	Get port status for an output channel
`void OutputManager_Retry(outputchannel channel);`	Try to restart a channel that has been shut off due to port error
`signed short OutputManager_GetAverageOutputCurrent(outputchannel channel);`	Get the output current for an output channel averaged during a longer period. Only valid for PWMi capable channels.
`unsigned short OutputManager_GetHS14CurrentWithErrorDetection(outputchannel channel);`	Get current feedback for high side output 1-4
PWMi	
`void CurrentControl_SetCurrentReference(outputchannel channel, unsigned short current);`	Set the current reference for an output channel
`void CurrentControl_SetDitherAmplitude(outputchannel channel, unsigned short amplitude);`	Set the dither amplitude for an output channel
`void CurrentControl_SetDitherFrequency(outputchannel channel, unsigned short frequency);`	Set the dither frequency for an output channel
Inputs	
`BOOL InputManager_SetAnalogMode(AnalogInputChannel channel, AnalogInputMode mode);`	Set analog mode for an analog input
`BOOL InputManager_GetDigitalInput(inputchannel channel);`	Get value of input in digital mode
`float InputManager_GetAnalogInput(AnalogInputChannel channel);`	Get value of input in analog mode
`BOOL InputManager_SetDigitalInBiasMode(DigitalInputChannel`	Set bias mode for a digital

`channel, DigitalInputBiasMode mode);`	input channel
`void InputManager_SetAnalogInFilterParameters(AnalogInputChannel channel, unsigned char filterLengh, unsigned char weightForgettingFilter);`	Set Analog in filter parameters
`void InputManager_SetInputSamplingFrequency(unsigned short frequencyHz);`	Set sampling frequency for analog inputs
`unsigned char InputManager_GetOverCurrentProtectionStatus(void);`	Get status of which inputs is in over current protection mode
Frequency/Encoder	
`BOOL InputManager_SetFrequencyInMode(FrequencyInputChannel channel, FrequencyInMode mode);`	Set Frequency Input mode (digital, encoder, freq). NOTE!!! You must also call InputManager_SetFrequencyInBiasMode after this call to make settings take effect!!!
`BOOL InputManager_SetFrequencyInBiasMode(FrequencyInputChannel channel, FrequencyInBiasMode mode);`	Set Frequency Input bias mode
`unsigned long InputManager_GetEncoderInput(FrequencyInputChannel channel);`	Get encoder value for inputs in encoder mode. For encoder mode two inputs must be combined. Both channels will return the same encoder value.
`float InputManager_GetFrequencyInputFrequency(FrequencyInputChannel channel);`	Get frequency for frequency input in frequency mode
`void InputManager_ResetEncoderValue(void);`	Reset the encoder value
`void InputManager_SetMinFreq(FrequencyInputChannel input, float freq);`	Configure the min frequency for a frequency input. Using a low min frequency means that it takes a longer time to detect a 0 Hz signal.

14.3. Using the FRAM memory

The CrossFire IX PCB contains an 8KB FRAM memory that can be used to store persistent data. It is very important to not use parts of the FRAM memory that is used for other purposes. Please note that the FRAM memory is relatively slow so avoid writing a lot of data to the FRAM continuously. Only write to the FRAM from the main loop (not from interrupts) to avoid re-entrance problems.

FRAM Range	Purpose
0x0-0x5FF	Reserved for use by CrossControl. In this area calibration values, production data, diagnostic data and other important data is written. Writing to this area might cause the unit to malfunction!

0x600-0x1FFF	Available for the OEM. However, parts of this area is used for persistent storage of CANopen OD if the CANopen slave stack is used.

There is a test function for the I2C FRAM (I2C_FRAM_WriteTest) included in the I2c driver giving the following data:

```
Performing I2C FRAM test
I2C FRAM Write speed: 31 KB/s
I2C FRAM Read speed: 28 KB/s
I2C FRAM test done
```

This is a list of the most common functions for FRAM. A complete list of functions is available in the Doxygen documentation for Core API.

`unsigned char FRAM_Write(unsigned short address, unsigned char data);`	Write a byte to FRAM
`unsigned char FRAM_WriteBytes(unsigned short address, unsigned char *data, unsigned long len);`	Write a number of bytes to FRAM
`unsigned char FRAM_WriteBytesUserArea(unsigned short address, unsigned char *data, unsigned long len);`	Write a number of bytes to FRAM user area
`unsigned char FRAM_ReadBytes(unsigned short address, unsigned char *data, unsigned long len);`	Read a number of bytes from FRAM

14.4. CAN driver

There are two CAN drivers available for CrossFire IX. There is one CAN driver defined in can.c available in the drivers folder. This is a generic CAN driver recommended in most cases. There is also the SYS TEC cdrvbxcan.c that fully integrates with the SYS TEC CANopen stack. Some code for system setup is used from can.c even if the cdrvbxcan.c is used.

This is a list of the most common functions for CAN. A complete list of functions is available in the Doxygen documentation for the drivers.

`void CAN_Config(CANBaudRates baudRate, CANmodes mode);`	Initialize CAN. Note that filter mask = 0 is set to accept all messages including rtr and extended frames.
`BOOL CAN_Recv(CanRxMsg *msg);`	Get CAN message from receive buffer
`unsigned char CAN_Send(CanTxMsg *msg);`	Send a CAN message
`BOOL CAN_SetMessageFilter(unsigned char filterNum, unsigned long filterId, unsigned long maskId, BOOL rtr, BOOL ide);`	Activate a CAN message filter

14.5. ADC

The ADC is used in scan mode with DMA. This means that the ADC is automatically switching between a list of ADC channels writing the result to an array. It is very important that the OEM does not use the ADC itself (disturbing the ADC scanning) but instead read values through the Core

API. The scanning is set up in void BSP_ADC_Config(void). The values are written to g_ADC1_data. The OEM should only get data through the Core API.

14.6. Ignition

CrossFire IX supports control by an ignition signal. The supervision of the signal is done completely in software. By calling the function Util_CheckIgnition() from the main loop, the ignition signal will be checked and the processor will be set into sleep mode in case the signal is low. If the ignition signal goes high again, the processor will resume and the software will make a reset to make the CrossFire IX start from scratch. The reason to check for the ignition signal from the main loop instead of from an interrupt is to get better control over when suspend mode is entered. For instance we want to have the chance to write data to FRAM before suspend is entered.

It is possible to disable to use of ignition by removing the pre-processor define USE_IGNITION.

14.7. Utility Functions

There are also a number of utility functions available. These are the most common:

`void Util_SetGreenLED(BOOL enable);`	Set the green LED.
`void Util_SetRedLED(BOOL enable);`	Set the red LED.
`void Util_JumpToBoot(void);`	Jump to bootloader to make it possible to do a firmware upgrade.
`void Util_WriteApplicationOKtoFRAM(void);`	Write application OK to FRAM. This function should be called when application is up and running to let the bootloader know the application is working properly.
`unsigned short Util_GetBoardVoltage(boardVoltage voltage);`	Get board voltage.
`void Util_Set12vPower(BOOL enable);`	Set the 12v sensor supply.
`BOOL Util_GetBoardTempDegC(signed short *value);`	Get the board temperature.
`void Util_SleepMS(unsigned long timeoutMS);`	Sleep a number of ms. This function uses the systick 1 ms interrupt to get an exact delay.
`void Util_SleepUS(unsigned long timeoutUS);`	Sleep a number of us. This function uses a simple for loop to get a delay without the need for the systick or other interrupt.
`BOOL Util_IsTimeout(unsigned long startTimeMS, unsigned long timeoutMS);`	Check for timeout
`unsigned long Util_GetTimeMS();`	Get the current ms counter. This value will wrap around but that is no problem if using relative values.

15. Tools

There are three tools included in the CrossFire IX freely programmable SDK:

- CrossFire IX Tool
- CrossFireIX Tool CANopen
- SakNfo Tool

The CrossFire IX Tool is used for firmware upgrade and for building firmware packages that can be sent over Wi-Fi (FOTA). The CrossFire IX Tool is located in the folder Tools\CrossFire IX Tool \bin_DN4\Release.

The CrossFire IX Tool CANopen is used for firmware upgrade and test for the CrossFire IX CANopen slave version. CrossFire IX Tool CANopen is located in the folder Tools\CrossFire IX Tool CANopen\CrossFire IX Tool\bin_DN4\Debug.

There is also the SakNfo (Sak=SwissArmyKnife for handling Info=Nfo) tool that is used to generate a CANopen configuration (basically an OD). The configuration is changed by modifying some source files and rebuilding the tool. The SakNfo tool is built using MS Visual Studio 2010 or later. More information about using this tool is found in the CrossFire IX CANopen - SADD.docx.

15.1. Installing CrossFire IX Tools

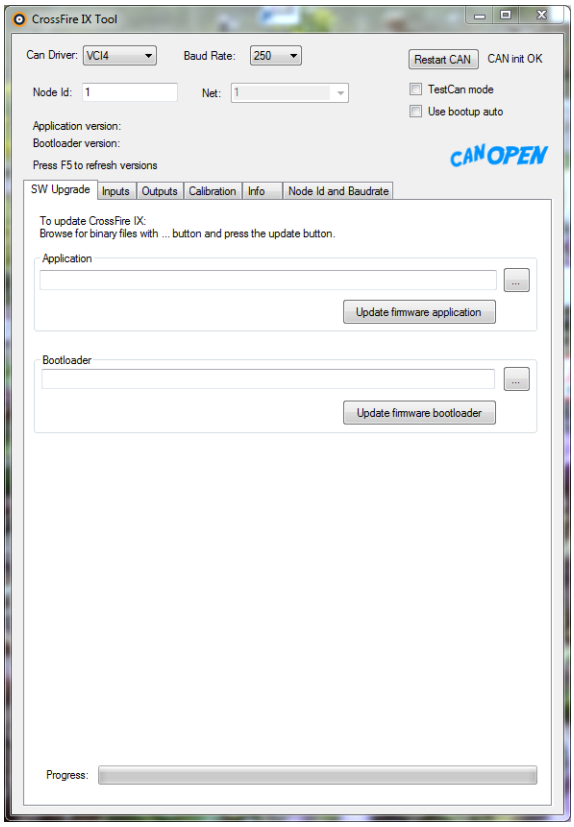
The following equipment is needed to run the CrossFire IX Tool:

- PC running Windows 7/8/10 32 or 64-bit
- IXXAT USB-to-CAN adapter
- CAN cabling for CrossFire IX with proper termination (120 ohm resistor at each end of the cable).

To run the tools:

- Install the IXXAT VCI4 drivers. The drivers can be downloaded from https://www.ixxat.com/support
- Install .NET4 runtime. The runtime can be downloaded from https://dotnet.microsoft.com/download/dotnet-framework-runtime
- There is no installation needed for CrossFire IX Tools. Just run the .exe file.

15.2. Using the CrossFire IX Tool for CANopen



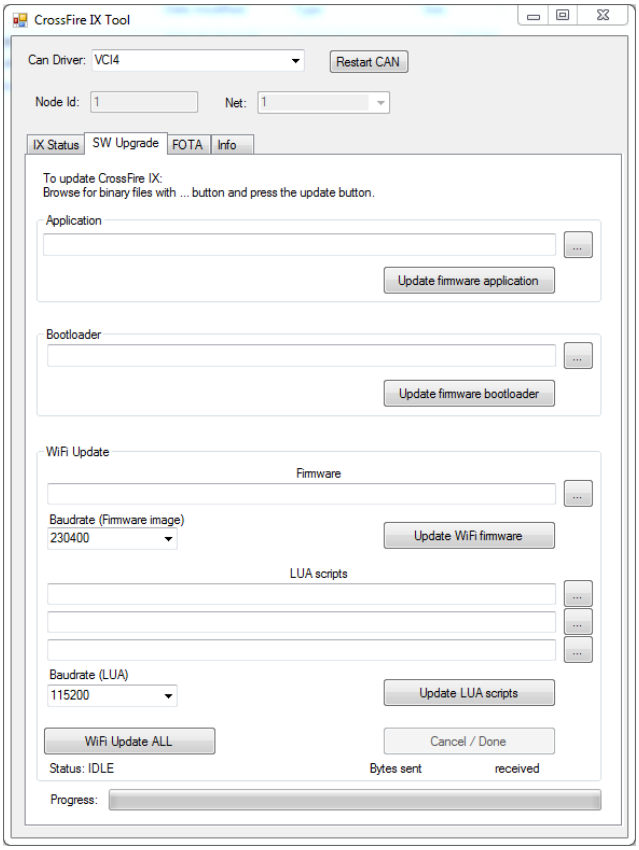
The CrossFire IX tool for CANopen is intended to run against the CANopen slave example. As the tool uses CANopen messages to perform software upgrade, it is not working against the “Data Logger Edition” examples.

To update the CANopen slave main application, browse for the application binary by pressing the “…” button and press “Update firmware application”.

To update the CANopen slave boot loader, browse for the boot loader binary by pressing the “…” button and press “Update firmware bootloader”.

15.3. Using the CrossFire IX Tool (not CANopen version)

The IX Status and Info tabs do only work if unit is built with support for the “testcan” protocol normally used in production test. The FOTA tab is used to build a firmware package that can be downloaded over Wi-Fi. Wi-Fi update is only possible if FOTA support is added to the CrossFire IX/ESP8266 firmware.



15.3.1. Upgrading the firmware of the CrossFire IX over CAN

The tool will automatically connect to the CAN bus at 1000 kbit/s when started. If this does not work, make sure the VCI4 drivers are installed properly, the USB-to-CAN adapter is connected and the no other application is using the CAN interface. Make sure that the CrossFire IX is connected to the USB-to-CAN adapter and that the CAN-cable is properly terminated (120 ohm resistor at both ends). Also make sure the CrossFire IX is in boot loader mode. How to get to the bootloader mode is dependent on the application running on the unit. The CrossFire IX Tool will send an id 0x555 CAN message to the unit to make it to go to boot loader mode. In some cases it is necessary to activate a digital input at start up to make the unit listen for 0x555 messages. This is done in for instance in the CAN to Wi-Fi gateway example to make sure the unit does not go to boot loader mode in case a 0x555 messages is sent by coincidence. Read the documentation for the software that is currently on the module.

To update the CrossFire IX main application, browse for the binary from the Application group box and press “Update firmware application”.

To update the CrossFire IX boot loader, browse for the binary from the Bootloader group box and press “Update firmware bootloader”. Make sure to not abort upgrade of the bootloader as this can cause the unit to not be able to boot.

To update the ESP8266 Wi-Fi slave processor, browse for the binary from the Wi-Fi Update/Firmware group box and press “Update WiFi firmware”. Baud rate should normally be 230400.

NOTE! The LUA update functions are not needed if you are not using the NodeMCU/LUA firmware on the ESP8266 which is not recommended!

16. References

CrossFire IX - Freely Programmable - Data Logger Edition - Programming Manual.docx

CrossFire IX - Technical manual.docx

CrossFire IX - CANopen Slave Developers Guide.docx

CrossFire IX CANopen - Firmware upgrade instructions.docx

CrossFire IX CANopen - SADD.docx

https://support.crosscontrol.com/

https://www.ixxat.com/support

https://atollic.com/

http://cppcheck.sourceforge.net/

https://www.systec-electronic.com/

https://www.iar.com/

17. Trademark, etc.